Stream Ciphers: Striving for Randomness

Carl Schünemann	Simon Thalmaier	Larysa Bondar
00107827	*00108692*	*00105168*
ch. $1-3$	ch. $4-5$	ch. $6-8$

2022
20th of July

Contents

1 Introduction

Random numbers are an important component of modern cryptography. In the case of encryption, they reduce redundancies in encrypted plaintexts, resulting in more secure transmission. [1, p. 107] The underlying issue is the difficulty of generating random numbers. In his paper *Various Techniques Used in Connection With Random Digits*, John von Neumann already warned in 1951 of this non-trivial task: "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." [2, p. 36] A cryptographic system that addresses this challenge is stream ciphers. The generation of random numbers has a central function in this concept. Stream ciphers are used if one symbol is to be encrypted or decrypted per time unit while using minimum resources. [3, p. 191]

This paper presents efforts to approximate randomness with stream ciphers. In particular, the difficulty of this attempt will be illustrated.

Initially, the first chapter introduces the central idea of stream ciphers. In the second chapter, the most traditional approach for constructing stream ciphers with shift registers is discussed mathematically. Expanding on this, the third chapter highlights the difficulties associated with this approach. Efforts to improve this idea and alternative solutions are analyzed in the fourth chapter. Finally, a recent competition is presented in the fifth chapter, aimed at determining the best of all participating stream ciphers in order to eliminate the previous difficulties.

2 The Idea of Stream Ciphers

Stream ciphers are one of two major symmetric encryption methods. They serve a different purpose than block ciphers, the second symmetric encryption method. The primary intent is to encrypt and decrypt messages approximately synchronously between the sender and receiver of a message. To achieve this, the message is not first divided into blocks and pre-processed before it is ciphered, as is the case with block ciphers. Instead, stream ciphers encrypt or decrypt the plaintext or ciphertext directly. [4, p. 223]

A prerequisite for this is a theoretically infinite, ideally truly random binary sequence. To encrypt the data, the sender combines the plaintext bitwise exclusive-or (XOR) with the random sequence. The recipient decrypts the ciphertext by also combining it bitwise exclusive-or with the same random sequence used by the sender.

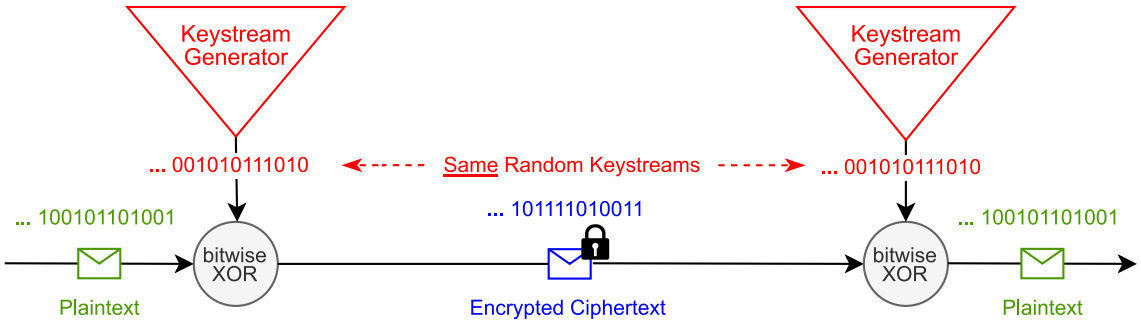


Figure 1: Stream cipher as a symmetric encryption method, based on [4, p. 232]

This only works if both communication partners have the same random sequence. This is a characteristic of symmetric encryption. [5, pp. 319-320] This problem could be trivially solved by the sender first generating a true random sequence of sufficient length and transmitting it to the recipient. However since this key sequence can also be seen as plaintext which must be transmitted securely, the situation results in a stream cipher with exactly the same problem as before.

2.1 ”True Pseudorandomness”

The central question that arises is: How can truly random numbers be generated by means of a computer, based on a short key? [6, p. 53] Without starting a philosophical discussion, it is necessary to define what characterizes a truly random bit sequence.

This can be described using an analogy to a Laplacean experiment, like the sequences of fair coin tosses: Within this sequence, the values 0 and 1 each occur with probability 0.5. In addition, there is no way to derive information about the rest of the sequence from knowledge of an arbitrarily long initial piece of the sequence. To predict the next bit, an attacker must thus have no better chance of success than 0.5. To achieve equal distribution of the bits, the experiment must be conducted theoretically for a long time. Combining a given a message M with this truly random sequence XOR results in a truly random ciphertext C. Based on this information, a so-called *perfect cipher system* is characterized by the fact that the a priori probability $P(M)$ is equal to the a posteriori probability $P(M \mid C)$ resulting in $P(M \mid C) = P(M)$. [7, pp. 52-23]

The real question is in fact: Can a computer generate numbers that only look truly random? Due to the determinism of a PC, which can be represented as a finite state machine, truly random numbers can never be generated. It is only possible to generate deterministic, so-called *pseudorandom numbers* (PRN). After the input of one or more initialization numbers, a *pseudorandom number generator* (PRNG) generates this pseudorandom number sequence. A deterministic inner state is used for this purpose. [7, pp. 195-196] In the following chapters it will be shown that a stream cipher can be interpreted as a PRNG according to these criteria. As can be seen in Figure 2, a stream cipher has this inner state of the PRNG. Initially, it is filled by the short key that both communication partners possess. A stream cipher uses the key to generate and output the pseudorandom *keystream* or *running-key* bitwise or bytewise. [4, p. 233]

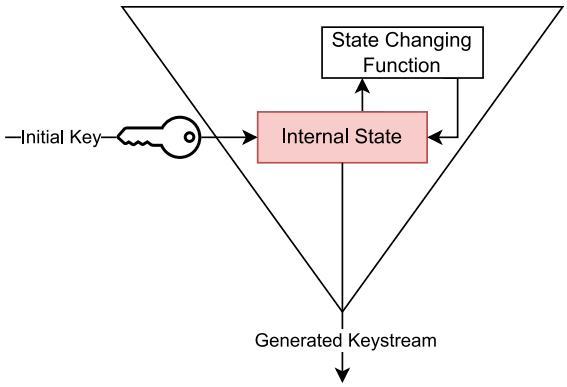


Figure 2: A stream cipher generates the keystream with the help of the initial key, using an inner state. Based on [4, p. 234]

Optimally, this generated bit sequence should have the same statistical properties as truly random bits. To estimate this statistical quality, a series of statistical tests exist. Pioneering for the evaluation of pseudorandom numbers were the *Golomb-postulates* formulated in the 1960s by the American mathematician Solomon W. Golomb. [8, p. 43] In order to keep the scope of this paper concise, the focus is not on this topic.

Ultimately, it should not be computationally feasible for an attacker to determine the prospective sequence. No matter how many bits of the keystream one has, the probability of guessing the next bit correctly should not be better than half.

3 Linear Feedback Shift Registers (LFSRs)

The underlying idea of stream ciphers is the generation of a pseudorandom bit stream. An important tool for this are so called *linear feedback shift registers (LFSRs)*. These kinds of shift registers are pseudorandom bit generators. This is by no means a new theory. Solomon Golomb wrote in 1967 in his book *Shift Register Sequences* that the approach to generate pseudorandom sequences using LFSRs had been researched for two decades. [8, p. 2] Today, this idea has been developed further, resulting in many, partially theoretical applications that use shift registers. One reason for its popularity is its mathematical interpretability which is discussed in more detail in this chapter.

3.1 Generating Periodic Numbers

A *register* is a logical unit that has a certain number of *memory cells*, each of which stores one bit of information. A set with k of these cells forms a register. In the literature, no uniform term is used for these atomic elements of a register. Synonymous terms for the memory cells are, for example, *memory elements* and *stages* [9, p. 81], *delay elements* [10, pp. 186-187], *tubes* [8, p. 27] or, more broadly, *bit sequence* [4, p. 429] or *bit vector* [7, p. 198]. This paper uses the term memory cell as defined by Nigel P. Smart. [11, p. 227]

The crucial factor of a *feedback shift register* (FSR) is the connection of the memory cells via a feedback function R. After each clock signal, the register in its regular configuration shifts the contents of each cell to the next. In the process, one bit is shifted out at one end of the register. At the other end, a memory cell is freed. The sequence of these shifted out bits forms the sequence generated by the FSR. The new bit in the cleared memory cell at the other end is calculated by the function R depending on the other bits. The shift direction differs in literature. In Smart and Schneier, for example, the shift is to the right [11, p. 227][4, p. 429] and in Lidl and Niederreiter to the left, which is adopted in this paper. The memory cell at the end where the content is shifted out has the least significant index 0. The cell for which a new bit is calculated per clock signal has index $k - 1$. [10, pp. 186-187].

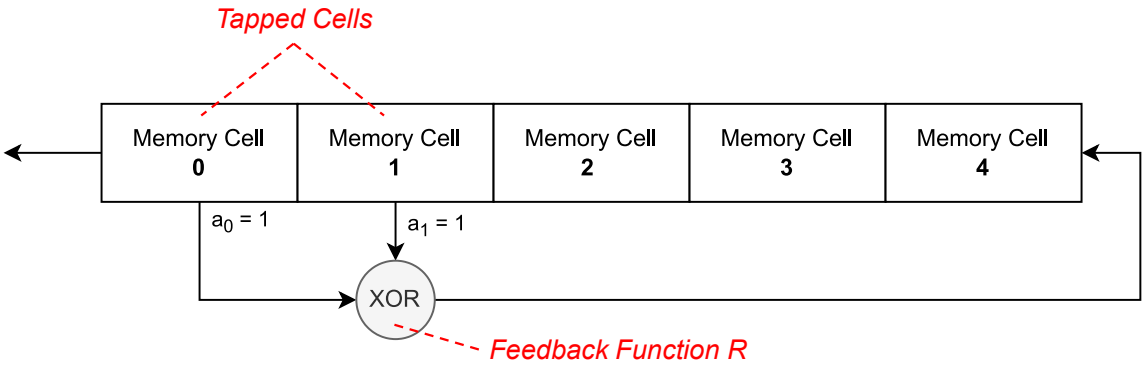


Figure 3: Basic concept of an LFSR of length $k = 5$. The linear XOR feedback function R combines the values of the tapped memory cells 0 and 1. Based on [4, p. 430]

The characteristic aspect of a *linear feedback shift register* (LFSR) is that the new bit $k - 1$ is calculated by a linear feedback function. That means, certain memory cells calculate the new value of the freed cell with the binary addition XOR. [9, p. 82] These memory cells whose values are used for this calculation are called *tapped* [11, p. 227]. Analogous to LFSRs, nonlinear FSRs use feedback functions, with nonlinear elements, such as binary multiplication. This concept is introduced in chapter 5.1. Further details can be found in Lidl and Niederreiter, 1986, page 187.

In literature, LFSRs are described with different mathematical approaches. Common approaches use linear algebra, polynomial algebra and the theory of finite fields [10, pp. 186ff.][11, pp. 228ff.]. Less frequently, approaches with formal power series are chosen [6, pp. 53ff.][12, pp. 201ff.]. In this paper, the first three approaches are investigated further.

