Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science

Master's Thesis

Random number generator based on multiplicative convolution transform

Nikolai Antonov

Supervisor: doc. RNDr. Daniel Prusa, Ph.D.

Study Program: Open Informatics

Field of Study: Software Engineering

May 22, 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Antonov Nikolai**	Personal ID number:	**492133**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Random number generator based on multiplicative convolution transform

Master’s thesis title in Czech:

Generátor náhodných čísel založený na multiplikativní konvoluční transformaci

Guidelines:

Develop the system for generating and testing sequences of pseudorandom bits. The main goal is to create and implement the model of random number generator(RNG), based on non-linear transformation called "multiplicative convolution". Another goal is to show cryptographic properties of created RNG and its resistance to potential attacks. Finally, show the statistical properties of generated pseudorandom sequences, using graphical tests and different statistical tests from modern statistical packages.

Bibliography / sources:

[1] Schneier B. Applied Cryptography. Protocols, algorithms, source code at C. - M.: Triumph, 2002. - 816 p. [2] W. Meier, O.Staffelbach. Fast correlation attack on stream ciphers. Journal of Cryptology, 1989. – V.1, N.3

Name and workplace of master’s thesis supervisor:

doc. RNDr. Daniel Průša, Ph.D., Machine Learning, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **18.03.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **19.02.2022**

doc. RNDr. Daniel Průša, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my mother Maria Maiorova and my granny Tatyana Maiorova for their incredible support, which anyone can only dream of. I am also grateful to my supervisors: Shamil Talgatovich Ishmukhametov (Kazan Federal University, Russia) for proposing an interesting problem and most importantly for the skill to interest me with applied mathematics and number theory; Mr. Daniel Prusa (Czech Technical University in Prague, Czech Republic) for his absolute kindness to me, the attention to the moments of my scientific difficulties and excellent advices on how to solve them. Finally, I am grateful to our tutor, Mrs. Ekaterina Turilova, the Department of Computer Science and the leadership of both universities for the opportunity to study under the Open Informatics program.

DECLARATION

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, May, 2020

ABSTRACT

ANTONOV, Nikolai: Random number generator based on multiplicative convolution transform. [Master’s Thesis] - Czech Technical University in Prague. Faculty of Electrical Engineering, Department of Computer Science. Supervisor: doc. RNDr. Daniel Prusa, Ph.D.

The creation of algorithmic random number generators is an important issue in the field of mathematical modeling, applied mathematics and modern cryptography. Despite the fact that many concepts and final solutions have been proposed to solve this problem, the well-known algorithmic generators are characterized by a tradeoff between their statistical properties, resistance to algebraic attacks and the speed of production of pseudorandom sequences.

The object of the thesis is to design and implement a system for generating pseudorandom bits that is free from the compromise mentioned above.

The introduction is considered about the stream ciphering model and the existing methods of producing pseudorandom sequences.

The first chapter proposes the new method of generating the pseudorandom sequences, using the multiplicative convolution transform to equip the ordinary shift register.

The second chapter presents the detailed report about the implementation of proposed ideas, as well as the details of code optimization.

The third chapter contains the results of graphical and statistical tests, theoretic and empirical estimates that prove the overall cryptographic resistance of the developed system.

The result of the master's thesis is the random number generator, which is based on non-linear transformation called "Multiplicative Convolution" and can produce high-quality pseudorandom sequences at good speed.

Keywords: multiplicative convolution transform, random number generator, pseudorandom sequences, stream cipher

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Consider the stream ciphering model

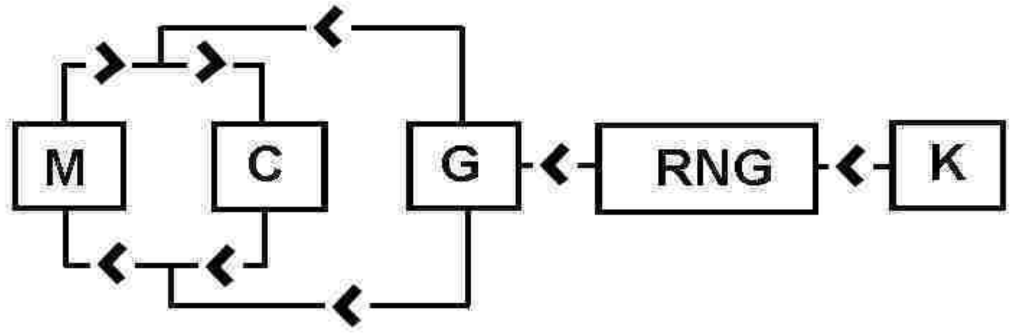


Fig. 1. Stream ciphering model

The first step in encrypting message M is to initialize the random number generator RNG with key K. Then the generator RNG creates a numerical sequence G with the properties of a sample from uniform distribution. This sequence usually consists of bits (zeros and ones) and called «gamma». It's important to notice that the length of the gamma G should be the same as the length of the message M. To obtain the ciphertext (cipher) C from the plaintext (message) M, there should be performed a bitwise addition of M and G. Decryption of the cipher C is also performed using bitwise addition — in this case, the cipher C and gamma G are the arguments, and the result is again the message M.

The cryptographic strength of the presented encryption method is based on the well-known fact from theories of signals and coding: the result of the interference of two signals, one of which carries some information, and the other is white noise, is white noise again. This means that the cipher C of message M has the characteristics of white noise and cannot give the adversary any information about the original message M.

Careful study of the proposed encryption model shows that the cryptographic strength of ciphering entirely depends on the properties of gamma created by the random number generator. The most important requirement for a gamma is its correspondence to the random sample from a uniform distribution. It is also important to generate the gamma with the same length as the length of a given message — encryption with a shorter gamma, as well as the use of the same gamma to encrypt another message, contributes to cracking the cipher.

It is rather easy to follow the rules regarding the length of the gamma and the frequency of its using, if a sufficiently long sample of bits corresponding to a uniform distribution is available. A much more difficult task is to produce such gamma. There are few sources of true randomness, but for the purposes of mathematical modeling, industrial informatics and the cryptography even more (including military cryptography) a random number generator must satisfy several requirements at once, and some of them seems to contradict the very the nature of true randomness.

For example, one of the most important requirements for the random number generator is the reproducibility of the gamma — the ability to get the same random sequence once again at any moment. It was that very requirement, which led to the creation of algorithmic random number generators. Any generator of this type operates on a deterministic algorithm, but all the generated sequences are completely predictable only if you know the initial state of the generator. For example, the key K from the stream ciphering model is the initial state of the RNG random number generator: to decrypt the cipher C you will need to generate the same gamma that was used for the encryption, which means that the generator must be again initialized by the same key K — the initial state that was set for encrypting the message M. It is important to note that, despite such obvious predictability, the generated sequences should have all the properties of random sample from uniform distribution. Otherwise, the gamma will be either a bad mask for message M or reveal the dependence on the initial state (key) of the generator, by which the key can be extracted.

The result of all given above is the fundamental problem of constructing algorithmic random number generators. The sequences they produce are also called pseudorandom (or pseudo noise) sequences, and the algorithmic generators themselves are called pseudorandom number generators. The problem of constructing pseudorandom number generators remains relevant to this day, and at the same time, considerable experience has been accumulated in this area.

Linear congruent generators

A pseudorandom number generator of this type generates a recurrent sequence according to the following formula

$$X_n = (a \cdot X_{n-1} + b) \mod m \,,$$

where

X_n is currently generated element of sequence;

X_{n-1} is previous element in the sequence;

and the parameters a, b, m $(0 < a < m,\ 0 \leq b < m,\ m > 0)$ are fixed during the entire process of production the pseudorandom numbers. An element X_0 serves as the initial state of the generator (key).

Like the most of algorithmic random number generators, this generator is periodic, and the period does not exceed m and can be maximum for a certain choice of parameters a and b.

Example: let $a = 7, b = 5, m = 18; X_0 = 4.$

Then we get the following output sequence:

4, 15, 2, 1, 12, 17, 16, 9, 14, 13, 6, 11, 10, 3, 8, 7, 0, 5, | 4, 15, 2, 1, 12, 17, 16, ……

The sequence begins to repeat after *18-th* element.

The significant advantage of a linear congruent generator is high speed of producing pseudorandom numbers, however, it is not a good idea to use this generator for encryption purposes: it was shown in [1], [2], [3], [4] that a linear congruent generator, as well as quadratic and cubic generators of formulas

$$X_n = (a \cdot X_{n-1}^2 + b \cdot X_{n-1} + c) \bmod m$$

$$X_n = (a \cdot X_{n-1}^3 + b \cdot X_{n-1}^2 + c \cdot X_{n-1} + d) \bmod m$$

can be hacked in a reasonable amount of time. In later works [5], [6], [7] a method for cracking any polynomial congruent generator as well as a truncated generator of this type was developed.

Thus, the linear congruent generator is useless for cryptographic purposes, but remains relevant and is used in computational modeling tasks.

Shift registers with linear feedback

The principles of constructing pseudorandom number generators based on shift registers have been studied by cryptographers since the first half of the last century. The sequences generated by shift registers are used both in cryptography and in coding theory. Stream ciphers based on shift registers served as a working tool of military cryptography long before the electronics.

In general, generators of this type contain

1. register — a set of cells, and you can put only one bit of information (zero or one) into each cell
2. feedback mechanism (some function of register bits)

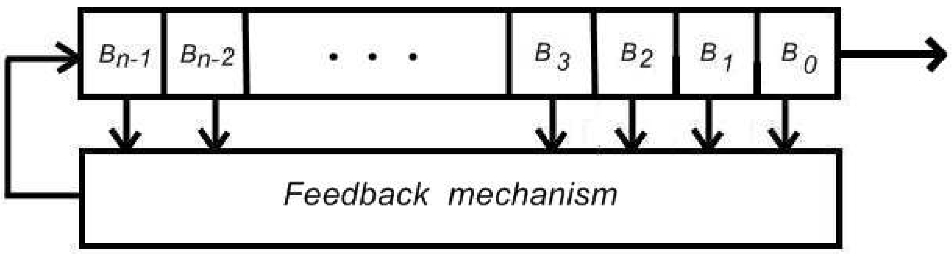
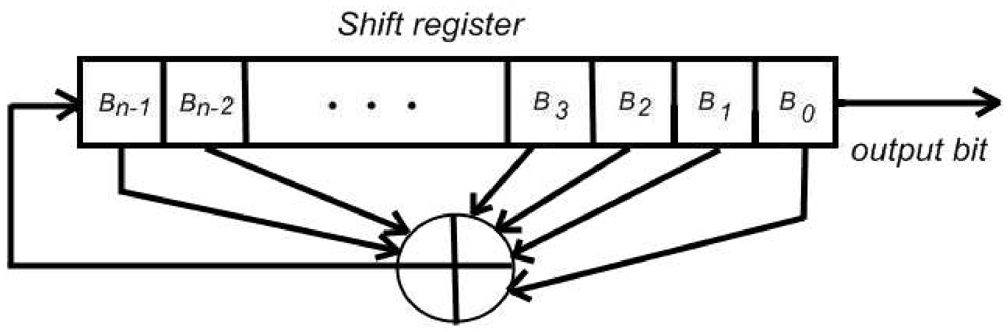


Fig. 2. The scheme of the register with feedback.

The shift register can be considered as a sequence of bits. The length of the shift register is equal to the number of bits — if it is equal to n bits, the register is called an *n-bit* shift register. Each time a bit is extracted, all other bits of the shift register are shifted to the right by one position. The new bit (feedback bit) is calculated as a function of all other bits of the register. The output of the shift register is the rightmost bit each time and it's called an output bit. The period of the shift register is the length of the output sequence before it starts to repeat.

A particular case of a shift register is a linear feedback shift register (abbreviated as *LFSR*). The feedback function is defined as the addition modulo two of some bits of the register. The list of these bits is called a sequence of taps (or picks points). Sometimes such a scheme is called a Fibonacci configuration. Due to the simplicity of the feedback sequence, a fairly developed mathematical theory can be used to analyze *LFSR*.



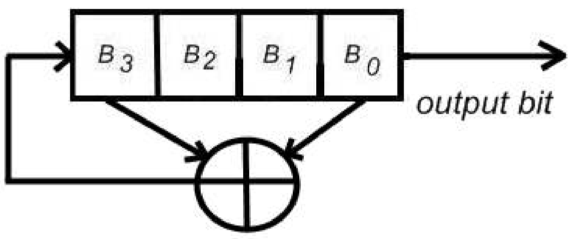


Fig. 3. Linear feedback shift register.

Example:

The figure shows a *4-bit LFSR* with taps from the first and fourth bits. If you initialize it with a set of bits *{1; 1; 1; 1}*, the register will take the following internal states until they start to repeat:

1111, 0111, 1011, 0101,
1010, 1101, 0110, 0011,
1001, 0100, 0010, 0001
1000, 1100, 1110

The output sequence of the register will be a string of the least significant bits:

1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 | ... the sequence starts to repeat ....

The *n-bit* register can be in one of $2^n - 1$ internal states at each moment of time. This means that, theoretically, such register can generate a pseudorandom sequence with a period of $2^n - 1$ bits. (This number equals exactly $2^n - 1$, not 2^n, because filling LFSR with zeros leads to the infinite sequence of zeros in the output, which is completely useless.)

Only with certain sets of taps the *LFSR* will cycle through all $2^n - 1$ internal states. Such *LFSR* are called maximum period registers. The resulting output is called the M-sequence. To ensure the maximum period of a particular *LFSR*, the corresponding polynomial formed from the sequence of register's taps should be primitive by modulo *2* (or, which is the same, should be primitive in the field *GF(2)*). The degree of the polynomial coincides with the length of the shift register.

In the general case, there is no simple way to generate primitive polynomials of a given degree in the field *GF(2)*. The easiest way is to choose a random polynomial and check if it is primitive or not. This is not easy and similar to checking the primality of a given random number, but many packages of mathematical programs can solve this problem. The software implementations of primitive trinomials work faster than the rest, because you need to perform the *XOR* operation only on two bits of the shift register to generate a new bit. For cryptographic algorithms, it is much more efficient to use dense primitive polynomials, i.e. polynomials with a large number of coefficients. When applying dense polynomials, especially as part of a key, the significantly shorter *LFSRs* can be used.

The linear feedback shift registers are considered to be good pseudorandom numbers generators; however, they have some undesirable nonrandom properties. The consecutive bits are subject to a linear dependency, which makes them useless for encryption. The internal state of the *LFSR* of length k determines the next k output bits of the generator, and even if the feedback polynomial is kept in secret, it can be determined by the $2k$ output bits of the generator using the efficient Berlekamp-Massey algorithm. In addition, a big pseudorandom numbers can be highly correlated if they are generated using consecutive bits. However, *LFSRs* are often used in encryption algorithms.