The sequence generated by an LFSR can be described as this relation:

$$s_{n+k} = a_{k-1}s_{n+k-1} + a_{k-2}s_{n+k-2} + \ldots + a_0 s_n + a \quad \text{for } n = 0, 1, \ldots$$

The variable s_{n+k} represents the value of the newly calculated bit after n clock signals, which changes state of the LFSR. The values $a_0, a_1, \ldots, a_{k-1}$ indicate the tapped memory cells. The constant a will be explained later. If a memory cell a_i is tapped, $a_i = 1$, if not $a_i = 0$. For an LFSR, a, a_i and s_i are elements of the finite field $\mathbb{F}_2$. This means they are elements of the Galois field of order $p = 2$, where p is a prime number. This corresponds to residue field $\mathbb{Z}/2\mathbb{Z}$. [13, p. 48] Initially, the LFSR of length k is assigned the values $s_0, \ldots, s_{k-1} \in \mathbb{F}_2$. The generated sequence $s_0, s_1, \ldots, s_{k-1}, s_k, \ldots, s_{k+n}$ is thus determined by the initial values. [10, pp. 186-187]

Figure 4 shows an LFSR of length $k = 5$ with the initial state: $s_0 = 1, s_1 = 1, s_2 = 0, s_3 = 1$ and $s_4 = 1$. The memory cells 0 and 1 are tapped, resulting in $a_0 = 1$ and $a_1 = 1$ with the remaining $a_i = 0$. After a clock signal, the binary addition $a_0 s_0 + a_1 s_1 = 1 \cdot 1 + 1 \cdot 1 = 0$ generates the new bit $s_{k-1+1} = s_{4+1} = s_5$, which is inserted in the freed cell after the shift

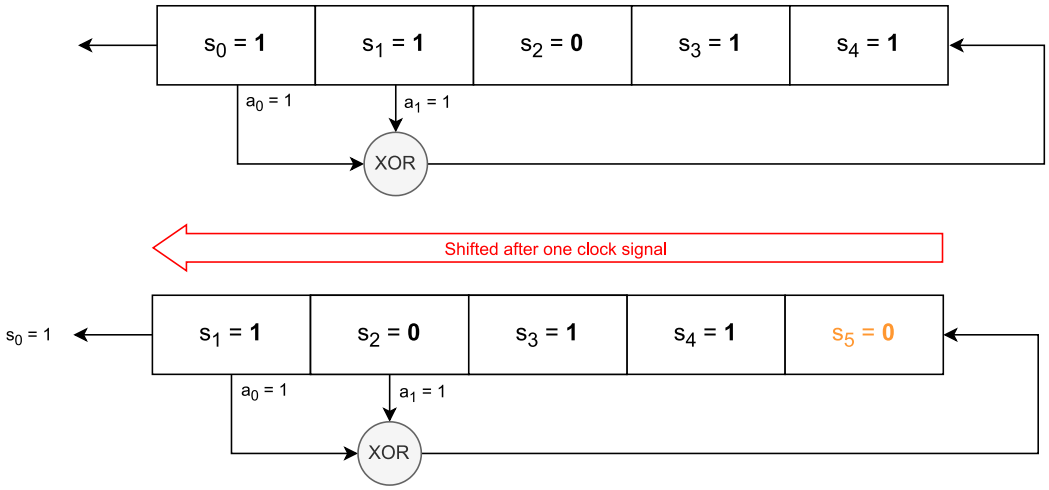


Figure 4: Workflow of an LFSR: After a clock signal, a new bit is calculated and the values of the memory cells are shifted to the left.

The sequence of shifted-out bits generated by an LFSR of length k is called a *k-th-order linear recurring sequence*. A distinction is made between *homogenous* k-th-order linear recurring sequences when $a = 0$ and *inhomogeneous* k-th-order linear recurring sequence when $a \neq 0$. The last case corresponds to the addition of a constant value in $\mathbb{F}_2$ to the equation above. [10, p. 186] Since LFSRs are usually constructed to produce homogenous k-th-order linear recurring sequences, the case $a \neq 0$ is not considered further here.

As the name suggests, it is characteristic of k-th-order linear recurring sequences that they will repeat. A sequence is said to be *periodic* if it will repeat after $r \in \mathbb{N}$ generated bits, such that $s_{n+r} = s_n$ for all $n \in \mathbb{N}_0$, where r denotes the period of the sequence. This periodic behavior can possibly occur after $n_0 \in \mathbb{N}_0$ non-periodic bits, such that: $s_{n+r} = s_n$ for all $n \geq n_0$. In this case the sequence is referred to as *ultimately periodic* and n_0 as *preperiod*. Any k-th-order linear recurring sequence with elements of $\mathbb{F}_2$ generated by an LFSR of length k is ultimately periodic. The period r takes at

most the value $r \leq 2^k - 1$. This can be explained since a register of length k can only hold $2^k - 1$ non-zero states. To avoid the trivial zero period, an LFSR of length k must not be initialized with the zero sequence $s_0 = s_1 = \ldots = s_{k-1} = 0$. [10, p. 189]

Figure 5 shows how the LFSR of length $k = 5$ from the previous example generates a sequence of period $r = 3$. The register returns to the initial state after 3 clock signals. The preperiod is $n_0 = 0$.

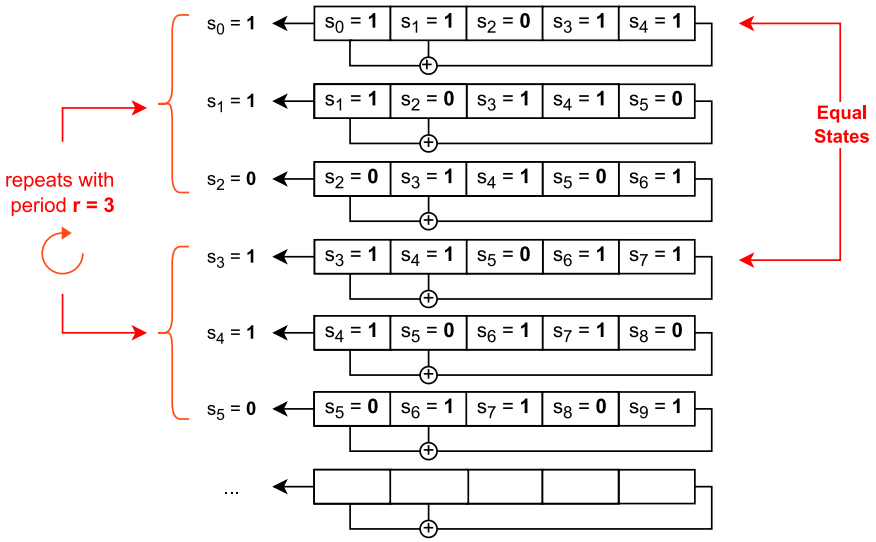


Figure 5: Periodic behavior of an LFSR. Once the state of an LFSR occurs a second time, the generated sequence will repeat.

Depending on the chosen values for $s_0, \ldots, s_{k-1} \in \mathbb{F}_2$ a different sequence with a possibly different period can be obtained. This can be illustrated by the following example: For each period, a state diagram is plotted. State i in the diagram corresponds to an inner state of the LFSR in decimal notation. This state is formed by the contents of the memory cells 0 to $k - 1$ at a point in time. A state transition is created by applying the feedback function R. This representation is adapted from Nigel P. Smart in [11, pp. 230]. Figure 6 shows all 32 states of the LFSR of length $k = 5$ from the previous two examples.

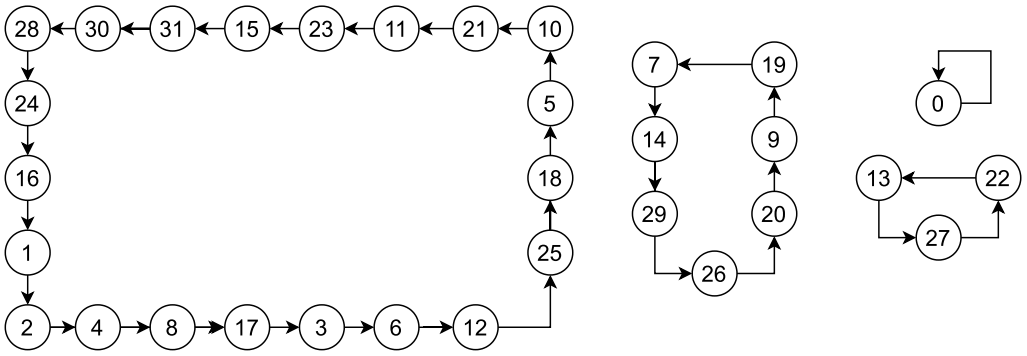


Figure 6: Representation of different periods of an LFSR as state machine diagrams. Based on [11, pp. 230-232]

3.2 m-sequences with Primitive Polynomials

Consequently, the ultimate goal is to generate a pseudorandom number with the largest possible period. One option of trial and error is to select other tapped memory cells and check if the period has improved. Additionally, the choice of initialization values is limited, since different values for the same register may produce sequences with different periods. Such an effort is unfavorable for a symmetric encryption system. The most favorable situation would be to always generate a sequence with the largest possible period for an LFSR, regardless of the initialization vector chosen.

To achieve this situation, a second way must be presented to describe homogeneous k-th-order linear recurring sequences: An LFSR of length k can be represented as the following regular $k \times k$ matrix A over $\mathbb{F}_2$: the matrix elements e in the diagonal below the main diagonal are assigned 1: $e_{2,1}, e_{3,2}, \ldots, e_{k,k-1} = 1$. The values of the tapped memory cells a_i as defined above, are written in the k-th column in ascending order by index. [10, p. 191] For consistency, the notation of the matrix was adopted from Lidl and Niederreiter. Nigel P. Smart transposes the matrix [11, p. 218]. As a result, the matrix $A \in \mathbb{F}_2^{k \times k}$ has this form:

$$\begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & a_0 \\ 1 & 0 & 0 & \ldots & 0 & a_1 \\ 0 & 1 & 0 & \ldots & 0 & a_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & a_{k-1} \end{pmatrix} \quad \text{if } k = 1, A \text{ is the } 1 \times 1 \text{ matrix } (a_0)$$

The crucial point is the examination of the characteristic polynomial $f(x)$ of the matrix:

$$f(x) = det(xI - A) = x^k - a_{k-1}x^{k-1} - a_{k-2}x^{k-2} - \ldots - a_0 \in \mathbb{F}_2[x]$$

In the case of an LFSR, this polynomial is an element of the polynomial ring $\mathbb{F}_2[x]$ over the finite field $\mathbb{F}_2$. Therefore, the coefficients of the polynomial – meaning the tapped memory cells – are elements of the finite field $\mathbb{F}_2$ [13, pp. 18-19]. Therefore, the signs of the coefficients before the indeterminant x can be reversed:

$$f(x) = x^k + a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \ldots + a_0 \in \mathbb{F}_2[x]$$

Only with the use of this characteristic polynomial it is possible to construct an LFSR which generates so-called *maximal period sequences* with elements of $\mathbb{F}_2$. This refers to a sequence with the largest possible period and without preperiod, generated by an LFSR for any initial values except 0. These sequences are also denoted as *m-sequences*. Such sequences are generated by an LFSR of length k if the characteristic polynomial of the matrix A is a so-called *primitive polynomial* in $\mathbb{F}_2[x]$. The period for all initial values except the zero initialization is $2^k - 1$. [10, p. 201]

In the literature it is difficult to find a coherent and uniform explanation to primitive polynomials. For this paper the approach of Lidl and Niederreiter via finite fields was chosen.

Defining a primitive polynomial starts with the polynomial f in $\mathbb{F}_2[x]$ of degree $m \geq 1$. For an element α in $\mathbb{F}_{2^m}$ it must hold that $f(\alpha) = 0$ is satisfied, whereby α is called the *root* of f. The finite field $\mathbb{F}_{2^m}$ consists of the elements of $\mathbb{F}_2[x]$ modulus the polynomial f, resulting in $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/f(x)$. [11, p. 11] At the same time, f must be the so-called *minimal polynomial* of α: Besides the property

$f(\alpha) = 0$, it must not be possible to factor it into further polynomials in $\mathbb{F}_2[x]$ of degree greater 0. This property is comparable to that of a prime number. Moreover, if there is a polynomial g in $\mathbb{F}_2[x]$ for which also holds $g(\alpha) = 0$, then g must divide the minimal polynomial f. The crucial characteristic of a minimal polynomial f is that it is the polynomial of minimal degree with $f(\alpha) = 0$, where the leading coefficient must be 1. For this polynomial f to be finally a primitive polynomial, it must hold that α generates the multiplicative group $\mathbb{F}_{2^m}^{\times}$. This group is equal to $\mathbb{F}_{2^m}$ without the 0. Thus $\langle\alpha\rangle = \mathbb{F}_{2^m}^{\times}$ must hold. [13, pp. 23, 31, 89]

Just as it is difficult to test whether a number is prime, it is complex to find out whether a polynomial is primitive. [4, p. 431] Therefore, in the following example, we will only show that a given primitive polynomial satisfies the properties above:

Given is the primitive polynomial $f(x) = x^4 + x^1 + 1$ in $\mathbb{F}_2[x]$ of degree $m = 4$. For $\alpha \in \mathbb{F}_{2^4}$ holds $f(\alpha) = 0$. This is how the finite field $\mathbb{F}_{2^4} = \mathbb{F}_2[x]/(x^4 + x^1 + 1)$ is composed:

0	1						
α	$\alpha+1$						
α^2	α^2+1	$\alpha^2+\alpha$	$\alpha^2+\alpha+1$				
α^3	α^3+1	$\alpha^3+\alpha$	$\alpha^3+\alpha+1$	$\alpha^3+\alpha^2$	$\alpha^3+\alpha^2+1$	$\alpha^3+\alpha^2+\alpha$	$\alpha^3+\alpha^2+\alpha+1$

It can be assumed that the polynomial f is a minimal polynomial. Now it will be shown that $\langle\alpha\rangle = \mathbb{F}_{2^m}^{\times}$, meaning that α generates the multiplicative group $\mathbb{F}_{2^m}^{\times}$.

$$
\begin{aligned}
\alpha^1 &= \alpha \\
\alpha^2 &= \alpha^2 \\
\alpha^3 &= \alpha^3 \\
\alpha^4 &= \alpha + 1 \\
\alpha^5 &= \alpha(\alpha^4) = \alpha(\alpha+1) \\
&= \alpha^2 + \alpha \\
\alpha^6 &= \alpha^2\alpha^4 = \alpha^2(\alpha+1) \\
&= \alpha^3 + \alpha^2 \\
\alpha^7 &= \alpha^3\alpha^4 = \alpha^3(\alpha+1) = \alpha^4\alpha^3 \\
&= \alpha^3 + \alpha + 1 \\
\alpha^8 &= \alpha^4\alpha^4 = (\alpha+1)(\alpha+1) \\
&= \alpha^2 + 2\alpha + 1 \\
&= \alpha^2 + 1 \\
\alpha^9 &= \alpha^5\alpha^4 = (\alpha^2+\alpha)(\alpha+1) \\
&= \alpha^3 + \alpha^2 + \alpha^2 + \alpha \\
&= \alpha^3 + \alpha \\
\alpha^{10} &= \alpha^5\alpha^5 = (\alpha^2+\alpha)(\alpha^2+\alpha) \\
&= \alpha^4 + 2\alpha + \alpha^2 \\
&= \alpha^2 + \alpha + 1
\end{aligned}
\qquad
\begin{aligned}
\alpha^{11} &= \alpha^5\alpha^6 = (\alpha^2+\alpha)(\alpha^2+\alpha) \\
&= \alpha^5 + \alpha^4 + \alpha^4 + \alpha^3 \\
&= \alpha^3 + \alpha^2 + \alpha \\
\alpha^{12} &= \alpha^6\alpha^6 = (\alpha^3+\alpha^2)(\alpha^3+\alpha^2) \\
&= \alpha^6 + 2\alpha^5 + \alpha^4 \\
&= \alpha^3 + \alpha^2 + \alpha + 1 \\
\alpha^{13} &= \alpha^6\alpha^7 = (\alpha^3+\alpha^2)(\alpha^3+\alpha+1) \\
&= \alpha^6 + \alpha^4 + \alpha^3 + \alpha^5 + \alpha^3 + \alpha^2 \\
&= \alpha^3 + \alpha^2 + \alpha + 1 + \alpha^2 + \alpha + \alpha^2 \\
&= \alpha^3 + \alpha^2 + 1 \\
\alpha^{14} &= \alpha^7\alpha^7 = (\alpha^3+\alpha+1)(\alpha^3+\alpha+1) \\
&= \alpha^6 + \alpha^2 + 1 + 2\alpha^3\alpha + 2\alpha^3 + 2\alpha \\
&= \alpha^3 + \alpha^2 + \alpha^2 + 1 \\
&= \alpha^3 + 1 \\
\alpha^{15} &= \alpha^8\alpha^7 = (\alpha^2+1)(\alpha^3+\alpha+1) \\
&= \alpha^5 + \alpha^3 + \alpha^2 + \alpha^3 + \alpha + 1 \\
&= \alpha^2 + \alpha + \alpha^2 + \alpha + 1 \\
&= 1
\end{aligned}
$$

Consequently, $f(x) = x^4 + x^1 + 1$ satisfies the properties of a primitive polynomial. The corresponding LFSR shown in figure 7 can be obtained with the help of the definition of the characteristic polynomial introduced previously: $f(x) = x^k + a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \ldots + a_0 \in \mathbb{F}_2[x]$

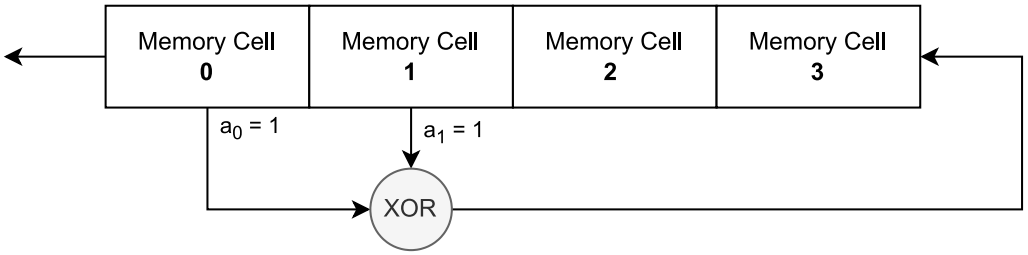


Figure 7: Structure of the LFSR which corresponds to the primitive polynomial $f(x) = x^4 + x^1 + 1$, based on [4, p. 430]

The corresponding state machine diagram for this LFSR of length $k = 4$ is shown in figure 8. As can be seen, apart from the trivial zero period, only one period with maximum length $2^k - 1 = 2^4 - 1 = 15$ exists. Thus, this LFSR, of which the characteristic polynomial is primitive, produces an m-sequence.

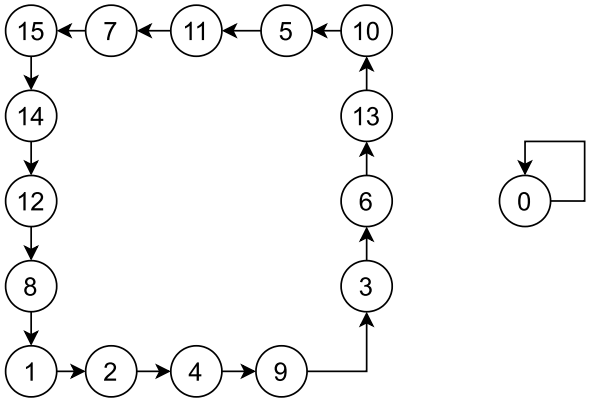


Figure 8: State machine diagram of the largest possible period of an LFSR whose characteristic polynomial is primitive. Based on [11, pp. 230-232]

Proof that such a primitive polynomial indeed generates an m-sequence is difficult to find in the literature. In many cases there are references to other authors or textbooks, without going into more detail or naming them. See, for example [11, p. 229]. I wanted to have found a coherent explanation in the literature to the question: *"Why do primitive polynomials in* $\mathbb{F}_2[x]$ *of degree* k *generate sequences with maximum period* $2^k - 1$*?"* For this paper, the example shown above should be sufficient.

In addition to having the longest possible period, m-sequences satisfy Solomon Golomb's randomness criteria mentioned initially [14, p. 2847]. Thus, m-sequences are statistically secure pseudorandom numbers. However, the fact that a statistically secure pseudorandom number, such as the m-sequence, does not necessarily equal a cryptographically secure number will be discussed in the next chapter.

4 Security of Stream Ciphers based on LFSRs

To evaluate the usability of encryption methods in a real-world environment, multiple factors need to be analyzed, like the ease of implementation, performance and security. Based on the discussed technical realization, a pseudo-random bit stream can be generated. In computer hardware stream ciphers are efficiently implemented with shift registers [9]. The next bits of the keystream can be calculated in advance to further improve their processing speed [15, p. 3]. Once data is received, it is XOR-combined with the keystream to en- or decrypt the data. By applying the primitive polynomial to an LFSR, their output always has the largest possible period indifferent to the initial values of the memory cells. Because the keystream is periodic and therefore infinite, messages with unknown length like real-time audio and video data are generally encrypted this way [11, p. 181]. Especially since there is no error propagation. If one bit of the encrypted message differs from its correct value, the following bits are not defective and can still be deciphered properly [11, p. 181]. These reasons established their wide usage in cryptographic contexts [16, p. 97]. However, stream ciphers also need to be considered cryptographically secure in order to be useful as an encryption tool. The next chapters demonstrate different attacks on stream ciphers and introduce improvements to their security aspects.