Geffe's random number generator

This random number generator uses a non-linear combination of three *LFSRs*. Two *LFSRs* serve as inputs to the multiplexer, and the third one controls the output of the multiplexer.

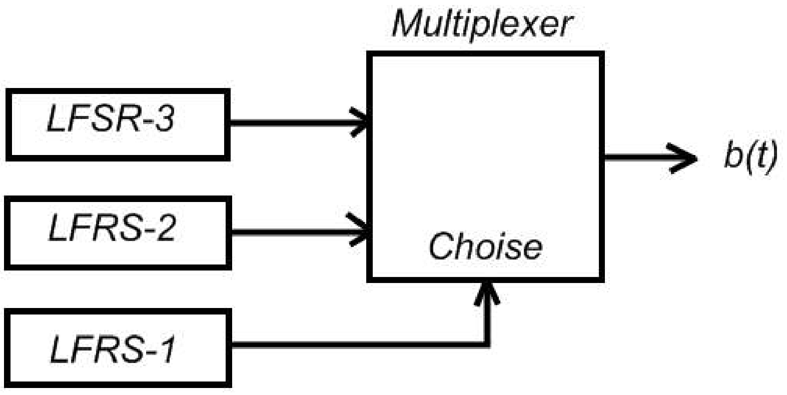


Fig. 4. Geffe's generator

If *a1*, *a2*, *a3* are the outputs of three *LFSRs*, the output of the generator can be described as follows:

$$b = (a_1 \wedge a_2) \oplus ((\neg a_1) \wedge a_3)$$

If the lengths of the *LFSRs* are *n1*, *n2* and *n3*, respectively, then the linear complexity of the generator is equal to:

$$(n_1 + 1) \cdot n_2 + n_1 \cdot n_3$$

The period of the generator is equal to the least common multiple of the periods of three registers. Provided that the degrees of the three primitive feedback polynomials are co-prime, the period of this generator will be equal to the product of the periods of three *LFSRs*. Although this generator looks good at the first glance, it is unreliable in cryptographic terms and unstable to a correlation attack [8], [9]. Over *75%* of the operating time, the generator output is equal to the output of *LFSR-2*. Therefore, if the sequence of feedback taps is known, you can determine the initial value of *LFSR-2* and generate the output sequence of this register. Next, you can calculate how many times the output of *LFSR-2* coincides with the output of the generator. If the initial value is not determined correctly, the two sequences will be matched at *50%* of the time, and if correct, then at *75%* of the time. Similarly, the output of the generator is equal to the output of *LFSR-3* in *75%* of the time. With such correlations this generator can be easily cracked. For example, if primitive polynomials are trinomials and the length of the largest *LFSR* is n, then to restore the internal states of all three *LFSRs*, you need a fragment of the output sequence with a length of $37n$ bits [10].

Threshold generator

In this generator there was made an attempt to overcome the problems with the *RNG's* stability and resistance using a variable number of *LFSRs* [11]. The threshold generator always includes an odd number of registers. To get the maximum period, you need to make sure that the lengths of all *LFSRs* are co-prime, and the feedback polynomials are primitive in *GF(2)*. If more than half of the *LFSR* output bits are equal to *1*, then the output of the generator will be *1*. If more than half of the *LFSR* output bits are *0*, then the generator output is *0*.

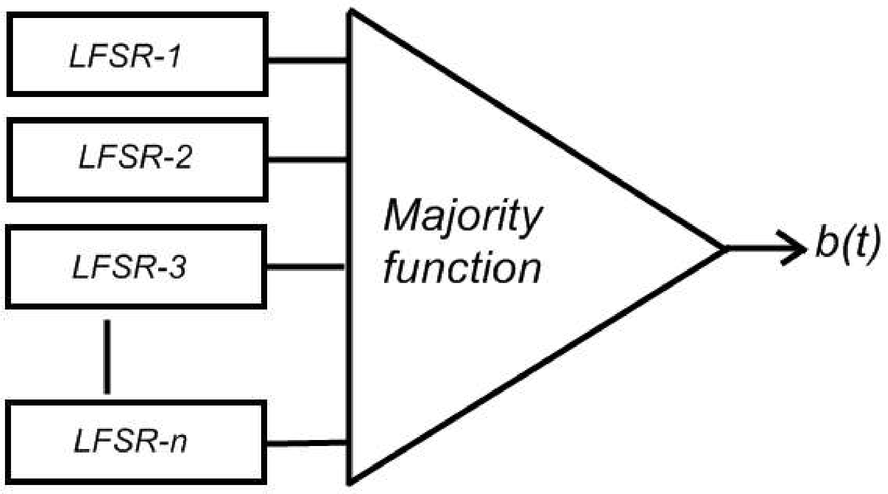


Fig. 5. The scheme of the threshold generator

For three *LFSRs*, the generator output can be represented as:

$$b = (a_1 \wedge a_2) \oplus (a_1 \wedge a_3) \oplus (a_2 \wedge a_3)$$

This equation is very similar to that used in the Geffe's generator. However, the threshold generator has the greater value of linear complexity:

$$n_1 \cdot n_2 + n_1 \cdot n_3 + n_2 \cdot n_3,$$

where *n1*, *n2* and *n3* are the lengths of the first, second and third *LFSR*.

This generator is not very good at its resistance to attacks. Each output bit gives *0.189* bits of information about the state of the *LFSR*, and the generator itself cannot resist to correlation attack.

Cascade generator

The Gollmann cascade described in [12], [13] is an enhanced version of the so called «start-stop» generator.

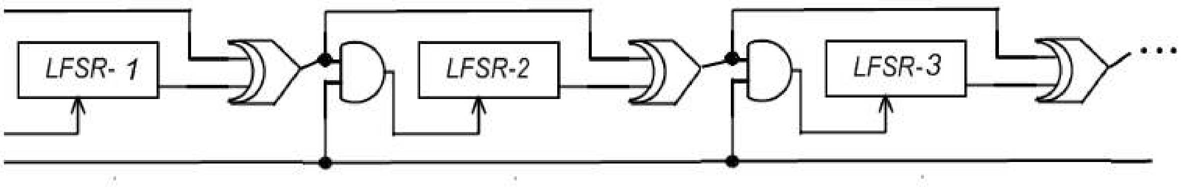


Fig. 6. The scheme of the cascade generator

It consists of a sequence of *LFSRs,* where each current register is controlled by the previous one. If at time moment *t* the output of *LFSR-1* is equal to *1*, then *LFSR-2* performs the tact. If at time moment *t* the output of *LFSR-2* is *1*, then *LFSR-3* performs the tact, and so on. The output of the last *LFSR* is the output of the generator. If the length of all *LFSRs* is the same and equals to *n*, then the linear complexity of the system of *k* registers of *LFSR* is equal to

$$n(2^n - 1)^{k-1}$$

Using cascades is a great idea: conceptually they are very simple and can be used to generate sequences with huge periods, high linear complexity and good statistical properties. However, they are vulnerable to attack called lock-in [13], that represents a method by which the adversary restores, firstly, the input of the last shift register in a cascade, and then breaks the entire cascade, one register by another. In some cases, this becomes a serious problem and reduces the effective length of the algorithm key. Subsequent analysis of Gallmann cascades showed that with the increasing of parameter *k*, the output sequences approach random [14], [15], [16]. The use of a large number of short *LFSRs* in a cascade generator is considered more preferable than a small number of long *LFSRs*, and the cascade length should be chosen at least fifteen [17].

Algorithm A5

The A5 cipher is a stream cipher used to encrypt *GSM* communications (*GSM* - Group Special Mobile - mobile group special communications). This is the European standard for mobile digital cell phones. It is used to encrypt the telephone / base station channel. The code of A5 is described in details in [18].

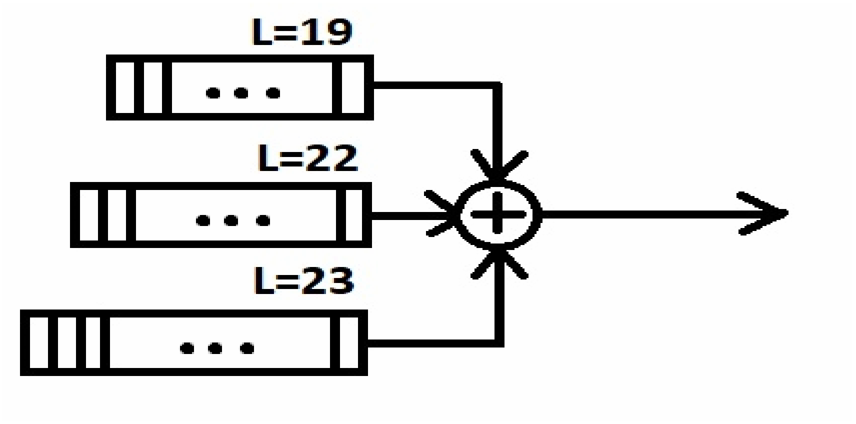


Fig. 7. The scheme of the random number generator involved in A5

The random number generator in A5 algorithm consists of three *LFSRs* with the lengths equal to *19, 22* and *23* bits, and all the feedback polynomials are sparse. The output is the result of the *XOR* operation on three output bits of *LFSRs*. There is a trivial cracking algorithm against A5, which takes about 2^{40} operations and possible for modern supercomputers. Despite this fact, the ideas underlying the A5 remains actual, and the generator itself is very efficient. It satisfies all of the known statistical tests, and its only weakness is that the registers are too short to prevent the cracking by bruteforcing.

CHAPTER 1. THE RANDOM NUMBER GENERATOR BASED ON MULTIPLICATIVE CONVOLUTION TRANSFORM

The algorithm for generating pseudorandom bits based on linear feedback shift register

Consider a shift register of length N with linear feedback defined by the polynomial $P = P(x)$ primitive over the field $GF(2)$.

Let us set the register to an arbitrary state S, where at least one bit is not equal to zero. The generation of a pseudorandom bits sequence with the length equal to M consists in repeating M times the following algorithm:

1) Extract the last bit of the register

2) Calculate the feedback bit in accordance with the primitive polynomial $P(x)$

3) Perform a register shift towards the output and place the feedback bit in the freed cell farthest from the output.

Let us save the design of the linear feedback shift register as the basis of our generator. From the other hand, let's take into account not only the bits of the register, but also their order numbers. This will gradually lead us to a new approach for obtaining pseudorandom bits — the linear feedback shift register equipped with multiplicative convolution transform.

The definition of multiplicative convolution

Let us have a prime number p, which satisfies the condition

$$p = 4t + 3,$$

where t is a natural number. Consider a nonzero vector $S = (b_1, b_2, ..., b_{p-1})$, from the set $\{0, 1\}^{p-1}$. Elements $b_1, b_2, ..., b_{p-1}$ of the vector S can also be called bits, each of them is either zero or one. The multiplicative convolution operation is performed both on the bits of the vector S and their order numbers, moreover, all operations are carried out over the $GF(p)$ field. The transformation results in a single bit (zero or one). Thus, the formal definition of the multiplicative convolution transformation is as follows:

$$f = f(b_1,\ b_2,\ ...,\ b_{p-1} \mid GF(p)),\ E(f)\ \in\ \{0,\ 1\}$$

Let us now give an algorithm for generating pseudorandom bits using a linear feedback shift register equipped with multiplicative convolution transform.

So, let us have a prime number p, which satisfies the condition

$$p = 4t + 3,$$

where t is a natural number. Consider a shift register with the length of $L = p - 1$ bits with linear feedback expressed by the primitive polynomial $P = P(x)$ over the field $GF(2)$. Let us apply a new approach to the generation of pseudorandom bits — we will perform one tact of the register using the multiplicative convolution transform.

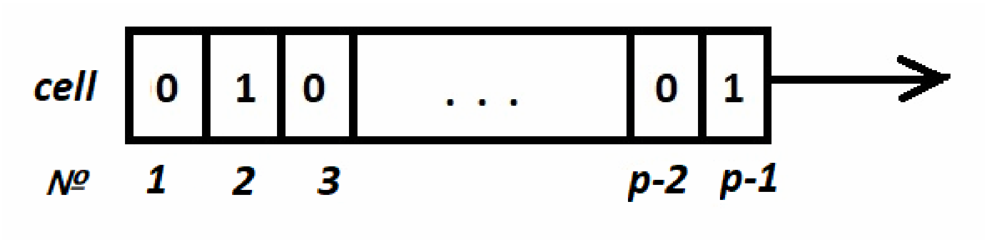


Fig. 8. Ready for multiplicative convolution transform

In a situation with a register equipped with a multiplicative convolution, the output bit is not just the last bit of the register. The output bit is obtained as a result of executing the algorithm on the current state of the register and the set of cell order numbers. The mentioned algorithm consists of the following steps:

1. Consider all the cells of the register that contain zero, and multiply their numbers in the field $GF(p)$. (We get some natural number N: $0 < N < p$)

2. Consider all the cells of the register that contain one, and also multiply their numbers in the $GF(p)$ field. (We get some natural number E: $0 < E < p$)

3. Form the output bit by adding on modulo 2 the bits found in cells under the numbers N and E

4. Calculate the feedback bit by applying the polynomial $P(x)$ to the current state of the register, and then perform the shift — all in the same way as the feedback is calculated when we deal with the usual shift register with linear feedback.

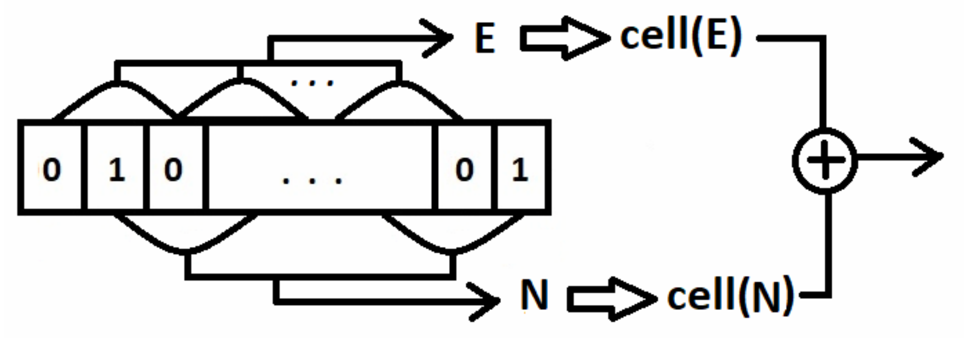


Fig. 9. Model of multiplicative convolution transform

Lemma

The numbers N and E from the algorithm above are always different.