4.1 Known Plaintext Attack

In cryptanalysis, attacks can be categorized based on the data available to the adversary. Besides the ciphertext only attacks and chosen plaintext attacks, there is also the group of *known plaintext attacks*. Here, the plaintext of an encrypted sequence and its location in the message are laid open. [9, p. 2-3] Because of their linear nature, LFSR-based stream ciphers are prone to these known plaintext attacks [11, p. 233]: Given the adversary has a segment s of the encrypted message and the corresponding plaintext p, the used keystream k can be reproduced by calculating $s_i \oplus p_i$. This is possible due to the mathematical properties of XOR like $b \oplus b = 0$ and $0 \oplus b = b$. The method can be especially abused for metadata like header fields since their structure and content are mostly known [17, p. 359]. Figure 9 illustrates the principle behind a basic known plaintext attack.

$$
\begin{array}{lllll}
\textit{Given:} & s & = (s_0, s_1, ..., s_n) \in \mathbb{F}_2 & := \text{encrypted message} \\
& p & = (p_0, p_1, ..., p_n) \in \mathbb{F}_2 & := \text{plaintext} \\
& k & = (k_0, k_1, ..., k_n) \in \mathbb{F}_2 & := \text{keystream} \\
& f(k_i, p_i) & = k_i \oplus p_i = s_i & := \text{encryption function}
\end{array}
$$

$$
\textit{Attack:} \qquad s_i \oplus p_i = (k_i \oplus p_i) \oplus p_i = k_i \oplus 0 = k_i
$$

Figure 9: Basic known plaintext attack on a stream cipher

If the period of a keystream is shorter than the gained segment of its plaintext, the rest of the message can be decrypted [12, p. 9]. Therefore, a large period is necessary to diminish this threat [9, p. 83]. Even if it is not possible for the adversary to recreate the complete keystream period, the original data p can be replaced by malicious content p' of the same length. This is also known as a bit-flipping

attack [18, p. 6]. To visualize such an attack, it is assumed that the position of the plaintext '10.000€' and its corresponding encrypted message s is known. The first digit is now replaced by a '9' in Figure 10. This is a clear security concern.

Sender (sends s):
$p = 00000001_{(2)} = 1_{(10)}$
$k = 10111110_{(2)}$
$s = k \oplus p = 10111110 \oplus 00000001 = 10111111$

Adversary (receives s, knows p and sends s'):
$p' = 00001001_{(2)} = 9_{(10)}$
$s \oplus p \oplus p' = 10111111 \oplus 000000001 \oplus 00001001$
$= 10111110 \oplus 00001001 = k \oplus p'$
$= 10110111 = s'$

Receiver (receives s'):
$k \oplus s' = 10111110 \oplus 10110111 = 00001001 = 9_{(10)}$

Figure 10: Replacing original data with modified text in a known plaintext attack

4.2 Linear Complexity and the Berlekamp-Massey Algorithm

Besides the period of a sequence, its *linear complexity* is also used as an indicator for the cryptographic usefulness of the keystream.

Definition: The linear complexity $L(s)$ of a finite binary sequence s is equal to the length and therefore degree of the shortest LFSR to generate s [11, p. 233]. L follows the properties [19, pp. 20-21]:

- s is the zero sequence with $(0, 0, ..., 0)$ $\Leftrightarrow$ $L(s) = 0$
- s has length n with format $(0, 0, ..., 1)$ $\Leftrightarrow$ $L(s) = n$
- s cannot be generated by an LFSR $\Rightarrow$ $L(s) = \infty$
- s is periodic with period r $\Rightarrow$ $L(s) \leq r$
- s is the one-periodic sequence of a primitive feedback polynomial with degree n $\Rightarrow$ $L(s) = n$

The *Berlekamp-Massey algorithm* presented in the paper 'Shift-register synthesis and BCH decoding' can be used to calculate the linear complexity of a sequence and its corresponding shortest LFSR. Given a primitive polynomial has degree n and its generated period has linear complexity L. If an adversary gains a keystream segment k with length $l \geq 2L$, the primitive polynomial can be determined [20, pp. 124-125]. With this information, the used LFSR can be recreated and the periodic keystream can be acquired [11, p. 232]. Hence, the linear complexity is directly connected to the required bits of the keystream to crack a stream cipher.

After obtaining a finite keystream snippet k by for example exploiting a known plaintext attack, it can be used as the input for the Berlekamp-Massey algorithm. The algorithm has an efficient linear run time of $O(n)$ for a sequence with length n. Its structure is displayed in Figure 11.

k $= (k_0, k_1, ..., k_n) \in \mathbb{F}_2 :=$ keystream sequence of the LFSR and input for the algorithm
n $:=$ length of the input sequence
i $:=$ current index of the input sequence
i' $:=$ previous index since the last increment of the linear complexity
$C(x)$ $= 1 + c_1x^1 + c_2x^2 + ... + c_ix^i$ (mod 2)
$:=$ feedback connection polynomial of the minimal LFSR generating k
c_i $:=$ a cell of the shift register. If tapped: $c_i = 1$ else $c_i = 0$
$B(x)$ $:=$ previous connection polynomial since the last increment of the linear complexity
L $:=$ linear complexity of the minimal LFSR
d $:=$ discrepancy between the input and the output generated by $C(x)$

Berlekamp-Massey(k) :

$n = |k|$
$C(x) = B(x) = 1$
$L = i = 0$
$i' = -1$
while $i < n$:
    $d = k_i \oplus c_1k_{i-1} \oplus c_2k_{i-2} \oplus ... \oplus c_Lk_{i-L}$
    if $d == 1$:
        $C_{tmp}(x) = C(x)$
        $C(x) = C(x) + (B(x) * x^{i-i'})$
        if $L <= \frac{i}{2}$:
            $L = i + 1 - L$
            $i' = i$
            $B(x) = C_{tmp}(x)$
    $i = i + 1$
$return(L, C(x))$

Figure 11: Explanation and structure of the Berlekamp-Massey algorithm [20]

As a demonstration, the LFSR in Figure 8 with a characteristic polynomial $G(x) = 1 + x + x^4$ is uniquely determined by inputting its output bit sequence into the Berlekamp-Massey algorithm. A characteristic polynomial $G(x)$ and connection polynomial $C(x)$ both represent the structure of an LFSR. However, they differ in their written mathematical form. Since the algorithm produces $C(x)$, the formula $G(x) = x^L * C(\frac{1}{x})$ is used to convert the polynomials. The expected output for the demonstration can be calculated by reversing the above equation, resulting in $C(x) = 1 + x^3 + x^4$. As input, the six-bit deciphered keystream segment $k = 110001$ is inserted which corresponds to the initial state $IV_{12} = 12_{(10)} = 1100$ of the register cells. The state and applied actions of each loop iteration over the input sequence are presented in Figure 12.

Current values of attributes		Resulting attribute assignments	
Input: $k = 110001$	$n = 6$		
$i = 0$ $i' = -1$	$L = 0$	d	$\leftarrow 1$
$C(x) = 1$	$B(x) = 1$	$C(x)$	$\leftarrow 1 + 1 * x^1 = 1 + x$
		L	$\leftarrow 1$ $i' \leftarrow 0$ $B(x) \leftarrow 1$
$i = 1$ $i' = 0$	$L = 1$	d	$\leftarrow 0 \oplus 1 \odot 1 = 0$
$C(x) = 1 + x$	$B(x) = 1$		
$i = 2$ $i' = 0$	$L = 1$	d	$\leftarrow 0 \oplus 1 \odot 1 \oplus 1 \odot 0 = 1$
$C(x) = 1 + x$	$B(x) = 1$	$C(x)$	$\leftarrow 1 + 1 * x^1 + (1 * x^{2-0}) = 1 + x + x^2$
		L	$\leftarrow 2$ $i' \leftarrow 2$ $B(x) \leftarrow 1 + x$
$i = 3$ $i' = 2$	$L = 2$	d	$\leftarrow 0 \oplus 1 \odot 0 \oplus 1 \odot 1 \oplus 1 \odot 0 = 1$
$C(x) = 1 + x + x^2$	$B(x) = 1 + x$	$C(x)$	$\leftarrow 1 + x + x^2 + ((1 + x) * x^{3-2})$
			$\leftarrow 1 + 2x + 2x^2 \pmod 2 = 1$
$i = 4$ $i' = 2$	$L = 2$	d	$\leftarrow 0$
$C(x) = 1$	$B(x) = 1 + x$		
$i = 5$ $i' = 2$	$L = 2$	d	$\leftarrow 1$
$C(x) = 1$	$B(x) = 1 + x$	C(x)	$\leftarrow 1 + ((1 + x) * x^{5-2}) = 1 + x^3 + x^4$
		L	$\leftarrow 5$ $i' \leftarrow 5$ $B(x) \leftarrow 1$
		Output: $C(x) = 1 + x^3 + x^4$	$L = 4$

Figure 12: Step-by-step execution of the Berlekamp-Massey algorithm

To validate the gained connection polynomial again, it is inserted into the equation $G(x) = x^L * C(\frac{1}{x}) = x^4 * (1 + \frac{1}{x^3} + \frac{1}{x^4}) = x^4 + x + 1$. The expected output was indeed computed correctly by the algorithm after six iterations. In the example, the adversary knew only six bits and recreated the LFSR successfully. In this case, at most $2 * L = 2 * 4 = 8$ bits would have been required for the algorithm to determine the connection polynomial. Alternatively, this can also be achieved with an equation as long as the length of the LFSR is known. To calculate which memory cells c_i are tapped, the following formula can be used [11, p. 232]:

$$\begin{pmatrix} k_{L-1} & k_{L-2} & ... & k_1 & k_0 \\ k_L & k_{L-1} & ... & k_2 & k_1 \\ ... & ... & ... & ... & ... \\ k_{2L-3} & k_{2L-4} & ... & k_{L-1} & k_{L-2} \\ k_{2L-2} & k_{2L-3} & ... & k_L & k_{L-1} \end{pmatrix} * \begin{pmatrix} c_1 \\ c_2 \\ ... \\ c_{L-1} \\ c_L \end{pmatrix} = \begin{pmatrix} k_L \\ k_{L+1} \\ ... \\ k_{2L-2} \\ k_{2L-1} \end{pmatrix}$$

Since the degree of a primitive polynomial is equal to its linear complexity, even for an LFSR with a period of $2^{512} - 1$ bits already 1024 bits of the keystream are sufficient to crack the stream cipher. Thus, pure LFSRs are of no value as cryptographic tools due to their linear behavior [11, p. 231].

5 Increasing the Cryptographic Qualities of LFSRs

In an attempt to increase the security of LFSRs, several adjustments to their basic structure can be made [16, p. 97]. The so far discussed exploits mainly abuse their linear nature. Therefore, LFSRs can be combined with nonlinear transformations, for example, the multiplication of two bits by an AND function, to diminish their weak points. In this section, a few concepts based on this idea are analyzed.

5.1 Nonlinear Feedback Shift Registers (NLFSR)

Instead of combining the tapped cells linearly, NLFSRs utilize a *nonlinear connection polynomial*. This approach seems secure since there are 2^{2^n} possible Boolean functions for n bits. However, it is mathematically proven that every bit of a keystream with period r can be correctly determined after at most r bits. For example, the algorithm by Boyar and Krawczyk recursively computes the whole keystream after $n + m$ bits of plaintext, where n is the length of the NLFSR and m is the number of degrees of freedom of the feedback function. The required plaintext is generally much smaller than the period of the NLFSRs. This is an improvement in comparison to LFSRs, however still not secure enough to be considered a safe encryption method on their own [16, p. 97]. There are even further restrictions that keep reducing their cryptographic value. For example, to still guarantee the effective calculation of the nonlinear function, the amount of tapped cells is limited to a realistic sum. [21] Consequently, the concept of NLFSRs is not further discussed in this paper.

5.2 Combining Linear Feedback Shift Registers with an Output Generator

Another solution is to combine the n output bits of multiple LFSRs based on a nonlinear function $f : \mathbb{F}_2^n \to \mathbb{F}_2$. A famous representative of this group is the *Geffe generator* which was developed by P.R. Geffe in 1973. It involves two LFSRs, whose outputs a_i and b_i are chained together by the nonlinear function $f(a_i, b_i, c_i) = k_i = a_i + c_i a_i + c_i b_i$, where c_i is produced by a third LFSR. The idea behind a Geffe generator is the selection of the keystream bit k_i based on c_i. If $c_i = 0$ then a_i is returned, else b_i is chosen. [22] Thus, the combination component is implemented as a *multiplexer* whose index value is c_i [15, p. 19]. The three LFSRs are not limited to the same length. This structure can be seen in Figure 13.

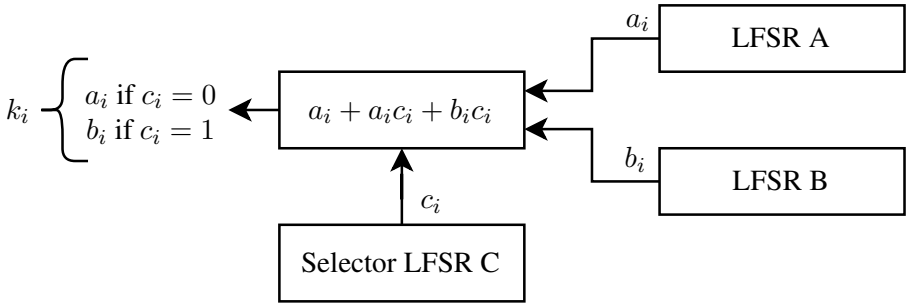


Figure 13: Graphic representation of a Geffe generator

This results in the following changes to the cryptographic metrics of its keystream k: [11, p. 234]

$$\begin{array}{lcl} \text{Linear Complexity } L_k & = & L_A + (L_A * L_C) + (L_B * L_C) \\ \text{Period } r_k & = & (2^{L_A} - 1) * (2^{L_B} - 1) * (2^{L_C} - 1) \end{array}$$

The increase in linear complexity and period improves the quality of the stream cipher. However, the Geffe generator has a statistical weakness that can be exploited. To prove this statement, its truth table is created in Figure 14.

a_i	b_i	c_i	k_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Figure 14: Truth table of a Geffe generator

It can be deducted, that an output bit k_i has the probability $p = \frac{3}{4}$ of being equal to the bit a_i of LFSR A. The same can be seen for LFSR B. Consequently, the Geffe generator can be easily broken with a *correlation attack* [16, p. 104].

A correlation attack is a brute-force plaintext attack in which the correlation between the keystream k and the generated sequences of one or multiple LFSRs is exploited. For the Geffe generator a random initial state IV_A for LFSR A is produced. The output a of the LFSR A corresponding to IV_A is then compared to k. If the guessed state is correct, they should be equal in approximately $\frac{3}{4}$ths of their bits. If so, the next step is to calculate the initial state for LFSR B and afterwards LFSR C, else a different initial state for LFSR A is chosen. The problem of estimating the correct initial states of all LFSRs at once is divided into determining only one at a time. This attack pattern is also called a divide-and-conquer attack [15, p. 17] and requires at most $(2^{L_A} + 2^{L_B} + 2^{L_C})$ times the 'total number of possible

connection polynomials' operations to assert the three initial states. Without this approach the first part of the equation is increased to $2^{L_A+L_B+L_C}$ attempts, due to the missing separated calculation of the initial state by the divide-and-conquer pattern. A correlation attack amplifies the probability of guessing the correct states for all LFSRs drastically. [11, p. 235]

To measure the proneness of a cipher to correlation attacks, a new term is introduced. A function is m^{th}*-order correlation-immune* when any subset of m input bits are uncorrelated to the output bit [23, p. 777]. Interestingly, a high linear complexity L of the combination function results in low correlation-immunity m [23, p. 779].

5.3 Extending Nonlinear Output Generators with Memory Cells

To counter the undesired effect between linear complexity and correlation-immunity, the combination function can be expanded with a memory component [15, p. 17]. The addition of memory turns the system into a nonlinear finite state machine [24, p. 209]. The combination function f is modified to not only produce the output bit k_i but also the next state of the machine σ_i.

The *summation combiner* uses this principle and allows for maximum linear complexity and correlation-immunity [25, p. 261]. The generated bits of two LFSRs are added together every tact and the resulting carry σ_i is saved in an one-bit memory, similar to an integer addition. This method proves to be highly nonlinear. The general structure is illustrated in Figure 15. [26, p. 70]

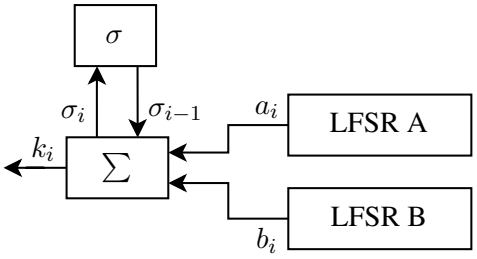


Figure 15: Basic overview of a summation combiner with two input LFSRs

Its combination functions are defined as:

$$\begin{aligned} f_k(a_i, b_i): \quad k_i &= a_i \oplus b_i \oplus \sigma_{i-1} && := \text{keystream bit} \\ f_\sigma(a_i, b_i): \quad \sigma_i &= a_i b_i \oplus a_i \sigma_{i-1} \oplus b_i \sigma_{i-1} && := \text{carry} \end{aligned}$$

The summation combiner accomplishes improvements in regards to the period r_k and linear complexity L_k of the keystream k. The period r_k is equal to $r_a * r_b$ for an integer addition of two sequences a and b with their corresponding periods r_a, r_b assuming $gcd(r_a, r_b) = 1$ [12, p. 220]. Also the gained linear complexity L_k is generally close to the period r_k with $L_k \leq r_a * r_b$ [12, p. 225]. In terms of the correlation between the state and its output, it still shows a slight bias which can be exploited by a correlation attack. This bias can be decreased by adding more LFSRs as input for the integer addition and increasing its memory size [26, pp. 81-82].

5.4 Clock-Controlled Linear Feedback Shift Registers

Until this point in the paper, all registers were shifted by one bit on every clock tick. A different idea for refining the security characteristics of LFSRs is changing their tick rate based on the output of another register. This introduces a nonlinear nature to the keystream. The *Stop-and-Go generator* for example has two LFSRs A and B, which output a_i and b_i respectively. The clock signal c_i is AND-combined with b_i before being connected to the LFSR A in order to shift the register only if $b_i = 1$. As long as $(b_i, b_{i+1}, ..., b_{i+n}) = (0, 0, ..., 0)$ the generator produces the unchanged bit a_i. [27, pp. 89-90] Since the Stop-and-Go generator suffers from a high correlation between b_i and a switch in the bits of the keystream, it is prone to correlation attacks [28, p. 156]. An improvement to this concept is the *shrinking generator*. Instead of outputting the same bit if $b_i = 0$, the LFSR A is shifted regardless however nothing is added to the keystream. This restricts the threat of correlation attacks [28, p. 159]. Even though the clock signal is not directly tampered with, the shrinking generator counts towards the group of the clock-controlled stream ciphers [15, p. 23]. In Figure 16 the general structures of both generators are shown.

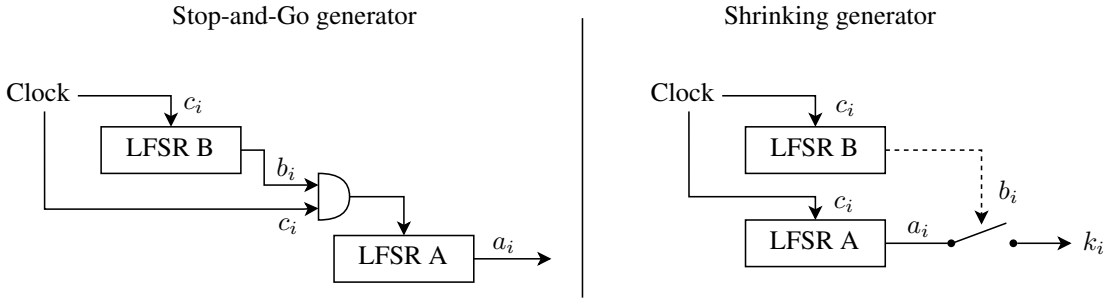


Figure 16: The Stop-and-Go [27, pp. 89-90] and shrinking generator [28, p. 159]

Let LFSR A have a maximal period of $r_A = 2^{L_A} - 1$ with linear complexity L_A and LFSR B has accordingly period $r_B = 2^{L_B} - 1$ with linear complexity L_B. On the condition that L_A and L_B are relative prime, meaning $gcd(L_A, L_B) = 1$, the period r_k of the output keystream k is equal to $r_k = 2^{L_B-1} * r_A = 2^{L_B-1} * (2^{L_A} - 1)$ [29, p. 25][28, p. 159]. Furthermore, the linear complexity of the output keystream L_k can be described as: $L_A * 2^{L_B-2} < L_k \leq L_A * 2^{L_A-1}$ [29, p. 25].

Because the shrinking generator only sends bits if $b_i = 1$, the time between the consecutive bits k_i and k_{i+1} can be measured. The keystream of LFSR B can then be immediately reconstructed. These *timing attacks* are a specific variant of so called *side-channel attacks*. To lower the possibility of such attacks, dummy operations can be implemented or the actual bit b_i can be hidden by caching the output stream. [28, pp. 163-164]

The metrics to measure the cryptographic applicability of stream ciphers such as their period, linear complexity and correlation-immunity only indicate required properties to guarantee minimal security not their actual real-world usability [29, p. 24]. Over time a lot of stream ciphers have been cracked. For them to be reliable even today, more adjustments need to be made.

6 Strongest Stream Ciphers - eSTREAM Contest

eSTREAM was a crypto-contest held from 2004 till 2008 and aimed at discovering especially well-designed stream ciphers. eSTREAM came about due to another contest, NESSIE, which had no winners in the ”Stream Ciphers” category because all of the ciphers had considerable vulnerabilities. Then, at the RSA Data Security Conference, it was argued whether stream ciphers can be considered obsolete if the Advanced Encryption Standard (AES) offers good solutions most of the time. However, there had been arguments that supported stream ciphers describing their advantages over block ciphers [30]:

- Exceptionally high throughput in software.
- Exceptionaly low resource consumption in hardware.

To prove this, the European Network of Excellence for Cryptology (ECRYPT) launched the eStream project. It had three phases and featured different stream cipher designs in two Profiles [30]:

Profile 1: Stream ciphers for software applications with high throughput.

Profile 1A: Ciphers that satisfy Profile 1 with an associated authentification method.

Profile 2: Stream ciphers for hardware applications with highly restricted resources (limited storage, gate count, or power consumption).

Profile 2A: Ciphers that satisfy Profile 2 with an associated authentification method.