Proof

The product of the numbers N and E is comparable by modulo p with the product of all numbers from *1* to *(p - 1)*

$$N \cdot E = 1 \cdot 2 \cdot \ldots \cdot (p-1) \pmod{p}$$

At the same time, according to Wilson's theorem

$$(p-1)! = -1 \pmod{p}$$

Hence,

$$N \cdot E = -1 \pmod{p}$$

If the numbers N and E are the same, this means that *-1* is a quadratic remainder in the field by prime modulo p of the form *4t + 3*. However, *-1* is not a quadratic remainder in any field formed by a prime modulo of this form. Thus, the numbers N and E are always different.

Remark

This lemma gives us the first reason to believe that the structure of the algorithm does not upset the balance of zeros and ones in the output sequence. The same frequency of zeros and ones in the output sequence will be later proved by the results of statistical tests.

Example

Consider a shift register with the length $L = 6$ and linear feedback defined by the polynomial $P(x) = x^6 + x^5 + 1$ primitive in the field $GF(2)$.

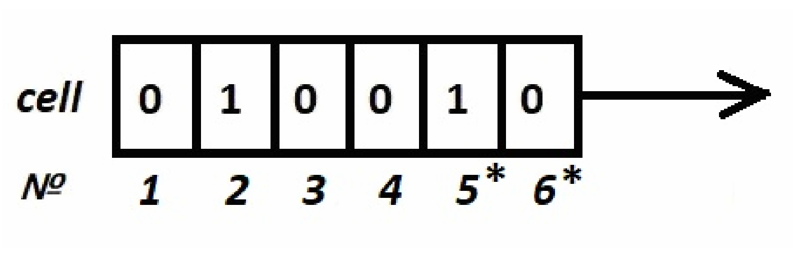


Fig. 10. Before an ordinary tact of LFSR

Let the initial state of the register be given by the bit vector *(0, 1, 0, 0, 1, 0)*. The last bit of the register, which is the closest one to the output, equals zero. So, the register output at this tact will be zero.

$$Output = 1$$

Let us compute the bit of feedback by adding by modulo *2* those bits that correspond to significant powers of the primitive polynomial *P (x)* — those are the bits in the fifth and sixth register cells.

$$Feedback = Cell(5) + Cell(6)\ \ (mod\ 2)\ \ \Rightarrow\ \ Feedback = 1$$

Then we perform the register shift and put the feedback bit in the freed cell.

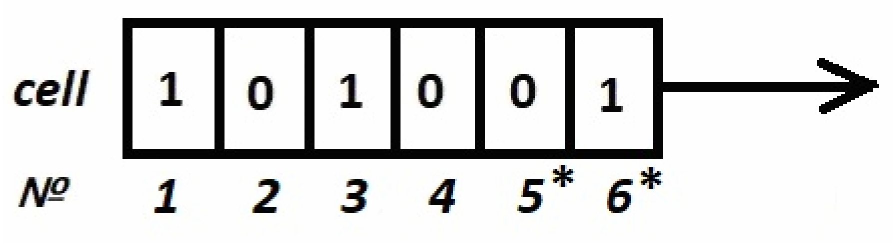


Fig. 11. After an ordinary tact of LFSR

Example

Consider the same shift register of length $L = p - 1 = 6$ with linear feedback expressed by the primitive polynomial in the field *GF (2)*. It is easy to verify that $p = 7$ is a prime and gives a remainder of *3* when divided by *4*.

Let the initial state of the register be given by the bit vector *(0, 1, 0, 0, 1, 0)*, as in the previous example. Register cells are numbered from *1* to *6:*

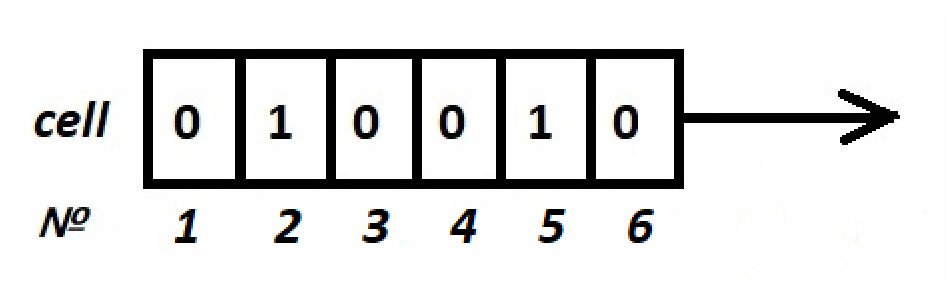


Fig. 12. Before a tact with multiplication convolution transform

According to the steps of the algorithm, we separately multiply the order numbers of all cells that contain zero and the order numbers of all cells that contain one:

$$N = 1 \cdot 3 \cdot 4 \cdot 6 \pmod{7} \;\Rightarrow\; N = 2$$

$$E = 2 \cdot 5 \pmod{7} \;\Rightarrow\; E = 3$$

Consider the contents of cells with order numbers 2 and 3:

$$Cell(2) = 1$$

$$Cell(3) = 0$$

The output bit is obtained by adding these bits, i.e.

$$1 + 0 = 1 \pmod{2} \Rightarrow Output = 1$$

Let us calculate the feedback bit by adding by modulo 2 those bits that correspond to significant powers of $P(x) = x^6 + x^5 + 1$ — those are the bits in the fifth and sixth register cells.

$$Feedback = Cell(5) + Cell(6) \pmod{2} \;\Rightarrow\; Feedback = 1$$

Then we perform the register shift and put the feedback bit in the freed cell.

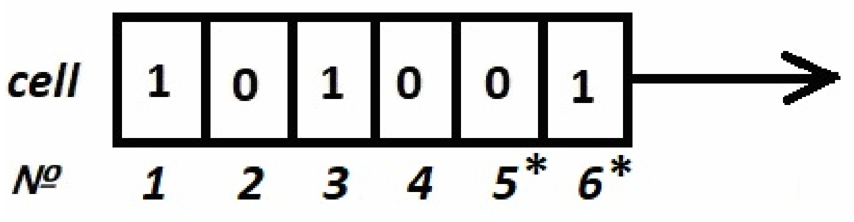


Fig. 13. After a tact with multiplication convolution transform

Remark

Note that the feedback calculation procedure is completely identical for both pseudorandom bit generation algorithms.

Complexity analyses. Linear feedback shift register

Consider a shift register of length equal to L bits with linear feedback expressed by the primitive polynomial $P = P(x)$ over the field $GF(2)$. Let us estimate the complexity of performing one register tact, that is, how many operations we need to produce one pseudorandom bit.

One tact of the linear feedback shift register consists of the following operations:

1) extracting the output bit

2) feedback calculation

3) register shift

4) placing the feedback bit in the freed cell

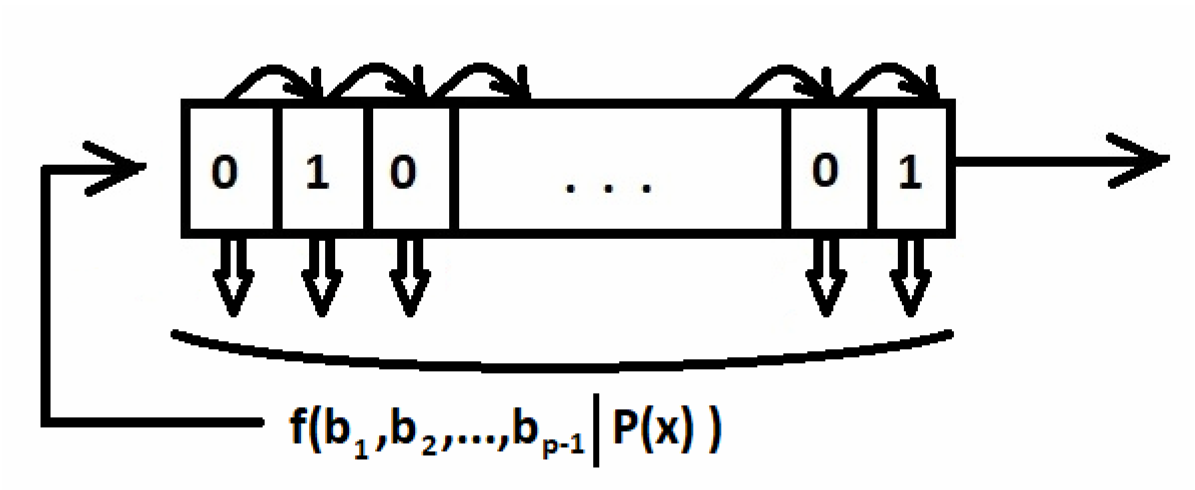


Fig. 14. The scheme of ordinary LFSR tact

The complexity of operation *2)* (feedback calculation) depends on the polynomial $P(x)$, and namely, on the number of its significant degrees. The more dense a polynomial is, that is, the more significant degrees it has, the more register cells form the feedback and the slower this operation is performed. And vice versa, when the polynomial is sparse (most of the coefficients at powers of $P(x)$ are zero), the calculation becomes faster. Thus, the difficulty of obtaining a feedback bit is proportional to the number of significant degrees of $P(x)$ and can be limited by the number L as the total length of the register. Therefore, the complexity of computing the feedback is $O(L)$.

The complexity of operation *3)* (register shift) is inherently proportional to the length of the register, however, due to the existence of effective solutions in the field of computer technology, its complexity is more justly estimated as a constant. So, the complexity of register shift is *O (1)*.

The complexity of operations *1)* and *4)* (extracting the output bit and putting the feedback bit in the empty cell) does not depend on the length of the register or on the primitive polynomial. Hence, it can be estimated as *O (1)*.

Adding individual estimates of the operations complexity, we obtain the desired estimate for the complexity of the one tact of linear feedback shift register:

$$Tact\_complexity = O(L)$$

Linear feedback shift register equipped with multiplicative convolution transform. Initial complexity analyses.

Let us have a prime number p, which satisfies the condition

$$p = 4t + 3,$$

where t is a natural number. Consider a shift register with the length of $L = p - 1$ bits and linear feedback expressed by the primitive polynomial $P = P(x)$ over the field $GF(2)$. Let us estimate the complexity of one tact of a register with multiplicative convolution transform.

One tact of such register consists of the following operations:

1) performing the multiplicative convolution transform over the current state of the register
2) extracting the output bit
3) calculation of feedback bit
4) register shift
5) placing the feedback bit in the freed cell

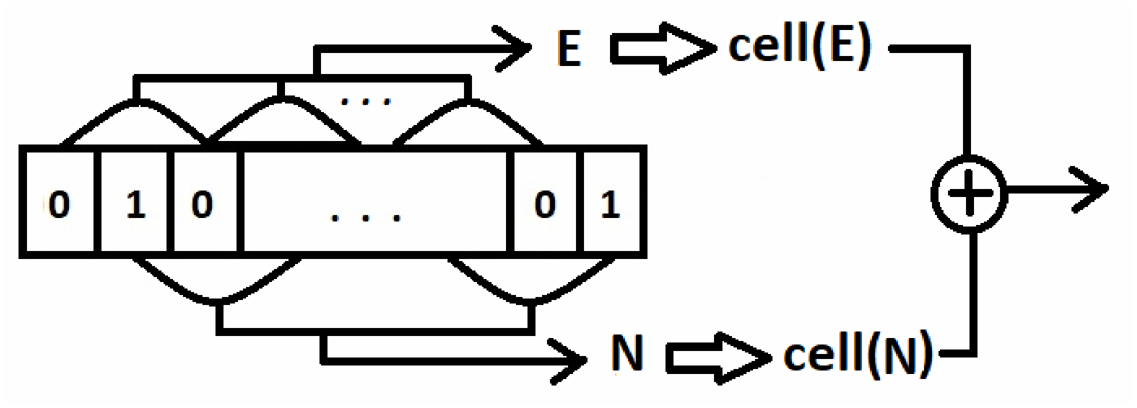


Fig. 15. Tact with multiplicative convolution transform

An analysis of operations *2) - 5)* was performed in the previous section - their total complexity is $O(L)$, which gives us an estimate of $O(p)$ with respect to the original prime number and register length.

We proceed to the estimate of complexity of multiplicative convolution transform. During its execution, it is necessary to perform $(p - 2)$ multiplication operations in the field $GF(p)$. This means that it is necessary to perform $(p - 2)$ ordinary multiplication operations and $(p - 2)$ division operations with the remainder by modulo p. The complexity of multiplying two numbers can be estimated as $O(loglog\ N)$, where N is the maximal of the two numbers. The complexity of taking the remainder (to finish the multiplication in the field $GF(m)$) is approximately $O(log\ m)$.

So, the multiplicative convolution transform is performed in no more than

$$(p-2)\cdot\log(\log(p))+(p-2)\cdot\log(p),$$

operations, which gives us an overall assessment of complexity

$$O(p\cdot(\log(p)).$$

Thus, the complexity of a single tact of the register equipped with a multiplicative convolution transform is the sum of the complexity of transformation and also the complexity of operations *2) - 5)*. It is equal to

$$O(p)+O(p\cdot(\log(p))$$

that gives us final estimate

$$O(p\cdot(\log(p))$$

Optimization of the algorithm for performing a multiplicative convolution

Comparing the results of the complexity analyses, it can be noted that the asymptotically higher complexity of the shift register, equipped with multiplicative convolution transform, grows out of a rather large number of multiplication operations in the field $GF(p)$ — it is almost p ordinary multiplication operations, followed by taking the remainder of division. The complexity of performing a multiplicative convolution on register state bits can be reduced if the multiplication operation is replaced by addition.

Let us take a closer look at the register cell order numbers — they are enumerated from 1 to $(p - 1)$, inclusive. These numbers form a cyclic multiplicative group $GF^*(p)$ modulo prime

p, which means that there exists a primitive element of the group — that is a such number that each register cell order number can be represented as a unique degree of the selected primitive element in $GF^*(p)$. Then the operation of multiplying the cell order numbers in the field $GF(p)$ can be replaced by adding the powers of the selected primitive element in $GF^*(p)$.

Example

Consider a prime number $p = 7$ and the register of length $L = p - 1 = 6$, equipped with a multiplicative convolution transform. Let the initial state of the register be set by the vector $(0, 1, 0, 0, 1, 0)$.

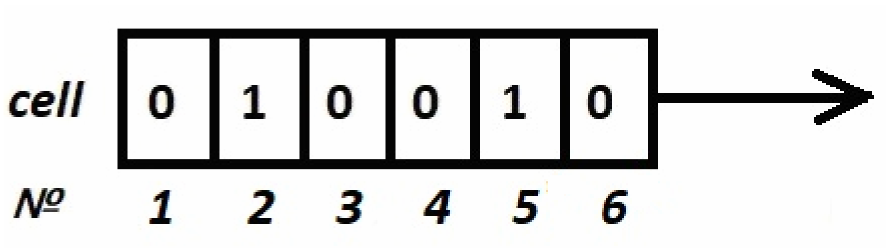


Fig. 16. Tact with improved multiplicative convolution transform

Let us perform a single tact of the register using an improved algorithm. At first, just for better clarity, we write the elements of the group $GF^*(p)$

$$GF^*(p) = GF(p) \setminus 0 = \{1, 2, 3, 4, 5, 6\}.$$

We proceed to the search for a primitive (generating) element. It is obvious that the remainder of division of any degree of one by modulo p is equal to one, and it cannot be a primitive element.

Let us check the next number:

$$2^1 = 2 \pmod 7,\ 2^2 = 4 \pmod 7,\ 2^3 = 1 \pmod 7,$$
$$2^4 = 2 \pmod 7,\ 2^5 = 4 \pmod 7,\ 2^6 = 1 \pmod 7.$$

We were not able to represent $3,\ 5$ and 6 as powers of two, which means that it is also not suitable as primitive element.

Let us check the three:

$$3^1 = 3 \pmod 7,\ 3^2 = 2 \pmod 7,\ 3^3 = 6 \pmod 7,$$
$$3^4 = 4 \pmod 7,\ 3^5 = 5 \pmod 7,\ 3^6 = 1 \pmod 7$$

We managed to represent all the elements of $GF^*(p)$ as comparable by modulo p with the unique degree of 3. Therefore, 3 is a primitive element.

Now we make the register tact. Let us separately multiply the order numbers of cells containing zero and the order numbers of cells which contain one:

$$N = 1\cdot 3\cdot 4\cdot 6 \ (mod\ 7) = 3^6 \cdot 3^1 \cdot 3^4 \cdot 3^3 \ (mod\ 7) = 3^{6+1+4+3} = 3^{14} \ (mod\ 7)$$

$$E = 2\cdot 5 \ (mod\ 7) = 3^2 \cdot 3^5 \ (mod\ 7) = 3^{2+5} = 3^7 \ (mod\ 7)$$

We apply the Fermat's little theorem to make sure that both powers of 3 belong to the interval from 0 to $(p - 2)$

$$N = 3^{14} \ (mod\ 7) = 3^{(14\ mod\ 6)} \ (mod\ 7) = 3^2 \ (mod\ 7) = 2$$

$$E = 3^7 \ (mod\ 7) = 3^{(7\ mod\ 6)} \ (mod\ 7) = 3^1 \ (mod\ 7) = 3$$

The output bit is the result of addition by modulo 2 of cells with order numbers 2 and 3.

New algorithm of multiplicative convolution transform

We extend the example above to a more general case and construct an optimized algorithm for performing multiplicative convolution over the bits of the register state.

Step 1.

For a given prime number p, find an arbitrary primitive element g of the group $GF^*(p)$.

Step 2.

For each element a of the group $GF^*(p)$, calculate its discrete logarithm by the base g. That means to find the degree d to which the element g should be raised that the result becomes comparable with the number a by modulo p

$$g^d = a \ (mod\ p)$$

The range of discrete logarithms should be the segment $[0; p-2]$. Save calculated values in available memory.

Step 3.

Calculate the remainders of dividing by the module p the powers of g taken from the interval of integers $[0; p-2]$, that is, find numbers $v_0, v_1, v_2, \dots, v_{p-2}$, such that

$$g^0 = v_0 \ (mod\ p),\ g^1 = v_1 \ (mod\ p),\ g^2 = v_2 \ (mod\ p),\ \dots,\ g^{p-2} = v_{p-2} \ (mod\ p)$$

Also save the calculated values in available memory as in the previous step.

Step 4.

Proceed to the calculation of the output bit – perform separate multiplication of the order numbers of cells containing zeros and ones, adding up the corresponding powers of the primitive element g.

Step 5.

As a result of the previous step, we get two different degrees of the primitive element g. The values of these degrees can be outside the interval $[0; p-2]$, and we apply the corollary of Fermat's little theorem

$$g^d = g^{(d \bmod (p-1))} \pmod p$$

As a result, we obtain two different degrees $d1, d2$ of the primitive element g, each of them is not less than zero and not bigger than $(p-2)$.

Step 6.

We calculate the remainder of dividing by p the degrees $d1, d2$ of the element g, by turning to the memory for the previously calculated results.

Hence, we obtain two unique numbers N and E.

Step 7.

We calculate the output bit by adding modulo 2 the bits contained in cells with order numbers N and E.

Step 8.

We calculate the feedback bit in accordance with the given polynomial $P(x)$, which is primitive in the field $GF(2)$. Then we shift the register and put the feedback bit in the cell farthest from the output of the register.

Complexity analyses of new algorithm

Steps 1, 2, 3.

There exist $\varphi(p-1)$ primitive elements in $GF^*(p)$, where φ – Euler function – determines the number of naturals from the segment $[1; p-2]$, which are co-prime with $(p-1)$.

In special case of performing this algorithm, a primitive element can be found by simple sequential checking of all possible numbers started from the number 2. Due to the large number of primitive elements in the group $GF^*(p)$, the search will end quickly enough and will have an amortized complexity of $O(p)$.

The calculation of discrete logarithms as well as exponentiation modulo p can be performed in the same cycle in p iterations. It should be noted that although the operation of raising to a power by a certain modulus has logarithmic complexity, for small numbers (in modern cryptography those are all numbers less than a billion) its complexity does not exceed a certain constant, and therefore the complexity of the cycle can be estimated as $O(p)$.

Finally, it is important to note that steps $1, 2, 3$ can be performed as a stage of preliminary calculations and do not directly affect the process of generating pseudorandom bits. Therefore, their total contribution to the complexity of the algorithm can be estimated as $O(1)$.

Step 4.

At this step, we perform separate multiplication of register cell order numbers. The multiplication of cell order numbers is performed as an addition of the corresponding degrees of a primitive element. In contrast to multiplication, the complexity of the addition operation can be estimated as a constant $O(1)$. There are $(p-2)$ addition operations performed, that is, the complexity of this step is $O(p)$.

Step 5.

At this step, two division operations with the remainder by $(p-1)$ are performed. Each such division has the complexity about $O(log\ (p))$.

Steps 6, 7.

At these final steps of the algorithm, two preliminary calculations are accessed and two corresponding register bits are added by modulo 2. The total complexity of these operations is the constant $O(1)$.

Step 8.

The complexity of this step completely coincides with the complexity of calculating the feedback for an ordinary register with linear feedback. It was previously shown that the complexity of this stage is proportional to the register length, which gives us an estimate of $O(p)$.

Thus, the overall complexity of the new algorithm is summarized as

$$O(1) + O(log(p)) + O(p).$$

The contribution of the division operations with the remainder can be considered insignificant - such operation is performed exactly two times per tact (constant number of times) and can also be bounded by p on the asymptotics. That gives us the reason to neglect this term and get the final expression for the complexity of the algorithm:

$$Tact\_Complexity = O(p)$$

CHAPTER 2. IMPLEMENTATION OF THE GENERATOR

Initial verification of ideas

For the initial verification of the transformation idea, a simplified implementation of the shift register was performed, and the *C# programming language* was chosen for this purpose. In the framework of this implementation, the shift register is designed as a stand-alone class called *LFSR*. This class includes two main fields: shift register and feedback polynomial — as well as methods for generating pseudorandom sequences.

Here is a shortened code fragment illustrating the structure of the *LFSR* class:

```
class LFSR
    {
      // --------------------------------------------------------
      // FIELDS
      // --------------------------------------------------------
      byte[] register;

      byte[] polynom;

      // --------------------------------------------------------
      // INPUT FUNCTIONS AND CONSTRUCTORS
      // --------------------------------------------------------

      public LFSR()
      {…}
      public LFSR(string degs_of_GF2_polynom, string initial_state)
      {…}
      bool CheckData(string degs_of_GF2_polynom, string initial_state)
      {…}

      // --------------------------------------------------------
      // METHODS
      // --------------------------------------------------------

      byte feedbackFunction()
      {…}
      public byte Tact()
      {…}
      public void GenRandomBinaryFile(int num_bytes)
      {…}

    }
```

Such an implementation can be used for research purposes, but its performance as well as the total functionality of the class needs to be improved. Let's pay attention to several points that made up the first stage of program optimization.

The first stage of optimization

First of all, let's look at how the shift register field can be implemented in a different way. One of the obvious factors affecting the running time of the current implementation is the slow execution of the register shift operation. The reason is that the shift register is so far implemented only as an array, and the register shift is performed by sequential rewriting bits from a given cell to a neighboring one, which is closer to the output.

Let us turn to the possible ways how the register shift can be performed in one command. For example, a register can be implemented with a sufficiently long unsigned integer. For example, it is convenient to implement a *32-bit* register using the *unsigned int* or a *64-bit* register using the *unsigned long*. In this case, the shift can be made in one command using the ">>" directive.

```
unsigned int register = 12345;   // initializing the register
register >> = 1;     // example of right shift of the register
```

However, the registers of length *16, 32, 64, 128* and other powers of two are not suitable to equip them with multiplicative convolution transform. Also it seems very problematic to implement the shift register entire functionality with *unsigned int, unsigned long* and other similar types if the required length of the register is somewhere between the powers of two. In the practice of implementing cryptographic algorithms it is widely known that the fastest software implementations of shift registers are usually performed in *«Assembler»* language or in *C language* with a smart compiler. Let us turn our attention to the modern standard of the *C language* – in particular, we can find a class called *bitset*.

Bitset is a special container class that is designed to store bit values (the elements of this container can have values: *0 or 1*, *true or false*).

The *bitset* class is very similar to a regular array, but, unlike an array, only one bit is allocated for each element of a *bitset* type, that is, the memory space taken by bit masks is optimized as much as possible. This circumstance allows us to use eight times less memory than it is required by the smallest elementary data type in *C ++,* the *char* type.

Each element (each bit) can be accessed individually: for example, we have an object of type *bitset* named *mybitset*, we need to access the fourth bit. To do this, after the array name, in square brackets we indicate the bit index, as if we would like to access the array element – *mybitset* [3]. Sometimes this data type is used as logical variables; individual elements in this case are links to logical values. As an addition to overloading several operators and providing direct access to bits, the *bitset* object can be converted to an *unsigned integer* value and to a binary *string*.

Here is a short list of the *bitset* class methods.

Member Functions of the bitset Container Class from C ++ Standard Template Library	
constructor	The *bitset* class constructor
operators	Overloaded operators for performing bitwise logical operations and organizing input/output of *bitset* objects
Access to bits	
operator[]	Overloaded square brackets operator for direct access to bits
Bit operations	
set	The function allows to initialize all bits with one or change the value of a single bit
reset	Reset specified bits (reset bits to 0)
flip	Convert the current register state to reverse code. That is, replace ones with zeros, and zeros with ones
Bitmask operations	
to_ulong	Convert bit object to integer long unsigned value
to_string	Convert a bit sequence to a string of type string, which will contain the characters 0 and 1
count	Returns the number of unit bits of the *bitset* object
size	Return the size of the *bitset* object (number of bits)
test	Gets the value of the specified bit of the *bitset* object
any	Checks the bit value for the presence of single bits in the *bitset* object
none	Checks the bit value for zero bits in the *bitset* object

Table 1. Member Functions of the bitset Container Class from C ++ Standard Template Library

Let us make an attempt to use the *bitset* class and construct new software implementation of the shift register in *C/C++ languages*. Here is a code fragment illustrating the structure of the new implementation:

```
class LFSR_82_79_47_44
{
   // REGISTER
   bitset<REG_82> reg;

public:

   // INPUT FUNCTIONS AND CONSTRUCTORS
   LFSR_82_79_47_44()
   {…}
   LFSR_82_79_47_44(vector<int> positionsOfTRUEbits)
   {…}
   void setState(vector<int> positionsOfTRUEbits)
   {…}
   string getState()
   {…}

   // RANDOM BIT GENERATION
   bool TactAsUsualReg()
   {…}
};
```

The second stage of optimization

Compared to its previous version, the new implementation has two significant features.

1) it is written in *C*, uses the special *bitset* class and, therefore, allows working with bits more efficiently

2) there is no feedback polynomial among the fields of the class

Let us tell more about the paragraph *2)*. Here was made an attempt to implement not a wide class of shift registers in general, but only one specific register defined by the polynomial $P(x) = x^{82}+x^{79}+x^{47}+x^{44}+1$, primitive in the field $GF(2)$. The goal, which was pursued in this case, was a faster implementation of the register tact function. Now for the case of ordinary register it looks like this:

```
bool TactAsUsualReg ()
 {
bool out = reg [0];
bool feedback = ((reg >> 81) ^ (reg >> 78) ^ (reg >> 46) ^ (reg >> 43)) [0];
```

```
 reg >> = 1;
 reg [81] = feedback;
return out;
}
```

As you can see, when calculating the feedback, only bitwise operations are involved, which can be executed very quickly. It is impossible to write the tact function in this way if we don’t know the primitive polynomial in advance and have it only as an input during after the running of program, as was provided in an earlier version.

The third stage of optimization

The current implementation of the register tact function can be accelerated in almost two times, and the class structure itself can be again expanded to a more general situation, when the register is mostly determined directly during the running of program and only partially before it starts.

Consider an implementation in which a primitive feedback polynomial is a field of type *bitset* of class *LFSR*. Thus, the feedback polynomial is implemented in the same way as the register itself. In this case, the structure of the *LFSR* class takes the following form:

```
class LFSR
{
   // REGISTER
   bitset<REGISTER_LENGTH> reg;

   // FEEDBACK POLYNOM
   bitset<REGISTER_LENGTH> feedbackPolynom;

public:
   // ---------------- CONSTRUCTORS ----------------
   LFSR(vector<int> nonzeroDegsInDescOrder)
   {…}
   LFSR(vector<int> nonzeroDegsInDescOrder, vector<int> positionsOfTRUEbits)
   {…}

   // ---------------- SET and PRINT METHODS ----------------
   void setState(vector<int> positionsOfTRUEbits)
   {…}
   string getState()
   {…}

   // ---------------- RANDOM BITS GENERATION ----------------
   bool TactAsUsualReg()
   {…}
}
```

The object *feedbackPolynom* stores the coefficient of *(N+1)*-th degree in its *N*-th cell (the sequence of cell order numbers starts from zero). Let us examine the form that the register tact function will take now:

```
bool TactAsUsualReg ()
{

bool outbit = reg [reg.size () - 1];

bool feedbackBit = (reg & feedbackPolynom) .count () & 1;

reg << = 1;
reg [0] = feedbackBit;

return outbit;
}
```

A computational experiment showed that this implementation of the tact function works almost twice faster than the implementation obtained at the previous stage of optimization.

Note that this implementation does not limit the form of the primitive polynomial, providing the ability to use any primitive polynomial of fixed length as the linear feedback. The practical advantage of this circumstance is the possibility of using both dense and sparse primitive polynomials.