The following table pictures the initial requirements for the candidates:

	key length (bits)	IV length (bits)	tag length (bits)
Profile 1	128	64 and 128	-
Profile 1A	128	64 and 128	32, 64, 96 or 128
Profile 2	80	32 and 64	-
Profile 2A	80	32 and 64	32 or 64

Software-optimized ciphers should significantly outperform the AES when used in a suitable stream cipher mode to justify stream cipher usage. It is important to point out that the contestants did not try to develop ciphers that were good at everything. The main focus was laid on raw encryption speed by large amounts of data after a single initialization. The security level was standard 128 bit, which matched contemporary applications very well. To evaluate all the contestants fairly, a special testing framework was introduced [31]. It consisted of a collection of shell scripts and C-code and tested three aspects of every submitted cipher:

1. *API compliance* checked if all the API requirements, like necessary interface, and key sizes are correct, if calls to the same function with the same parameters produce the same results or whether the encryption functions produce the same encrypted text from the same plaintext.
2. *Correctness* of the code was verified by generating and comparing test vectors.

3. *Performance* was tested through the following criteria:

    - *Encryption rate* for long streams.
    - *Packet encryption* tested at which packet length a stream cipher would take the lead from a block cipher.
    - *Agility* tested the time spent switching from one session to another when encrypting many streams in parallel on one processor.
    - *Key and Initial Vector (IV) setup efficiency*.

Hardware-optimized ciphers should be significantly smaller and faster than AES in a restricted environment in at least one significant aspect [32]. These ciphers would be useful in, for example, sensor networks that are especially constrained in the number of logic gates or power. For this reason, the security level was chosen to be 80 bits.
Like with any stream ciphers, an initialization vector (IV) and a secret key had to be used. For the 1A and 2A profiles, an authentification tag was also required.
There were several evaluation criteria for each phase:

- Security.
- Performance when compared to the AES in an appropriate mode.
- Performance, when compared to other submissions.
- Justification and supporting analysis simplicity and flexibility.
- Completeness and clarity of submission.

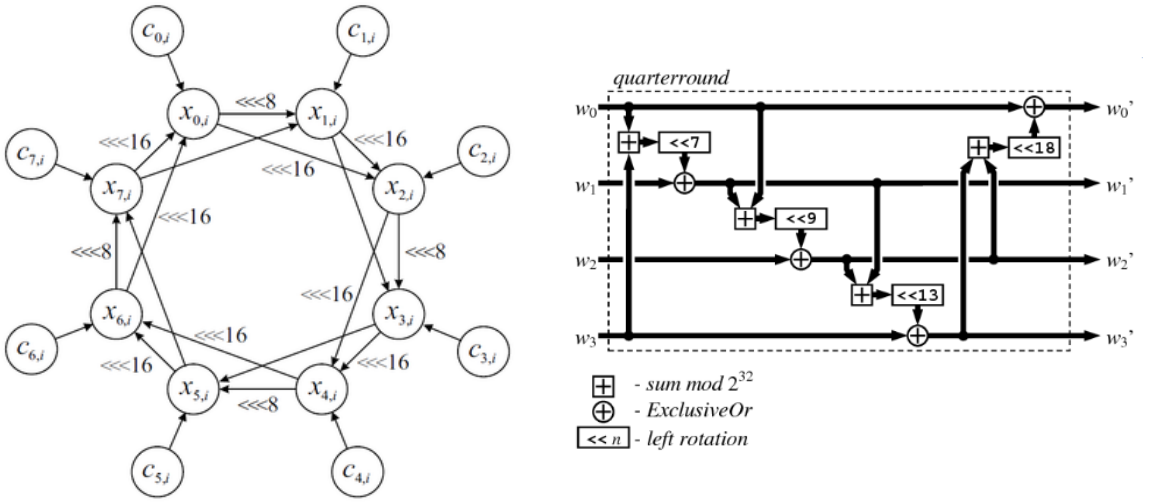
To go to the next phase a submission had to be notably better than AES.
eSTREAM lasted four years and in 2008 a voting at SASC 2008 took place, where the finalists of the eSTREAM were evaluated on a scale from -5 to +5 and then the average was calculated. It is important to point out that this evaluation was not held by eSTREAM and should only be used for illustration. The following table demonstrates the results of the voting:

Software optimized			Hardware optimized	
Rabbit	2.80		Trivium	4.35
Salsa20	2.80		Grain v1	3.50
SOSEMANUK	1.20		MICKEY v2	0.17
HC-128	0.60			

This table also shows that the opinions about the hardware-optimized ciphers were more unified than the ones about the software-optimized ones. The ciphers that made it into the eStream portfolio can be divided into several categories [33]:

ARX-based

ARX-based (Addition/Rotation/XOR) are ciphers that only use module addition, interword rotation, and XOR to achieve required security strength. These operations are fast and easy to implement in software as well as they are immune to timing attacks because they run in constant time [33]. Among the ciphers that made it into the eStream portfolio Rabbit and Salsa20 belong to ARX-based stream ciphers. Their internal structure can be seen in Figure 17.



(a) Internal structure of the Rabbit cipher [34]

(b) Internal structure of the Salsa20/r cipher [35]

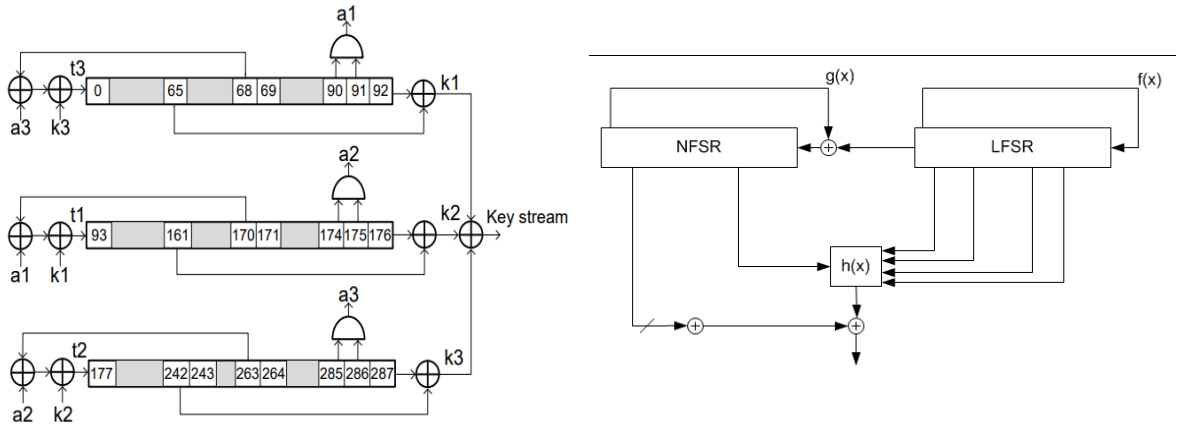
Figure 17: ARX-based eStream finalists

Rabbit is a software-efficient, synchronous stream cipher that uses a 128-bit key and a 64-bit IV. A set of eight state registers, each 32-bit long and eight 32-bit counters are used to provide encryption based on ARX operations. Testing during the eSTREAM process confirmed that the Rabbit cipher was one of the most efficient software-oriented stream ciphers submitted [36].

Salsa20/r is a software-oriented cipher that supports keys of 128 bits and 256 bits. During its operation, the key, a 64-bit nonce (unique message number), a 64-bit counter, and four 32-bit constants are mapped to the 512-bit initial state. After r iterations of the Salsa20/r round function, the updated state is used as a 512-bit keystream output. Due to its implementation Salsa20 resembles a block cipher [37]. Although many attacks on Rabbit and Salsa20 have been proposed, as of 2015, none of them have been more successful than brute force attacks.

NLFSR-based

NLFSR-based stream ciphers take NLFSR as the basic components. It uses both the nonlinear feedback and the nonlinear output to provide good sequence properties and security [33]. Among the eStream finalists, Trivium and Grain v1 belong to NLFSR-based ciphers. Their internal structure can be seen in Figure 18.



(a) Internal structure of the Trivium cipher [38]

(b) Internal structure of the Grain v1 cipher [39]

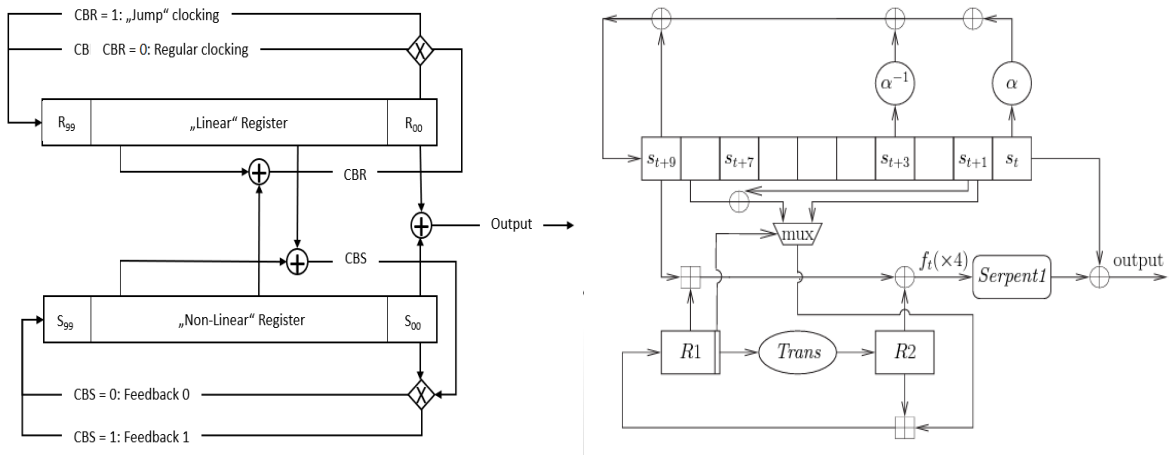
Figure 18: NLFSR-based eStream finalists

Trivium is a hardware-oriented stream cipher the main advantage of which is its simple design. It consists of three interconnected NLFSRs with feedback functions and linear or quadratic filter functions. It takes an 80-bit key and an 80-bit IV, and generates up to 264 keystream bits. It is important to note that Trivium was designed as an exercise, to explore how far a stream cipher can be simplified without sacrificing its security, speed or flexibility and still outperform AES [40].

Grain v1 combines the LFSR with the NLFSR, where LFSR provides good statistical characteristics and NLFSR adds nonlinear disturbance. A filter function is used to mix the state from both registers to further improve the security. It takes an 80-bit key and an 80-bit IV, using the LFSR and NLFSR each of 80-bit length, and a filter function of 5 variables and algebraic degree 3 [41]. A 2003 paper described a possible weakness in the cipher initialization [42]. As of October 2006, no key recovery attacks better than brute force attacks are known against Grain v1.

LFSR-based

LFSR-based stream ciphers can be either bit-oriented (driven by one or more bit-unit LFSRs for a large cycle and good statistical properties, and usually combined with means of a filter, combiner, or clock control to realize the nonlinearly scrambling [33]) or word-oriented (consists of the linear driver infinite field extension and a non-linear finite state machine (FSM), which can be regarded as a generalization of filter generators [33]). Their internal structure can be seen in Figure 19.



(a) Internal structure of the MICKEY v2 cipher [43] (b) Internal structure of the SOSEMANUK cipher [44]

Figure 19: LFSR-based eStream finalists

Mickey v2 is a bit-oriented LFSR-based cipher. It uses irregular timing of shift registers, as well as new methods that provide a sufficiently large period and pseudo-randomness of the key sequence and resistance to attacks [43]. As of 2013, a differential fault attack has been reported [45].

SOSEMANUK is a word-oriented LFSR-based cipher influenced by the stream cipher SNOW and the block cipher SERPENT. Like the SNOW cipher, the SOSEMANUK algorithm uses two basic concepts: an LFSR and an FSM. The data obtained using the LFSR is fed to the input of the FSM, where it is non-linearly transformed. The four output values of the state machine are then table-swapped and XORed with the appropriate shift register values. The key schedule for the cipher is compiled using the Serpent24 primitive [46]. Several attacks on SOSEMANUK have been published [47] [48]. However, none of the proposed attacks breaks the claimed 128-bit security of the cipher.

Random shuffled

Random shuffled stream ciphers achieve high efficiency of software implementation by employing randomly shuffled tables that shuffle a list to create random permutations, based on the RC4 cipher design. Since RC4 has got a number of applications in practice, it inspired subsequent designs of this type of structure, against which the state recovery attacks often require extremely high time complexity [33]. The internal structure of HC-128 can be seen in Figure 20.

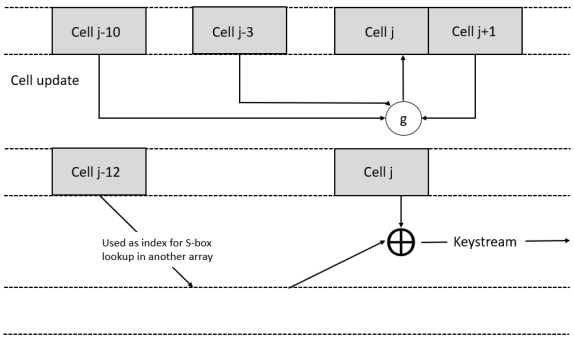


Figure 20: Internal structure of the HC-128 random shuffled cipher [49]

A representative of this category in the eStream portfolio is the *HC-128* cipher. Its internal state consists of two tables, each with 512 registers of 32 bits in length. At each step, one register of one of the tables is updated using a non-linear feedback function, while one 32-bit output is generated from the non-linear output filtering function. The cipher specifications state that 264 bits can be generated from each key/IV pair [50].

Despite the design's high profile, there have been no significant cryptanalytic advances against HC-128 since the publication of the eSTREAM portfolio.

7 Trivium

In this section, we would like to show the functionality of a stream cipher on the example of the Trivium. This cipher was chosen for two reasons:

1. Trivium has a very simple design, perfect for illustrative purposes.
2. Trivium scored very highly at the SASC 2008 evaluation.

7.1 Functionality

Trivium consists of the following phases [40]:

1. **Initialization**

   - IV: 80 bits.
   - Key: 80 bits.
   - Internal state: 288 bits ($s_1, \ldots, s_{288}$).
   - 4 * 288 = 1152 randomization cycles (after 1152 cycles a key stream is generated).
   - Key stream: N $\leq 2^{64}$ (z_0, ..., z_n), where N is the length of the plaintext.

   At first, the 80-bit key (K) and an 80-bit initialization vector (IV) are loaded into the 288-bit internal state, all the remaining bits are set to 0s, except the last three, which are set to 1s. Then the bits in the register are randomized and rotated over 4 full cycles without generating keystream bits to improve randomization as well as dependency between the internal state, key and the initialization vector . The pseudo-code below shows the process in more detail:

   $(s_1, s_2, \ldots, s_93) \leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0)$
   $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$
   $(s_{178}, s_{279}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$

   for i = 1 to 4 * 288 **do**
   $t_1 \leftarrow s_{66} \oplus s_{91} \odot s_{92} \oplus s_{93} \oplus s_{171}$
   $t_2 \leftarrow s_{162} \oplus s_{175} \odot s_{176} \oplus s_{177} \oplus s_{264}$
   $t_3 \leftarrow s_{243} \oplus s_{286} \odot s_{287} \oplus s_{288} \oplus s_{69}$
   $(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
   $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
   $(s_{178}, s_{279}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
   end for

   The initialization procedure ensures that every bit of the initial state depends on every bit of the key and every bit of the initialization vector. This effect is achieved already after 2 full cycles (2*288 cycle executions). 2 more cycles are needed to complicate the bit relationships. For example, the first 128 bytes of the keystream, obtained from the null key and the initialization vector, have approximately the same number of 1s and 0s evenly distributed. Even with the simplest and identical keys, the Trivium algorithm produces a sequence of numbers that is close to random.

2. **Stream generation**

   The keystream generation consists of an iterative process that extracts the values of 15 specific state bits and uses them both to update 3 bits of the state and to compute 1 bit of keystream z_i. The state bits are then rotated and the the process repeats itself until the requested $N \leq 2^{64}$ bits of keystream have been generated. The following pseudo-code describes the process in more detail:

   for i = 1 **to** N **do**

   $t_1 \leftarrow s_{66} \oplus s_{93}$
   $t_2 \leftarrow s_{162} \oplus s_{177}$
   $t_3 \leftarrow s_{243} \oplus s_{288}$
   $z_i \leftarrow t_1 \oplus t_2 \oplus t_3$
   $t_1 \leftarrow t_1 \oplus s_{91} \odot s_{92} \oplus s_{171}$
   $t_2 \leftarrow t_2 \oplus s_{175} \odot s_{176} \oplus s_{264}$
   $t_3 \leftarrow t_3 \oplus s_{286} \odot s_{287} \oplus s_{69}$
   $(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
   $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
   $(s_{178}, s_{279}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$

   end for

To better understand the algorithm, consider this example:

1. We initialize a plaintext. For simplicity, it we will take an 'a'.
   plainText = 0110 0001

2. We initialize a random initialization vector (IV) and a random key (K), 80 bits each.
   K = 0010 0101 1001 0101 1010 1011 0000 0011 0000 1101 0011
   1100 0010 0100 0001 0001 0111 0110 1110 1010

   IV = 0111 0000 1100 0001 0101 0111 1100 0110 1101 0111 1110
   1000 1011 0111 0001 0000 1110 1110 0000 0111

3. For this example we initialize three shift registers, 288 bits in total. Where *regA* is 93 bits long, *regB* - 84 bits, and *regC* is 111 bits long. The registers have to be loaded with the *IV* and the *K*. The first 80 bits of *regA* are loaded with the *K*, and the rest are 0.

   regA = 0010 0101 1001 0101 1010 1011 0000 0011 0000 1101 0011
   1100 0010 0100 0001 0001 0111 **0**110 1110 1**0**10 0**000** 0000
   00**00** **0**

   The first 80 bits of *regB* are loaded with *IV*, the rest is 0.
   regB = 0111 0000 1100 0001 0101 0111 1100 0110 1101 0111 1110
   1000 1011 0111 0001 0000 1**1**10 **1**110 0000 0111 0**000**

   The last three bits of *regC* are 1, the rest are 0.

   regC = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
   1000 0000 0000 0000 0000 0**0**00 0000 0000 0000 0000 00**0**0
   0000 0000 0000 0000 0000 **111**

4. As it is shown in the pseudo-code above, the bits are randomized with the help of XOR and AND operations, applied to the state register bits at the positions:

   $regA_{66}, regA_{69}, regA_{91}, regA_{92}, regA_{93}$,
   $regB_{69}, regB_{78}, regB_{82}, regB_{83}, regB_{84}$,
   $regC_{66}, regC_{87}, regC_{109}, regC_{110}, regC_{111}$.

   This is repeated for four cycles.
   Operations with these chosen bits guarantee the longest period before bits start to repeat.
   After the randomization we get the following:

   ```
   regA = 1101 0111 1001 1001 0110 0111 0001 0010 1010 1000 0101
          1000 1010 1011 1111 1100 0110 1011 1001 0110 1100 1110
          0100 1
   regB = 1101 1101 1101 0101 0111 0101 0011 1110 0001 0101 0011
          1000 0011 0001 0111 0111 0011 0011 1001 1110 1100
   regC = 1111 0011 0000 0111 1001 0110 0101 0001 0100 0010 1011
          1100 0000 0000 0011 1110 0110 0001 1010 1000 1101 0111
          0011 1010 1001 1000 1000 01
   ```

5. Next step is the key generation. The procedure is similar to the Key and IV initialization, but now a key stream, the same length as the *plainText*, is generated from the output variables:
   $z_i \leftarrow t_1 \oplus t_2 \oplus t_3$
   Where
   $t_1 = regA_{66} \oplus regA_{93}$
   $t_2 = regB_{69} \oplus regB_{84}$
   $t_3 = regC_{66} \oplus regC111$

6. t_3, t_1, t_2 are then added in the beginning of the *regA*, *regB*, *regC* respectively, shifting this way bites. After eight iterations we have our *keyStream*:
   `keyStream = z = 0101 0111 = 'W'`

7. Finally an encrypted text is generated by XORing corresponding bits from the key stream and from the plainText.
   `encryptedText = 0101 0111 ⊕ 0110 0001 = 0011 0110 = 6`

8. By XORing the encrypted text with a keyStream, we can get our plain text back.
   `plainText = 0011 0110 ⊕ 0101 0111 = 0110 0001 = 'a'`

7.2 Security

According to the authors of the cipher [40], it is harder to find *correlations* between the internal state bits and the keystream in Trivium because as opposed to LFSR based ciphers, Trivium's state register is updated in a non-linear way, so it is harder for an attacker to recover the state.

The non-linear nature of the cipher makes it nearly immune to the attacks that rely on determining its *period* as well as *algebraic attacks*. *Resynchronization attacks* are avoided by cycling the state four times (step 4 in the example above) before producing any output. The authors name *Guess and Determine attacks* as the ones of the most concern. In 2013 another team of researchers has shown that Trivium can be broken by a *cube attack*, if the number of initialization rounds is reduced to 799 [51].

Only in 2020 [38] a not reduced version of Trivium was broken with the help of *Experimental Attacks and Differential fault analysis (DFA)*. The idea is to inject faults into the Trivium clock cycle and get the faulty outputs. Then with the help of DFA the internal state is recovered through a comparison between correct and faulty outputs, after that, a specially designed Trivium version goes back and gets the initial key and the internal state. According to the authors, 100% of keys have been recovered using this method.

8 Conclusion

In this paper, we have shown the principles, functionality, and mathematical basis of stream ciphers as well as evaluated their security. It is easy to notice that stream ciphers are a huge family united by the idea of streams of random numbers. And continuing the thought of John von Neumann, "...no such thing as a random number - there are only methods to produce random numbers." [2, p. 36] we see that the variety of stream cipher designs is built on a pseudo-random flow of digits, which is just good enough to make things difficult for an attacker. Although all stream ciphers have an LFSR in them, it would be very insecure to use only an LFSR to create a random digit stream. LFSRs are not cryptographically secure because they are easy to reverse engineer. That is why stream ciphers usually have a non-linear part to improve security. As a result of our research, the following advantages of stream ciphers over block ciphers can be defined:

1. Due to bit-by-bit operations, stream ciphers are more efficient and faster than block ciphers.
2. They are better suited for environments with limited resources.
3. If there's an error in one symbol, it'll be less likely to affect the next.
4. Easy to implement in hardware, using Flip-Flops and gates.