The fourth stage of optimization

We will carry out the process of vectorizing the code. To do this, we add some instructions for the compiler to a special file *CMakeLists.txt* (the implementation was performed within the framework of the *CLion* software environment). Finally the file’s content will be as follows:

```
cmake_minimum_required (VERSION 3.5)
project (MCT_RNG)

set (CMAKE_CXX_STANDARD 17)
set (CMAKE_CXX_STANDARD_REQUIRED ON)
set (CMAKE_CXX_EXTENSIONS OFF)

find_package (OpenMP)
if (OPENMP_FOUND)
 set (CMAKE_C_FLAGS "$ {CMAKE_C_FLAGS} $ {OpenMP_C_FLAGS}")
 set(CMAKE_CXX_FLAGS "$ {CMAKE_CXX_FLAGS} $ {OpenMP_CXX_FLAGS}")
 set(CMAKE_EXE_LINKER_FLAGS"$ {CMAKE_EXE_LINKER_FLAGS}
{OpenMP_EXE_LINKER_FLAGS}")
endif ()

if ("$ {CMAKE_CXX_COMPILER_ID}"STREQUAL"GNU")
 set (CMAKE_CXX_FLAGS"$ {CMAKE_CXX_FLAGS} -fopt-info-vec -fopt-info-vec-missed
-Ofast -march = native")
endif ()

add_executable (MCT_RNG main.cpp)
```

Now each cycle and, in general, each piece of code is checked by the compiler for the possibility to optimize it according to the principle *SIMD – Single Instruction on Multiple Data*. The code fragment may be also explicitly checked for vectorization, if the certain directive ***#pragma omp simd*** is used.

The fifth stage of optimization and the final version of the program

The last step in optimizing the program was to add code that implements the multiplicative convolution transform and extends the functionality of the *LFSR* class. Let us present a list of significant elements in the final version of the program.

Fields of shift register and primitive polynomial.
Implemented as objects of the *bitset* class, the register length is fixed, the form of a primitive polynomial can be arbitrary.

Array fields for storing the results of preliminary calculations.
Used when performing an optimized version of the multiplicative convolution transform.

Constructor 1.
It takes a primitive feedback polynomial as an input. The polynomial is given by a set of significant degrees arranged in descending order. The initial state of the register is initialized by default with all bits equal one.

Constructor 2.
It takes two arguments as an input: a primitive feedback polynomial given by a set of significant degrees, arranged in descending order, and also the initial state specified by the list of register cells that should contain one. Zero bit will be placed in all other cells of the register by default.

GET – SET Methods.

The *setState()* method is used to change the current state of the register. The *getState()* and *getPolynom()* methods are used to output the current state of the register and the feedback polynomial to the console.

Methods for generating pseudorandom numbers.

The functions *TactAsUsualReg()* and *TactWithMulConv()* allow to produce a pseudorandom bit, interacting with this register as an ordinary one with linear feedback or as a register equipped with a multiplicative convolution, respectively.

Similarly:

— the functions *NextByteAsUsualReg()* and *NextByteWithMulConv()* allow to get a pseudorandom byte

— the *ArrayOfRandomBytesAsUsualReg()* and *ArrayOfRandomBytesWithMulConv()* functions allow you to get an array of pseudorandom bytes of a given length

— the *NextUIntAsUsualReg()* and *NextUIntWithMulConv()* functions return an unsigned pseudorandom integer.

Methods for generating pseudo-random sequences written to a file.

The functions *ToTextFileAsUsualReg()* and *ToTextFileWithMulConv()* provide a way to get a pseudorandom sequence of bits of a given length and write it to a text file as a sequence of zeros and ones. As a result, the contents of the file will look something like this:

```
111000110111000111001000000111111000111111000111000110110111111110001001110001
110001110011100000011100000011111101110011100000000000011111100111000100100 11
101100010000111111000100011011100111000111000011011100101001001001110001001 11
000011100011110110110110101110001001000111000000000111000100100000000100100111
111111000000111011111111100011111111100001100111000110111000100101000110111000
110111010100110000101011110111101011101101001001100100100010011011011101010 01
001010011111100000101011011011010101011011001000111001000110101001100001101 10
11111011000100100011101001001110011001000000001100011100
```

The functions *ToBinaryFileAsUsualReg()* and *ToBinaryFileWithMulConv()* do the same task, but only write the output bits to a binary file.

The estimation of speed of the program

The diagram below shows the average bit generation rate depending on the length of the register that was used. To obtain these results the final version of the program was taken and each register was equipped with a multiplicative convolution transform.

When generating bits, the following primitive polynomials were used.

$$P_1(x) = x^{58} + x^{39} + 1$$
$$P_2(x) = x^{82} + x^{79} + x^{47} + x^{44} + 1$$
$$P_3(x) = x^{102} + x^{101} + x^{36} + x^{35} + 1$$
$$P_4(x) = x^{126} + x^{125} + x^{90} + x^{89} + 1$$
$$P_5(x) = x^{150} + x^{97} + 1$$
$$P_6(x) = x^{166} + x^{165} + x^{128} + x^{127} + 1$$

The initial state of all the registers before the start of the experiment consisted solely of ones.

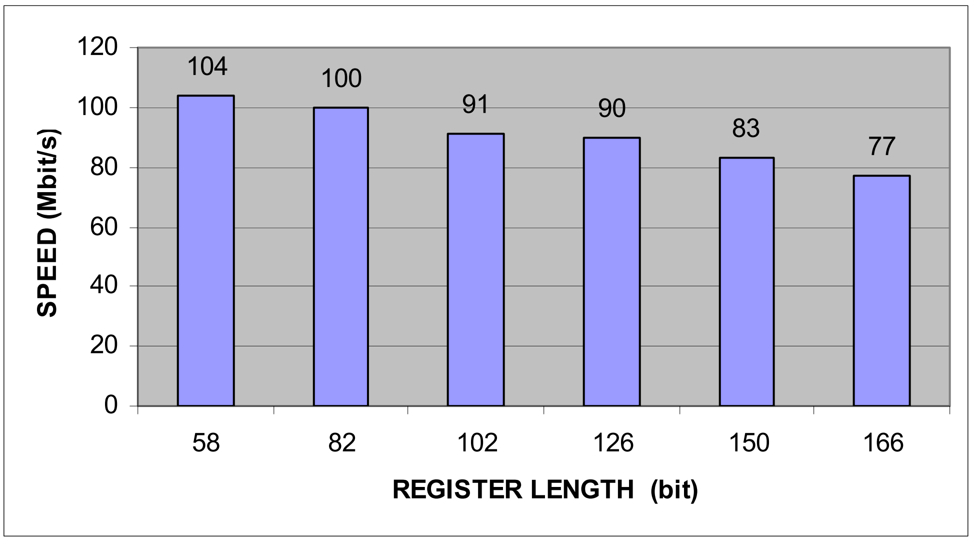


Fig. 17. The speed of producing pseudorandom bits

Conclusion:

The software implementation supports a fairly high pseudorandom bit generation rate. Particular attention can be paid to the *82-bit* register. Its length is large enough to resist the brute force attack, while the pseudorandom bit generation rate of this register is about *100* Megabits per second.

Let us compare the average speed of two registers: let them have the same length and same feedback polynomial, but also let us equip one of them with multiplicative convolution transform. The comparison is correct, because the registers differ only in the fragment of their code which implements the multiplicative convolution transform.

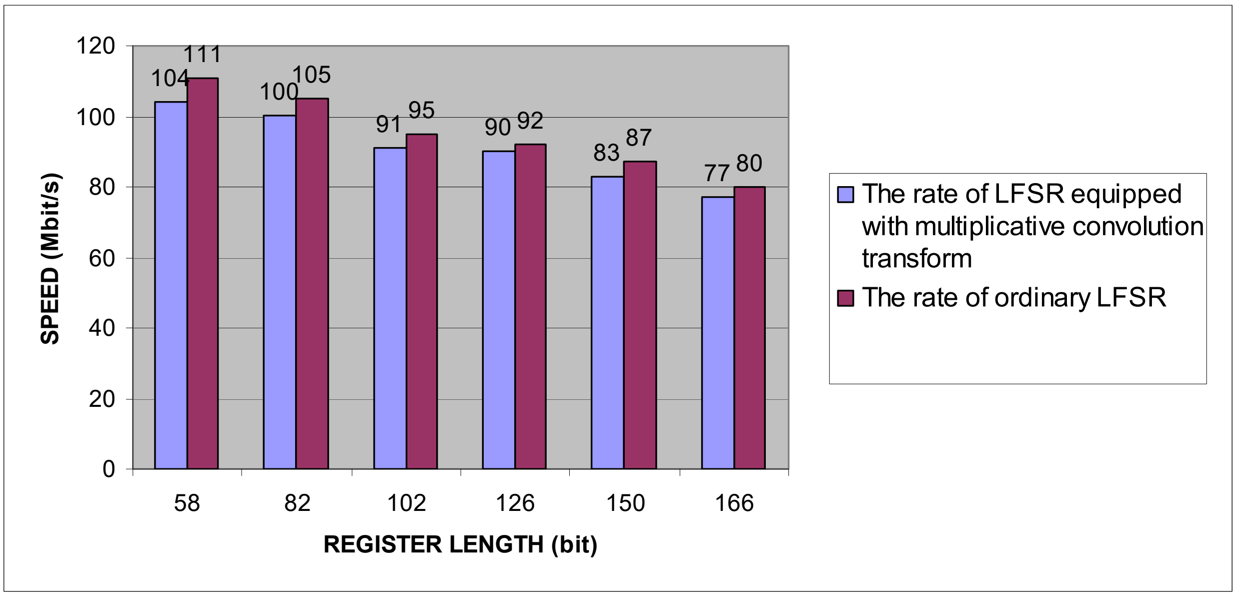


Fig. 18. Comparison of pseudorandom bits producing rate

Conclusion:

The pseudorandom bit generation rates are almost the same — the difference is only about *5%* regardless of the length of the register.

Testing the program code

To verify the written program code, a special system of program tests was constructed. This system includes *15 tests* for various components of the system (so-called *Unit*-Tests). Each test uses specific numerical parameters. These parameters are unique for each test and stored in special text files.

Since the components of the functionality of the *LFSR* class have some dependencies on each other, it is better to perform a full testing of all system components in accordance with the following scheme:

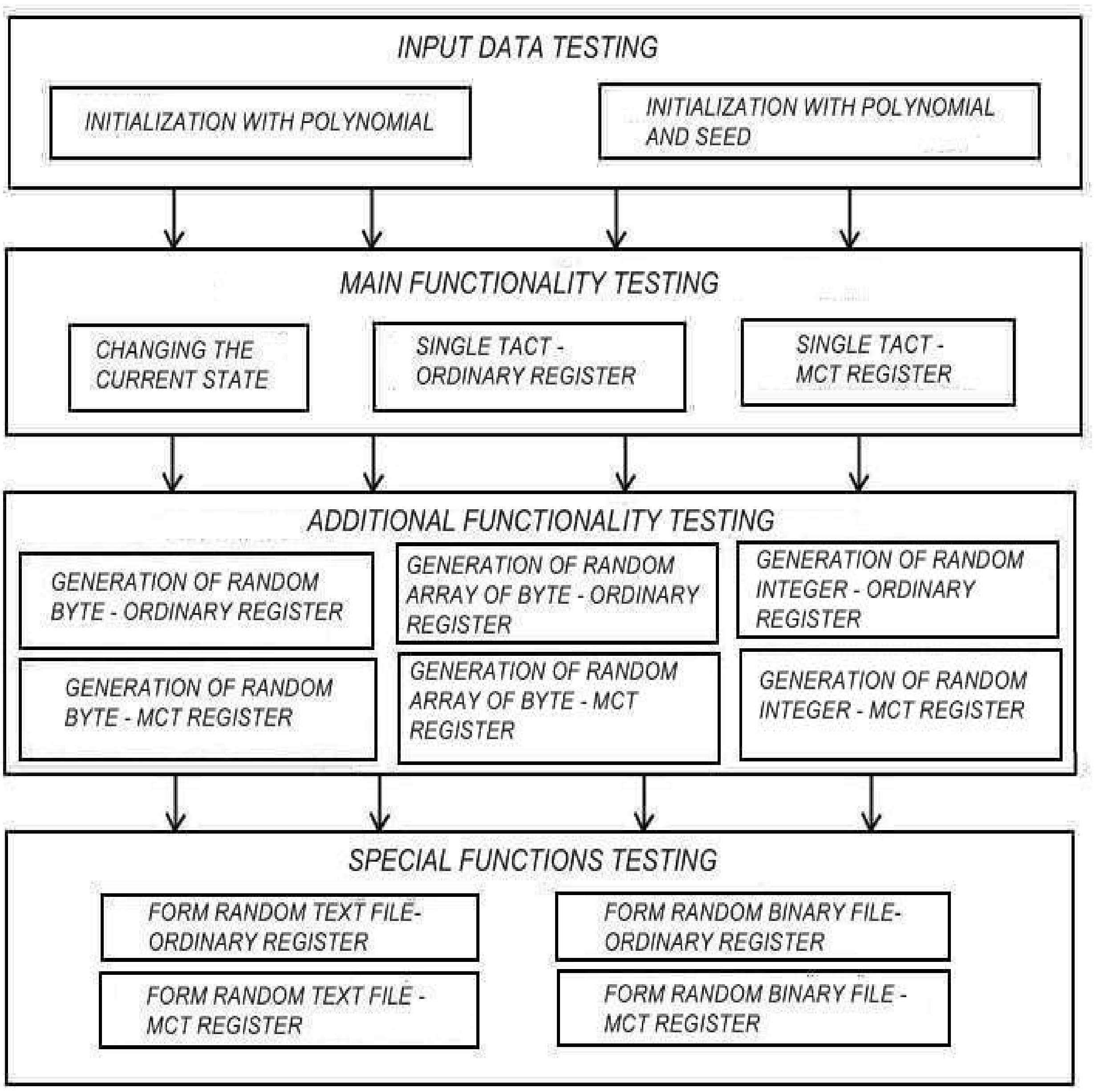

INPUT DATA TESTING
INITIALIZATION WITH POLYNOMIAL
INITIALIZATION WITH POLYNOMIAL AND SEED
MAIN FUNCTIONALITY TESTING
CHANGING THE CURRENT STATE
SINGLE TACT - ORDINARY REGISTER
SINGLE TACT - MCT REGISTER
ADDITIONAL FUNCTIONALITY TESTING
GENERATION OF RANDOM BYTE - ORDINARY REGISTER
GENERATION OF RANDOM ARRAY OF BYTE - ORDINARY REGISTER
GENERATION OF RANDOM INTEGER - ORDINARY REGISTER
GENERATION OF RANDOM BYTE - MCT REGISTER
GENERATION OF RANDOM ARRAY OF BYTE - MCT REGISTER
GENERATION OF RANDOM INTEGER - MCT REGISTER
SPECIAL FUNCTIONS TESTING
FORM RANDOM TEXT FILE- ORDINARY REGISTER
FORM RANDOM BINARY FILE- ORDINARY REGISTER
FORM RANDOM TEXT FILE - MCT REGISTER
FORM RANDOM BINARY FILE - MCT REGISTER

Fig. 19. Testing of program modules

Let's take a closer look at the test modules shown in the diagram.