The main disadvantages of stream ciphers, however, are:

1. Due to looping in LFSRs, stream ciphers are not very software efficient.
2. Stream ciphers are very vulnerable to bit-flipping and code reuse attacks because of their bit-by-bit approach.
3. Because stream ciphers work not with blocks, but with bits, an adversary can change a strategically chosen bit in the encrypted text to change the decrypted one.

Stream ciphers are usually used where the amount of data is unknown or may be continuous, for example in wireless networking, or in the military. Nowadays the most common stream cipher used is ChaCha, which is based on Salsa20 we mentioned earlier, and its variations.
Although stream ciphers are not especially widely used, it has been shown that they have their own niche, where they have considerable advantages over block ciphers and therefore more research should be put into this type of ciphers.

List of Figures

References

[1] B. Y. Zhang and G. Gong, "Randomness properties of stream ciphers for wireless communications," *The Sixth International Workshop on Signal Design and Its Applications in Communications*, pp. 107–109, 2014. DOI: 10.1109/IWSDA.2013.6849074.

[2] J. von Neumann, "Various techniques used in connection with random digits," in *Monte Carlo Method*, ser. National Bureau of Standards Applied Mathematics Series, A. S. Householder, G. E. Forsythe, and H. H. Germond, Eds., vol. 12, Washington, DC: US Government Printing Office, 1951, ch. 13, pp. 36–38.

[3] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography* (CRC Press series on discrete mathematics and its applications), rev. reprint with updates, 5. printing. Boca Raton: CRC Press, 2001, ISBN: 0-8493-8523-7.

[4] B. Schneier, *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, 2nd edition. München: Pearson Studium, 2006, ISBN: 3-8273-7228-3.

[5] K. Schmeh, *Kryptografie: Verfahren, Protokolle, Infrastrukturen*, 6th edition. Heidelberg: dpunkt.verlag, 2016, ISBN: 978-3-86490-356-4.

[6] A. Beutelspacher, H. B. Neumann, and T. Schwarzpaul, *Kryptografie in Theorie und Praxis: Mathematische Grundlagen für elektronisches Geld, Internetsicherheit und Mobilfunk*, 1st edition. Wiesbaden: Vieweg+Teubner Verlag, 2005, ISBN: 3-528-03168-9. DOI: 10.1007/978-3-322-93902-9.

[7] W. Ertel and E. Löhmann, *Angewandte Kryptographie*, 6th edition. München: Hanser, 2020, ISBN: 978-3-446-46353-0.

[8] S. W. Golomb, *Shift register sequences: Secure and limited-access code generators, efficiency code generators, prescribed property generators, mathematical models*, 1st edition. San Francisco: Holden-Day, Inc., 1967.

[9] M. Stamp and R. M. Low, *Applied cryptanalysis: Breaking ciphers in the real world*, 1st edition. Hoboken, New Jersey: Wiley-Interscience a John Wiley & Sons Inc., 2007, ISBN: 978-0-470-1-1486-5.

[10] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, 1st edition. Cambridge: Cambridge University Press, 1986, ISBN: 0-521-30706-6.

[11] N. P. Smart, *Cryptography made simple* (Information security and cryptography), 1st edition. Cham et al.: Springer, 2016, ISBN: 978-3-319-21936-3.

[12] R. A. Rueppel, *Analysis and Design of Stream Ciphers* (Communications and Control Engineering Series), 1st edition. Berlin and Heidelberg: Springer, 1986, ISBN: 978-3-642-82865-2. DOI: 10.1007/978-3-642-82865-2.

[13] R. Lidl and H. Niederreiter, *Finite fields*, 2nd edition. Cambridge: Cambridge University Press, 1997, ISBN: 978-0-521-06567-2. DOI: 10.1017/CBO9780511525926.

[14] G. Gong, T. Helleseth, and P. V. Kumar, "Solomon w. golomb—mathematician, engineer, and pioneer," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 2844–2857, 2018, ISSN: 0018-9448. DOI: 10.1109/TIT.2018.2809497.

[15] M. J. Robshaw, "Stream ciphers," *RSA Labratories*, vol. 25, 1995. [Online]. Available: http://www.networkdls.com/Articles/tr-701.pdf (visited on 04/29/2022).

[16] K. Pommerening, *Cryptology part iv: Bitstream ciphers*, 2000. [Online]. Available: https://www.staff.uni-mainz.de/pommeren/Cryptology/Bitstream/ (visited on 05/13/2022).

[17] C. Eckert, *IT-Sicherheit: Konzepte - Verfahren - Protokolle*, 10th edition. Berlin/Boston: De Gruyter, 2018, ISBN: 978-3-11-055158-7. DOI: 10.1515/9783110563900.

[18] K. Graves, *CEH certified ethical hacker study guide*. John Wiley & Sons, 2010.

[19] T. W. Cusick and P. Stănică, *Cryptographic Boolean functions and applications*, 1st ed. Amsterdam and Boston: Academic Press/Elsevier, 2009, ISBN: 978-0-08-095222-2. [Online]. Available: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10286068.

[20] J. Massey, "Shift-register synthesis and bch decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969, ISSN: 0018-9448. DOI: 10.1109/TIT.1969.1054260.

[21] K. Pommerening, "Cryptanalysis of nonlinear feedback shift registers," *Cryptologia*, vol. 40, no. 4, pp. 303–315, 2015, ISSN: 0161-1194. DOI: 10.1080/01611194.2015.1055385. (visited on 05/13/2022).

[22] F. Handayani and N. P. R. Adiati, "Analysis of geffe generator lfsr properties on the application of algebraic attack," ser. AIP Conference Proceedings, AIP Publishing, 2019, p. 020 029. DOI: 10.1063/1.5132456.

[23] T. Siegenthaler, "Correlation-immunity of nonlinear combining functions for cryptographic applications (corresp.)," *IEEE Transactions on Information Theory*, vol. 30, no. 5, pp. 776–780, 1984, ISSN: 0018-9448. DOI: 10.1109/TIT.1984.1056949.

[24] H. Wu, "Cryptanalysis and design of stream ciphers," Ph.D. dissertation, 2008. [Online]. Available: https://www.esat.kuleuven.be/cosic/publications/thesis-167.pdf (visited on 04/25/2022).

[25] R. A. Rueppel, "Correlation immunity and the summation generator," in *Advances in Cryptology — CRYPTO '85 Proceedings*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed., vol. 218, Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 260–272, ISBN: 978-3-540-16463-0. DOI: 10.1007/3-540-39799-X_20. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-39799-X_20.pdf (visited on 05/15/2022).

[26] W. Meier and O. Staffelbach, "Correlation properties of combiners with memory in stream ciphers," *Journal of Cryptology*, vol. 5, no. 1, pp. 67–86, 1992, ISSN: 0933-2790. DOI: 10.1007/BF00191322.

[27] T. Beth and F. C. Piper, "The stop-and-go-generator," in *Advances in Cryptology*, ser. Lecture Notes in Computer Science, T. Beth, N. Cot, and I. Ingemarsson, Eds., vol. 209, Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 88–92, ISBN: 978-3-540-16076-2. DOI: 10.1007/3-540-39757-4_9.

[28] A. Klein, *Stream ciphers*. London: Springer, 2013, ISBN: 9781447150787.

[29] D. Coppersmith, H. Krawczyk, and Y. Mansour, "The shrinking generator," in *Annual International Cryptology Conference*, Springer, 1993, pp. 22–39.

[30] M. Robshaw, "The estream project," in *New Stream Cipher Designs*, Springer, 2008, pp. 1–6.

[31] C. D. Cannière, "Estream software performance," in *New Stream Cipher Designs*, Springer, 2008, pp. 119–139.

[32] T. Good and M. Benaissa, "Asic hardware performance," in *New stream cipher designs*, Springer, 2008, pp. 267–293.

[33] L. Jiao, Y. Hao, and D. Feng, "Stream cipher designs: A review," *Science China Information Sciences*, vol. 63, no. 3, pp. 1–25, 2020.

[34] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner, "The stream cipher rabbit," *ECRYPT Stream Cipher Project Report*, vol. 6, p. 28, 2005.

[35] S. Jarosław, "Low-cost hardware implementations of salsa 20 stream cipher in programmable devices," 2013.

[36] M. Boesgaard, M. Vesterager, and E. Zenner, "The rabbit stream cipher," in *New stream cipher designs*, Springer, 2008, pp. 69–83.

[37] D. J. Bernstein, "The salsa20 family of stream ciphers," in *New stream cipher designs*, Springer, 2008, pp. 84–97.

[38] F. E. Potestad-Ordóñez, M. Valencia-Barrero, C. Baena-Oliva, P. Parra-Fernández, and C. J. Jiménez-Fernández, "Breaking trivium stream cipher implemented in asic using experimental attacks and dfa," *Sensors*, vol. 20, no. 23, p. 6909, 2020.

[39] D. Roy and D. Dalai, "An observation of non-randomness in nfsr-based stream ciphers with reduced initialization round," *Journal of Hardware and Systems Security*, vol. 5, Jun. 2021. DOI: 10.1007/s41635-021-00113-5.

[40] C. D. Cannière and B. Preneel, "Trivium," in *New stream cipher designs*, Springer, 2008, pp. 244–266.

[41] M. Hell, T. Johansson, and W. Meier, "Grain: A stream cipher for constrained environments," *International journal of wireless and mobile computing*, vol. 2, no. 1, pp. 86–93, 2007.

[42] Ö. Küçük, "Slide resynchronization attack on the initialization of grain 1.0," *eSTREAM, ECRYPT Stream Cipher Project, Report*, vol. 44, p. 2006, 2006.

[43] S. Babbage and M. Dodd, "The stream cipher mickey 2.0," *ECRYPT Stream Cipher*, pp. 191–209, 2006.

[44] C. Berbain, O. Billet, A. Canteaut, *et al.*, "Sosemanuk: A fast software-oriented stream cipher," vol. 4986, Nov. 2008. DOI: 10.1007/978-3-540-68351-3_9.

[45] S. Banik, S. Maitra, and S. Sarkar, "Improved differential fault attack on mickey 2.0," *Journal of Cryptographic Engineering*, vol. 5, no. 1, pp. 13–29, 2015.

[46] C. Berbain, O. Billet, A. Canteaut, *et al.*, "Sosemanuk, a fast software-oriented stream cipher," in *New stream cipher designs*, Springer, 2008, pp. 98–118.

[47] Y. Tsunoo, T. Saito, M. Shigeri, *et al.*, "Evaluation of sosemanuk with regard to guess-and-determine attacks," *SASC 2006 Stream Ciphers Revisited*, p. 25, 2006.

[48] J.-K. Lee, D. H. Lee, and S. Park, "Cryptanalysis of sosemanuk and snow 2.0 using linear masks," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2008, pp. 524–538.

[49] S. Raizada, "Some results on analysis and implementation of hc-128 stream cipher," 2015.

[50] H. Wu, "The stream cipher hc-128," in *New stream cipher designs*, Springer, 2008, pp. 39–47.

[51] P.-A. Fouque and T. Vannet, "Improving key recovery to 784 and 799 rounds of trivium using optimized cube attacks," in *International Workshop on Fast Software Encryption*, Springer, 2013, pp. 502–517.