Input Testing

These tests are designed to verify that part of the functionality of the *LFSR* class which is implemented in the constructors. At the very beginning of the program, the user needs to set such parameters as the primitive polynomial and, if desired, the initial state of the generator. By default, the register will be filled with ones. The created set of tests is designed to confirm the correct interpretation of the input data by the program when they are correct, as well as determine the behavior of the program on the specially chosen incorrect data. The corresponding tests are implemented by the functions *Constructor1_test()* and *Constructor2_test()*.

Testing the core functionality of a class

First of all, the user will need the method to reset the initial state of the generator or change it at an arbitrary moment of time and, of course, the user should have a method for generating pseudorandom bits. The created set of tests is designed to verify the program behavior while changing the current state of the generator and generating pseudorandom bits. Such initial data as a primitive polynomial are considered to be correct a priori, and the current state of the register and the set of output bits are checked. The corresponding tests are implemented by the functions *SetState_test(), TactAsUsualReg_test(), TactWithMulConv_test()*.

Testing Additional Functionality

Modern applications that use random number generators do not always need to generate a sequence of pseudorandom bits. Quite often, pseudorandom generators are required to generate a pseudorandom integer from a specified range, or there is a special need to generate an array of pseudorandom bytes. The *LFSR* class provides an opportunity to generate a single pseudorandom byte, an unsigned integer, and an array of pseudorandom bytes of a specified length. The created tests are designed to verify the program in all of these cases. Initial data such as the primitive polynomial and the initial state of the register are considered to be correct a priori, while the output sequence is checked. The Corresponding tests are implemented by the functions

NextByteAsUsualReg_test()

NextByteWithMulConv_test ()

ArrayOfRandomBytesAsUsualReg_test ()

ArrayOfRandomBytesWithMulConv_test ()

NextUIntAsUsualReg_test ()

NextUIntWithMulConv_test ()

Testing Special Features

To check the cryptographic properties of the generator, it is important to save the generated pseudorandom sequence in a specific file or write it to a file in a special form. Most modern pseudorandom sequence testing packages accept the data in two different forms: in the form of a binary file into which an array of pseudorandom bytes is written or in the form of a text file in which pseudorandom bits are converted to characters *0* and *1*. The created set of tests is designed to verify the process of generating pseudorandom bits and bytes followed by writing them to a file. Initial data such as the primitive polynomial and the initial state of the register are considered to be correct a priori, while the output sequence and file contents are checked. The corresponding tests are implemented in functions

ToTextFileAsUsualReg_test()
ToTextFileWithMulConv_test()
ToBinaryFileAsUsualReg_test()
ToBinaryFileWithMulConv_test().

After the end of testing all the components of the random number generator functionality, the program creates a brief report, an example of which is presented below

```
TEST OF THE FIRST CONSTRUCTOR...
Input file name: UTest_Constructor1.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF THE SECOND CONSTRUCTOR...
Input file name: UTest_Constructor2.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF 'set_state' METHOD...
Input file name: UTest_SetState.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF SINGLE RANDOM BIT GENERATION (USUAL REGISTER)...
Input file name: UTest_TactAsUsualReg.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF SINGLE RANDOM BIT GENERATION (MUL_CONV)...
Input file name: UTest_TactWithMulConv.txt
Output file name: OutProbe.txt
```

```
Result: CORRECT

TEST OF SINGLE RANDOM BYTE GENERATION (USUAL REGISTER)...
Input file name: UTest_NextByteAsUsualReg.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF SINGLE RANDOM BYTE GENERATION (MUL_CONV)...
Input file name: UTest_NextByteWithMulConv.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF RANDOM BYTES SEQUENCE (USUAL REGISTER)...
Input file name: UTest_ArrayOfRandomBytesAsUsualReg.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF RANDOM BYTES SEQUENCE (MUL_CONV)...
Input file name: UTest_ArrayOfRandomBytesWithMulConv.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF UNSIGNED INT VALUE GENERATION (USUAL REGISTER)...
Input file name: UTest_NextUIntAsUsualReg.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF UNSIGNED INT VALUE GENERATION (MUL_CONV)...
Input file name: UTest_NextUIntWithMulConv.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF RANDOM SEQUENCE WRITTEN TO FILE (USUAL REGISTER)...
Input file name: UTest_ToTextFileAsUsualReg.txt
Output file name: OutProbe.txt
```

```
Result: CORRECT

TEST OF RANDOM SEQUENCE WRITTEN TO FILE (MUL_CONV)...
Input file name: UTest_ToTextFileWithMulConv.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF RANDOM BYTES WRITTEN TO BINARY FILE (USUAL REGISTER)...
Input file name: UTest_ToBinaryFileAsUsualReg.txt
Output file name: OutProbe.txt
Result: CORRECT

TEST OF RANDOM BYTES WRITTEN TO BINARY FILE (MUL_CONV)...
Input file name: UTest_ToBinaryFileWithMulConv.txt
Output file name: OutProbe.txt
Result: CORRECT
```

CHAPTER 3. CRYPTOGRAPHIC RESISTANCE OF SHIFT REGISTER EQUIPPED WITH MULTIPLICATIVE CONVOLUTION TRANSFORM

Graphic tests for statistical evaluation of pseudorandom sequence quality

In all the tests presented below there was used a register with a length of $L=82$, equipped with multiplicative convolution transform. Feedback was given by the polynomial $P(x) = x^{82}+x^{79}+x^{47}+x^{44}+1$, primitive in the field $GF(2)$. The initial state of the register consisted solely of ones.

1) Histogram of elements distribution

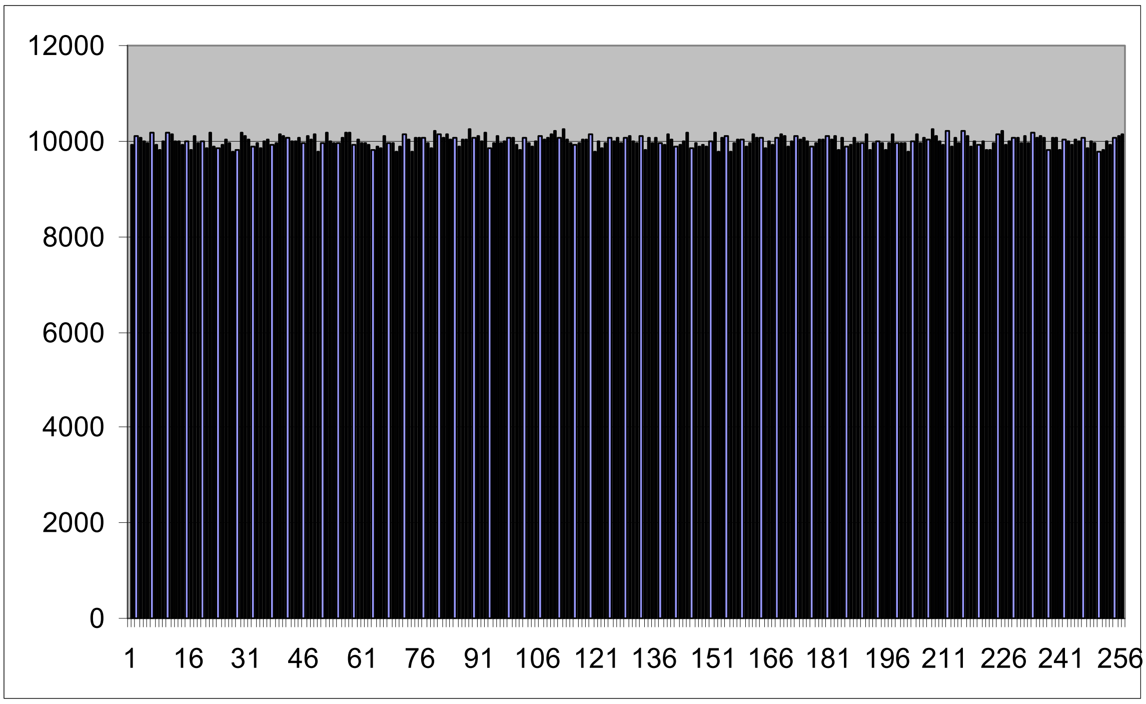
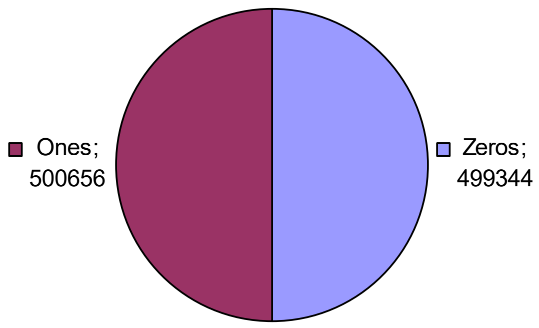
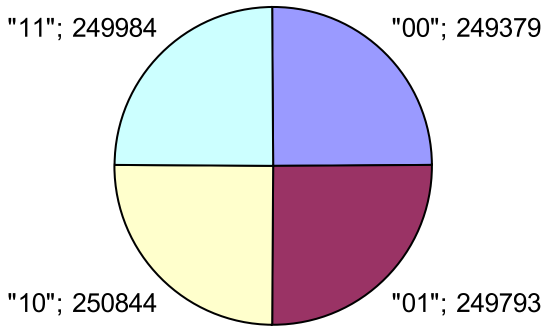
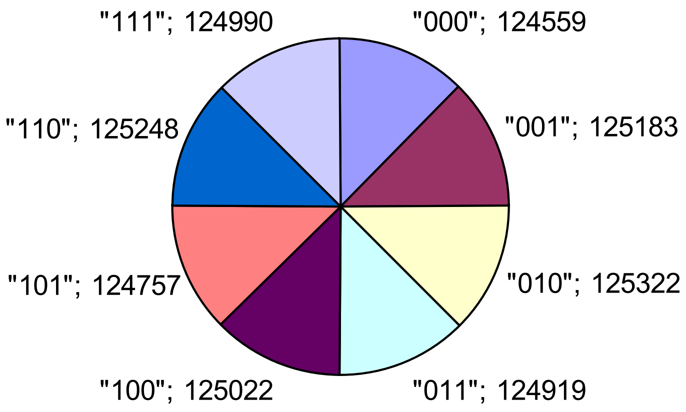


Fig. 20. The distribution of pseudorandom numbers

Using the random number generator under study we obtained 25600000 pseudorandom natural numbers, each in the range from 1 to 256. In accordance with the data presented in the diagram, there are all numbers from 1 to 256 without any exception, and the spread in the frequencies of their appearance is very small. Test passed.

2) Series graphical test

Let us generate a sequence of one million pseudorandom bits and check the uniformity of the distribution of characters in the sequence. To do this, we analyze the frequency of bit series of length equal to 1, 2, and 3 without overlapping.

	Fig. 21. Monobit series distribution
	Fig. 22. Two-bit distribution
	Fig. 23. Three-bit series distribution

As it is presented in the diagram, the difference in the frequency of occurrence between various series of the same length is extremely small, which indicates the uniform distribution of characters. Test passed.

***3)* Autocorrelation function**

Let us generate a sequence of one million bits and then make a replacement in accordance with the following rule:

$$1 \to 1$$

$$0 \to -1$$

Next, we find the correlation peaks c_j in accordance with the formula

$$c_j = \frac{\sum_{i=0}^{n-1} b_i \cdot b_{(i+j) \bmod n}}{\sum_{i=0}^{n-1} b_i^2}, \quad j = 0, 1, \ldots, n-1$$

Let us present the results in the graph.

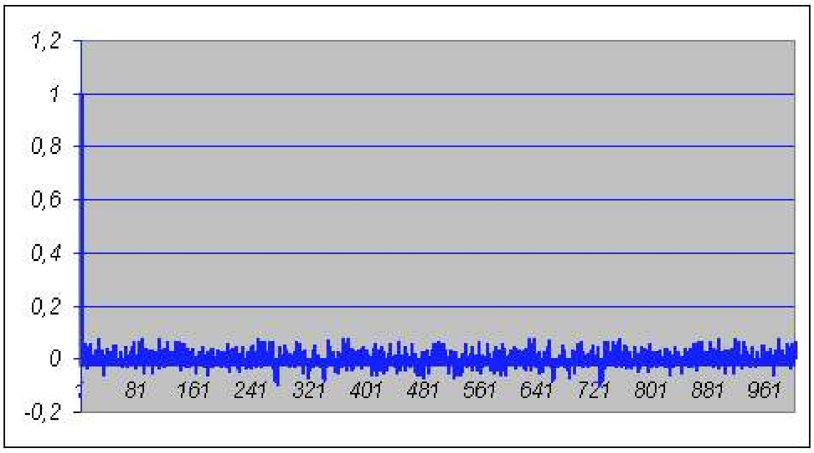


Fig. 24. Autocorrelation function

As it is presented in the diagram, all the correlation peaks are very small, excluding the very first point, where there is a single high peak. This fact indicates the random nature of the elements of the sequence. Test passed.

Remark

The first and second graphical tests were carried out to demonstrate the basic statistical properties of the developed generator. The third test with the construction of an autocorrelation function indicates not only good statistical properties, but also cryptographic stability of the generator: correlation bursts at any point except the very first one would indicate a linear relationship between the bits of the output sequence, as well as a low linear complexity of the entire output sequence. The encryption using the gamma which is created by such a generator is extremely unreliable, because due to the low linear complexity of the output sequence, the output of the generator can be simulated by an ordinary shift register of a fixed length, which means an adversary can reproduce the gamma and decrypt the message. In addition, the linear dependency between the bits of the output sequence can lead to the determination of the initial state of the generator, which again leads to cracking the cipher.

However, according to the results of this test, a correlation burst is observed only at the very first point, which suggests that there is no linear relationship between the bits of the output sequence and the linear complexity of the entire sequence is sufficiently high.

Testing the generator using NIST Statistical Test Suite

This set of statistical tests was developed as the result of the work by the National Institute of Standards and Technology (in the *USA*) to create quality standards for pseudo-random sequences [19]. *NIST* tests are efficiently implemented in packages running under *LINUX* and *UNIX* operating systems and have open source code in C. Successful passing of all *NIST* tests without exception is among the minimum set of requirements for random number generators. The list of tests is as follows:

- Frequency Test (Monobit)

The number of zeros and ones in a truly random sequence of bits is approximately the same.

- Frequency Test within a Block

In a truly random bit sequence, the repetition rate of ones and zeros in a block of a particular length is approximately the same.

- A test for a sequence of identical bits

The number of blocks consisting of one, two, three or more units are compared with the number of such blocks in a truly random binary sequence.

- Longest Run of Ones in a Block

In this test the length of the longest series of ones within a block of a certain length is determined. Then it is compared to the expected value for a truly random binary sequence.

- Binary Matrix Rank Test

The sub-rows of the original binary sequence are considered and the measure of the linear dependence of the selected sub-rows is determined.

- Discrete Fourier Transform (Spectral) Test

A discrete Fourier transform is applied to the original binary sequence. The test will fail if there are periodic repetitions in the sequence.

- Non-Overlapping Template Matching Test

Before testing the source sequence, a pattern which is a binary sequence of a certain length is selected. The frequency with which the selected pattern is encountered in the original sequence must satisfy a known distribution.

- Overlapping Template Matching Test

Before testing the source sequence, a pattern which is a binary sequence of a certain length is selected. The frequency with which the selected pattern is encountered in the original sequence must satisfy a known distribution. The difference between this test and the previous test is that the pattern search is performed in one-bit increments, regardless of whether the pattern was fixed at the previous step.

- Maurer's Universal Statistical test

A truly random sequence cannot be significantly compressed without losing information.

- Linear Complexity Test

A truly random sequence cannot be constructed using a linear feedback shift register.

- Serial test

During this test a frequency of each of the prepared templates in a given sequence is calculated. In the case of a truly random sequence, the frequencies of appearance of each of the patterns are approximately the same.

- Approximate Entropy Test

During this test a frequency of the smallest of the two consecutive blocks of an original sequence is included into the larger block is calculated. The block lengths differ by one. The number of occurrences of a smaller block in a larger one must correspond to a known distribution.

- Cumulative Sums (Forward) Test

The sums of elements of the subsequences of a given sequence that have the same length as the original sequence must satisfy a known distribution.

- Random Excursions Test

The test is a set of eight separate studies related to the calculation of the sum of the elements from the subsequences of a given sequence. The decision on the degree of randomness of the original sequence is taken for each study separately.

- Random Excursions Variant Test

The test is a variation of the previous test and is a set of eighteen separate studies related to the calculation of the sums of the elements from the subsequences of a given sequence. The decision on the degree of randomness of the original sequence is taken for each study separately.

Let us generate *100* sequences each of the length equal to *1 000 000* bits. We will perform a study using the *NIST* statistical test suite.

The results are presented in the table. The percentage of sequences that passed one or another test is displayed in the *Proportion* column.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
9	7	12	8	12	13	6	13	13	7	0.595549	98/100	Frequency
44	19	7	10	7	5	4	2	1	1	0.000001	90/100	BlockFrequency
8	9	6	14	10	10	12	9	13	9	0.816537	98/100	CumulativeSums
10	11	10	9	6	11	8	7	10	18	0.383827	98/100	CumulativeSums
18	6	8	7	7	9	17	7	8	13	0.042808	98/100	Runs
12	7	11	12	14	12	8	5	9	10	0.657933	97/100	LongestRun
7	11	9	8	13	5	16	11	8	12	0.401199	100/100	Rank
13	8	9	10	7	7	11	8	14	13	0.719747	98/100	FFT
11	9	18	8	6	7	15	4	13	9	0.055361	99/100	NonOverlappingTemplate
11	7	11	12	11	12	11	11	10	4	0.759756	100/100	NonOverlappingTemplate
11	7	14	8	7	11	12	4	17	9	0.162606	99/100	NonOverlappingTemplate
11	7	14	6	12	16	4	11	8	11	0.191687	99/100	NonOverlappingTemplate
8	8	13	15	17	8	4	9	6	12	0.085587	98/100	NonOverlappingTemplate
7	16	12	6	10	10	12	5	12	10	0.366918	100/100	NonOverlappingTemplate
8	10	9	6	8	13	15	4	12	15	0.191687	100/100	NonOverlappingTemplate
11	9	12	8	3	14	13	10	12	8	0.419021	98/100	NonOverlappingTemplate
7	7	4	9	13	12	9	10	15	14	0.275709	100/100	NonOverlappingTemplate
10	12	12	16	10	9	13	6	7	5	0.319084	99/100	NonOverlappingTemplate
9	5	10	11	9	7	15	10	9	15	0.455937	99/100	NonOverlappingTemplate
5	13	8	7	14	12	11	14	6	10	0.350485	99/100	NonOverlappingTemplate
9	7	7	10	12	11	11	13	6	14	0.678686	98/100	NonOverlappingTemplate
9	12	13	9	11	9	8	9	10	10	0.987896	98/100	NonOverlappingTemplate
11	8	15	8	8	13	10	9	9	9	0.834308	100/100	NonOverlappingTemplate
7	9	14	13	12	14	9	5	11	6	0.366918	100/100	NonOverlappingTemplate
1	6	12	12	13	14	11	11	9	11	0.145326	100/100	NonOverlappingTemplate
9	12	12	12	9	8	13	8	6	11	0.851383	98/100	NonOverlappingTemplate
8	16	13	13	8	7	12	6	8	9	0.383827	99/100	NonOverlappingTemplate
7	13	10	9	10	13	11	8	7	12	0.867692	100/100	NonOverlappingTemplate

9	10	14	10	11	14	9	9	9	5	0.719747	100/100	NonOverlappingTemplate
7	8	6	18	7	9	10	7	14	14	0.108791	100/100	NonOverlappingTemplate
8	17	6	9	5	11	13	8	10	13	0.224821	100/100	NonOverlappingTemplate
7	11	13	12	13	9	11	13	7	4	0.455937	100/100	NonOverlappingTemplate
8	11	5	14	11	7	10	11	12	11	0.719747	98/100	NonOverlappingTemplate
8	11	10	13	4	12	12	9	10	11	0.739918	99/100	NonOverlappingTemplate
14	7	7	12	7	13	8	8	6	18	0.108791	98/100	NonOverlappingTemplate
8	13	6	11	16	8	11	5	14	8	0.236810	98/100	NonOverlappingTemplate
5	12	7	6	10	8	9	14	12	17	0.171867	100/100	NonOverlappingTemplate
26	11	6	13	10	9	5	6	9	5	0.000060	99/100	NonOverlappingTemplate
8	12	16	8	6	12	10	6	15	7	0.224821	100/100	NonOverlappingTemplate
10	11	4	8	6	9	13	14	14	11	0.350485	97/100	NonOverlappingTemplate
11	12	16	9	10	11	2	8	11	10	0.262249	100/100	NonOverlappingTemplate
9	14	11	16	7	4	5	16	9	9	0.062821	99/100	NonOverlappingTemplate
11	5	15	11	8	7	8	10	10	15	0.401199	98/100	NonOverlappingTemplate
5	11	10	9	15	13	10	8	8	11	0.637119	100/100	NonOverlappingTemplate
6	8	12	8	16	14	11	9	7	9	0.419021	100/100	NonOverlappingTemplate
8	14	11	10	8	10	12	13	7	7	0.779188	100/100	NonOverlappingTemplate
9	12	14	16	10	3	9	11	7	9	0.224821	98/100	NonOverlappingTemplate
10	7	7	13	12	13	8	11	10	9	0.867692	100/100	NonOverlappingTemplate
11	13	9	6	12	11	8	14	9	7	0.719747	100/100	NonOverlappingTemplate
7	6	9	16	11	12	7	9	11	12	0.514124	100/100	NonOverlappingTemplate
13	6	10	13	10	14	5	9	9	11	0.554420	98/100	NonOverlappingTemplate
5	11	13	17	8	8	5	9	10	14	0.145326	99/100	NonOverlappingTemplate
7	7	7	7	13	14	11	15	8	11	0.419021	100/100	NonOverlappingTemplate
13	7	17	9	4	14	8	10	9	9	0.181557	98/100	NonOverlappingTemplate
7	13	6	8	14	9	9	12	8	14	0.534146	100/100	NonOverlappingTemplate
12	6	6	14	10	15	4	12	11	10	0.224821	97/100	NonOverlappingTemplate
9	11	9	9	12	9	11	11	12	7	0.983453	97/100	NonOverlappingTemplate
7	12	10	8	11	9	10	10	12	11	0.983453	100/100	NonOverlappingTemplate
15	16	15	7	8	13	4	10	7	5	0.037566	99/100	NonOverlappingTemplate
10	11	12	15	9	12	13	5	9	4	0.304126	99/100	NonOverlappingTemplate
11	6	11	7	13	7	10	14	9	12	0.678686	98/100	NonOverlappingTemplate
7	13	6	5	11	20	9	7	15	7	0.015598	99/100	NonOverlappingTemplate
7	6	6	11	13	12	15	10	10	10	0.534146	99/100	NonOverlappingTemplate
5	11	11	13	9	9	12	12	8	10	0.834308	99/100	NonOverlappingTemplate
6	13	8	10	13	12	8	11	11	8	0.816537	99/100	NonOverlappingTemplate

10	6	6	10	9	17	9	15	12	6	0.171867	99/100	NonOverlappingTemplate
8	11	7	15	9	6	14	7	12	11	0.474986	98/100	NonOverlappingTemplate
13	16	14	14	6	6	6	7	13	5	0.051942	99/100	NonOverlappingTemplate
14	14	4	11	10	9	7	10	10	11	0.534146	100/100	NonOverlappingTemplate
9	12	9	9	5	13	10	7	12	14	0.637119	99/100	NonOverlappingTemplate
14	14	8	9	11	15	3	7	10	9	0.202268	100/100	NonOverlappingTemplate
13	7	8	14	9	10	8	8	12	11	0.816537	99/100	NonOverlappingTemplate
11	15	9	11	13	9	11	5	9	7	0.595549	97/100	NonOverlappingTemplate
10	9	14	9	13	9	10	9	6	11	0.867692	100/100	NonOverlappingTemplate
9	6	15	10	14	5	7	12	13	9	0.304126	98/100	NonOverlappingTemplate
8	9	10	14	13	8	8	3	15	12	0.236810	99/100	NonOverlappingTemplate
5	12	8	12	17	8	12	5	8	13	0.153763	98/100	NonOverlappingTemplate
11	18	11	7	15	8	9	9	4	8	0.102526	99/100	NonOverlappingTemplate
13	3	17	3	8	12	10	8	11	15	0.021999	100/100	NonOverlappingTemplate
10	9	7	11	10	11	10	9	12	11	0.994250	97/100	NonOverlappingTemplate
11	14	11	10	7	8	13	10	13	3	0.366918	100/100	NonOverlappingTemplate
9	9	15	6	6	9	10	16	13	7	0.249284	98/100	NonOverlappingTemplate
11	9	18	8	6	7	15	4	13	9	0.055361	99/100	NonOverlappingTemplate
8	7	16	10	8	12	11	6	9	13	0.494392	100/100	NonOverlappingTemplate
17	13	7	8	2	12	6	15	6	14	0.011791	100/100	NonOverlappingTemplate
13	17	9	10	5	5	5	15	11	10	0.066882	98/100	NonOverlappingTemplate
12	8	13	11	6	13	10	7	9	11	0.798139	98/100	NonOverlappingTemplate
8	7	11	9	12	6	10	11	12	14	0.779188	100/100	NonOverlappingTemplate
11	11	6	9	15	7	8	11	8	14	0.554420	98/100	NonOverlappingTemplate
7	11	11	11	11	8	8	15	6	12	0.678686	99/100	NonOverlappingTemplate
7	7	7	11	12	14	10	11	11	10	0.834308	100/100	NonOverlappingTemplate
11	10	11	5	8	5	5	13	17	15	0.058984	98/100	NonOverlappingTemplate
14	10	11	8	12	13	3	8	14	7	0.262249	98/100	NonOverlappingTemplate
6	12	8	11	9	15	7	6	15	11	0.334538	100/100	NonOverlappingTemplate
7	10	7	11	7	11	13	9	12	13	0.816537	100/100	NonOverlappingTemplate
10	6	12	8	3	8	8	14	14	17	0.062821	99/100	NonOverlappingTemplate
9	7	8	15	14	7	9	13	11	7	0.494392	99/100	NonOverlappingTemplate
11	11	8	7	5	11	12	13	9	13	0.699313	98/100	NonOverlappingTemplate
10	14	11	7	8	6	10	10	11	13	0.779188	100/100	NonOverlappingTemplate
11	14	5	6	9	10	6	17	14	8	0.108791	98/100	NonOverlappingTemplate
5	11	12	10	12	8	12	4	16	10	0.249284	100/100	NonOverlappingTemplate
16	5	6	8	10	19	16	5	7	8	0.004981	98/100	NonOverlappingTemplate

13	5	17	14	10	4	3	13	10	11	0.021999	98/100	NonOverlappingTemplate
10	10	4	11	11	13	13	6	10	12	0.574903	100/100	NonOverlappingTemplate
13	13	7	6	16	10	13	10	4	8	0.171867	98/100	NonOverlappingTemplate
7	16	10	4	12	10	16	8	10	7	0.145326	100/100	NonOverlappingTemplate
7	11	10	7	17	8	12	11	8	9	0.514124	100/100	NonOverlappingTemplate
9	10	9	15	14	11	7	8	12	5	0.474986	99/100	NonOverlappingTemplate
10	13	10	14	8	12	4	5	14	10	0.275709	99/100	NonOverlappingTemplate
8	12	12	13	11	9	13	8	6	8	0.779188	99/100	NonOverlappingTemplate
8	13	11	9	13	7	9	8	12	10	0.897763	100/100	NonOverlappingTemplate
6	16	9	10	12	9	12	9	5	12	0.419021	100/100	NonOverlappingTemplate
14	19	10	8	9	6	11	10	10	3	0.051942	98/100	NonOverlappingTemplate
7	13	13	14	10	12	12	9	2	8	0.213309	100/100	NonOverlappingTemplate
12	10	10	9	9	9	10	10	11	10	0.999777	99/100	NonOverlappingTemplate
6	11	9	14	6	14	15	9	9	7	0.334538	100/100	NonOverlappingTemplate
6	13	11	9	6	14	7	14	11	9	0.474986	99/100	NonOverlappingTemplate
11	9	10	7	9	8	7	12	10	17	0.554420	97/100	NonOverlappingTemplate
7	13	11	7	8	14	16	8	8	8	0.383827	98/100	NonOverlappingTemplate
10	9	12	7	14	10	11	14	9	4	0.494392	99/100	NonOverlappingTemplate
10	6	11	9	11	8	14	6	11	14	0.616305	98/100	NonOverlappingTemplate
13	11	10	6	14	10	8	9	9	10	0.851383	100/100	NonOverlappingTemplate
11	10	9	10	13	7	11	10	13	6	0.867692	99/100	NonOverlappingTemplate
7	10	13	6	12	8	11	6	15	12	0.455937	100/100	NonOverlappingTemplate
12	14	11	8	7	16	6	8	9	9	0.419021	99/100	NonOverlappingTemplate
9	8	8	14	10	9	15	8	9	10	0.779188	100/100	NonOverlappingTemplate
11	13	5	15	7	9	11	10	9	10	0.616305	99/100	NonOverlappingTemplate
9	14	12	7	15	9	8	9	5	12	0.437274	100/100	NonOverlappingTemplate
7	2	14	13	9	13	17	11	3	11	0.013569	99/100	NonOverlappingTemplate
11	6	17	8	10	9	6	13	10	10	0.383827	100/100	NonOverlappingTemplate
13	10	11	11	12	11	9	5	4	14	0.401199	98/100	NonOverlappingTemplate
9	12	12	12	15	12	9	5	7	7	0.474986	98/100	NonOverlappingTemplate
12	8	9	13	11	9	10	11	8	9	0.978072	99/100	NonOverlappingTemplate
9	14	11	4	14	11	7	10	10	10	0.534146	97/100	NonOverlappingTemplate
8	9	13	10	12	7	12	12	7	10	0.883171	99/100	NonOverlappingTemplate
8	10	11	16	10	6	11	9	15	4	0.213309	99/100	NonOverlappingTemplate
10	15	11	10	8	15	7	11	8	5	0.401199	99/100	NonOverlappingTemplate
8	10	9	14	11	4	8	10	9	17	0.262249	100/100	NonOverlappingTemplate
12	8	9	8	12	12	12	7	4	16	0.304126	99/100	NonOverlappingTemplate

7	8	14	11	13	10	11	11	7	8	0.798139	99/100	NonOverlappingTemplate
9	8	9	9	15	9	14	9	10	8	0.798139	100/100	NonOverlappingTemplate
9	8	7	13	18	8	10	12	9	6	0.262249	98/100	NonOverlappingTemplate
10	11	6	9	9	12	7	12	6	18	0.236810	99/100	NonOverlappingTemplate
13	4	6	9	9	17	9	12	11	10	0.224821	98/100	NonOverlappingTemplate
9	8	11	13	5	10	8	11	12	13	0.759756	99/100	NonOverlappingTemplate
14	10	7	8	16	11	11	9	7	7	0.474986	99/100	NonOverlappingTemplate
9	6	8	16	11	11	8	10	11	10	0.699313	99/100	NonOverlappingTemplate
12	9	8	11	13	6	13	12	9	7	0.759756	99/100	NonOverlappingTemplate
17	8	5	8	11	18	7	11	9	6	0.042808	100/100	NonOverlappingTemplate
7	15	6	10	9	12	8	7	13	13	0.474986	99/100	NonOverlappingTemplate
9	6	16	11	13	11	10	12	5	7	0.334538	99/100	NonOverlappingTemplate
13	7	8	13	9	9	11	11	8	11	0.911413	98/100	NonOverlappingTemplate
7	12	2	13	14	13	8	10	10	11	0.236810	99/100	NonOverlappingTemplate
8	9	7	13	15	7	14	13	7	7	0.350485	97/100	NonOverlappingTemplate
6	12	11	8	13	15	14	5	7	9	0.275709	99/100	NonOverlappingTemplate
9	9	15	6	6	9	10	16	13	7	0.249284	98/100	NonOverlappingTemplate
8	12	8	11	12	15	11	10	8	5	0.616305	98/100	OverlappingTemplate
8	16	9	12	5	8	10	10	11	11	0.574903	99/100	Universal
11	6	16	15	16	9	9	4	5	9	0.037566	99/100	ApproximateEntropy
5	9	8	3	4	3	8	7	11	8	0.232760	66/66	RandomExcursions
2	9	4	9	9	5	8	2	9	9	0.110952	65/66	RandomExcursions
6	4	9	8	7	7	3	7	7	8	0.772760	66/66	RandomExcursions
5	5	4	7	8	6	10	4	7	10	0.534146	66/66	RandomExcursions
8	6	7	8	3	1	8	8	6	11	0.178278	66/66	RandomExcursions
7	8	7	6	6	5	3	11	8	5	0.568055	64/66	RandomExcursions
4	3	5	9	6	10	10	7	5	7	0.378138	66/66	RandomExcursions
8	4	9	8	8	8	5	9	4	3	0.468595	63/66	RandomExcursions
5	5	9	9	6	4	5	6	9	8	0.706149	66/66	RandomExcursionsVariant
5	9	6	9	3	4	8	9	5	8	0.500934	64/66	RandomExcursionsVariant
6	7	3	8	6	7	6	10	5	8	0.739918	65/66	RandomExcursionsVariant
7	5	5	6	7	5	7	6	10	8	0.888137	65/66	RandomExcursionsVariant
7	6	5	3	8	5	7	9	8	8	0.772760	65/66	RandomExcursionsVariant
7	4	7	8	4	4	7	14	6	5	0.122325	64/66	RandomExcursionsVariant
9	1	7	5	11	4	5	13	6	5	0.022503	66/66	RandomExcursionsVariant
5	8	5	4	8	4	6	11	9	6	0.468595	66/66	RandomExcursionsVariant
2	9	2	10	8	5	5	8	5	12	0.043745	66/66	RandomExcursionsVariant

6	7	5	8	2	10	6	8	7	7	0.602458	65/66	RandomExcursionsVariant
4	8	7	5	7	4	5	13	5	8	0.232760	65/66	RandomExcursionsVariant
5	5	6	6	6	8	9	6	8	7	0.949602	65/66	RandomExcursionsVariant
4	5	3	9	8	8	10	4	8	7	0.407091	65/66	RandomExcursionsVariant
5	2	5	5	12	6	5	8	11	7	0.110952	65/66	RandomExcursionsVariant
6	5	6	3	7	4	5	9	7	14	0.090936	66/66	RandomExcursionsVariant
6	7	5	2	6	2	7	9	11	11	0.074177	66/66	RandomExcursionsVariant
6	4	5	7	3	5	8	12	7	9	0.275709	65/66	RandomExcursionsVariant
5	5	5	6	4	7	10	8	11	5	0.437274	65/66	RandomExcursionsVariant
16	10	9	10	10	9	9	8	9	10	0.883171	100/100	Serial
13	9	10	12	11	9	3	14	9	10	0.514124	99/100	Serial
10	13	7	10	11	7	9	13	13	7	0.779188	97/100	LinearComplexity

Table 2. Results for the uniformity of P-values and the proportion of passing sequences

Conclusion:

The random number generator passes all *NIST* statistical tests without any exception.

Testing the generator using DIEHARDER Statistical Test Suite

A random number generator, which passes all *DIEHARD* tests without any exception, deserves a very high praise. The set of *DIEHARD* tests is considered to be one of the strongest statistical criteria for matching the sequence with a truly random sample. The *DIEHARDER* package includes all of the original *DIEHARD* tests and also some other efficient statistical tests, which were either taken from different well-known statistical packages or invented by the *DIEHARDER* author Robert G. Brown. There is a description of the most important tests in this package given below.

Birthdays spacings Test

If several points are randomly selected on a sufficiently long interval, then under the hypothesis that the points are randomly chosen, the distribution of distances between these points asymptotically approaches the Poisson distribution.

Overlapping Permutations Test

If there are considered some numbers in the sequence under study, then subject to the condition of the random nature of the sequence, all possible permutations of this set of numbers should occur approximately the same number of times.

Ranks of binary matrices Test

It represents a calculation of characteristics associated with the ranks of matrices, which are composed of a certain quantity of bits from the sequence under study.

Monkey theorem Test

Several numbers are selected from the sequence under study, and the sequences of a certain number of binary digits, forming these numbers, are perceived as words. The number of words overlapping with the words of the entire flow of the sequence under study must satisfy a known distribution.

Count the 1's Test

A certain quantity of numbers of the studied sequence is selected; the quantity of single binary digits within them is calculated. The results are converted into “letters”, and the frequency of one, two, three, four and five letters is determined. These values must satisfy a known distribution.

Parking Lot Test

Circles of a unit radius are placed in a square of a certain size in accordance with the sequence under investigation. If a given sequence is truly random, some of the circles will be placed in a square without hitting previously placed circles, while others may overlap, and in this case, an attempt to place the circle must be repeated. According to the results of numerous series of experiments, the number of circles placed in a square should satisfy the normal distribution.

Minimum Distance Test

Several thousands of points are selected in accordance with a given sequence; the points are located in a square of a certain size. Next, the distance between each pair of selected points is calculated. The values of the squares of distances should have an exponential distribution.

Random Spheres Test

Several thousands of points are selected in a cube with a certain edge length and in accordance with a given sequence. Each point can form a sphere, which radius coincides with the distance to the nearest of the remaining points. The obtained values of the spheres volume should have an exponential distribution.

The Squeeze Test

In accordance with a given sequence there is a construction of real number sequence performed within the interval from zero to one. The elements of the constructed sequence are multiplied by a certain number while the result does not become equal to one. The quantity of real numbers required to achieve this goal should have a known distribution.

Overlapping Sums Test

The elements of a given sequence are converted into a sequence of real numbers. The resulting sequence is divided into parts of the same certain lengths, and the sum is calculated for each part. These amounts must satisfy a normal distribution.

Runs test

The elements of a given sequence are converted into a sequence of real numbers. The number of increasing and decreasing subsequences must satisfy a known distribution.

The Craps Test

The elements of a given sequence are considered as the results of a series of dice games. The number of shots and victories in each game must satisfy a known distribution.

Let us generate a pseudorandom sequence of bits of one Gigabyte size – the *DIEHARDER* package truly requires a huge amount of pseudorandom bits. The results of performed test are given below in the table:

TEST	ntup	P-value	Result
Birthdays	0	0,13881023	Accepted
OPERM5	0	0,97296931	Accepted
32x32 Binary Rank	0	0,16151113	Accepted
6x8 Binary Rank	0	0,45944387	Accepted
Bitstream	0	0,40380726	Accepted
OPSO	0	0,12770146	Accepted
OQSO	0	0,59009804	Accepted
DNA	0	0,37326513	Accepted
Count the 1s (stream)	0	0,12092036	Accepted
Count the 1s (byte)	0	0,55610056	Accepted
Parking Lot	0	0,86287213	Accepted
Minimum Distance (2d Circle)	2	0,13431179	Accepted
Minimum Distance (3d Sphere)	3	0,70260177	Accepted
Squeeze	0	0,99497285	Accepted
Sums	0	0,62783447	Accepted
Runs	0	0,8960232	Accepted
Runs	0	0,65004557	Accepted
Craps	0	0,44784785	Accepted
Craps	0	0,94947798	Accepted

Marsaglia and Tsang GCD Test	0	0,10036426	Accepted
Marsaglia and Tsang GCD Test	0	0,39318209	Accepted
STS Monobit	1	0,52305782	Accepted
STS Runs	2	0,87630962	Accepted
STS Serial (Generalized)	1	0,81616488	Accepted
STS Serial (Generalized)	2	0,69121144	Accepted
STS Serial (Generalized)	3	0,83590182	Accepted
STS Serial (Generalized)	3	0,67625365	Accepted
STS Serial (Generalized)	4	0,75638406	Accepted
STS Serial (Generalized)	4	0,86986159	Accepted
STS Serial (Generalized)	5	0,98313448	Accepted
STS Serial (Generalized)	5	0,98606682	Accepted
STS Serial (Generalized)	6	0,79339032	Accepted
STS Serial (Generalized)	6	0,86131382	Accepted
STS Serial (Generalized)	7	0,87747464	Accepted
STS Serial (Generalized)	7	0,97862638	Accepted
STS Serial (Generalized)	8	0,63723755	Accepted
STS Serial (Generalized)	8	0,99406754	Accepted
STS Serial (Generalized)	9	0,73885199	Accepted
STS Serial (Generalized)	9	0,41677782	Accepted
STS Serial (Generalized)	10	0,97259126	Accepted
STS Serial (Generalized)	10	0,62835195	Accepted
STS Serial (Generalized)	11	0,3896247	Accepted
STS Serial (Generalized)	11	0,72121761	Accepted
STS Serial (Generalized)	12	0,27786178	Accepted
STS Serial (Generalized)	12	0,96933785	Accepted
STS Serial (Generalized)	13	0,73270604	Accepted
STS Serial (Generalized)	13	0,7604679	Accepted
STS Serial (Generalized)	14	0,36197618	Accepted
STS Serial (Generalized)	14	0,92609617	Accepted
STS Serial (Generalized)	15	0,73139774	Accepted
STS Serial (Generalized)	15	0,81629852	Accepted
STS Serial (Generalized)	16	0,38504181	Accepted
STS Serial (Generalized)	16	0,73248843	Accepted

RGB Bit Distribution	1	0	Failed
RGB Bit Distribution	2	4,51E-06	Weak
RGB Bit Distribution	3	0,3253679	Accepted
RGB Bit Distribution	4	0,60881708	Accepted
RGB Bit Distribution	5	0,24114125	Accepted
RGB Bit Distribution	6	0,84836296	Accepted
RGB Bit Distribution	7	0,08663389	Accepted
RGB Bit Distribution	8	0,43784641	Accepted
RGB Bit Distribution	9	0,35432442	Accepted
RGB Bit Distribution	10	0,95077501	Accepted
RGB Bit Distribution	11	0,33248275	Accepted
RGB Bit Distribution	12	0,83070462	Accepted
RGB Generalized Minimum Distance	2	0,02851004	Accepted
RGB Generalized Minimum Distance	3	0,21956052	Accepted
RGB Generalized Minimum Distance	4	0,79459365	Accepted
RGB Generalized Minimum Distance	5	0,80477444	Accepted
RGB Permutations	2	0,68900684	Accepted
RGB Permutations	3	0,81933242	Accepted
RGB Permutations	4	0,37407556	Accepted
RGB Permutations	5	0,39435996	Accepted
RGB Lagged Sum	0	0,57710069	Accepted
RGB Lagged Sum	1	0,69321097	Accepted
RGB Lagged Sum	2	0,97368289	Accepted
RGB Lagged Sum	3	0,15860727	Accepted
RGB Lagged Sum	4	0,51932562	Accepted
RGB Lagged Sum	5	0,02360578	Accepted
RGB Lagged Sum	6	0,76558134	Accepted
RGB Lagged Sum	7	5,935E-05	Weak
RGB Lagged Sum	8	0,73440171	Accepted
RGB Lagged Sum	9	0,22853015	Accepted
RGB Lagged Sum	10	0,69787528	Accepted
RGB Lagged Sum	11	0,01185123	Accepted
RGB Lagged Sum	12	0,98500036	Accepted
RGB Lagged Sum	13	0,07335197	Accepted

RGB Lagged Sum	14	0,81134264	Accepted
RGB Lagged Sum	15	3E-08	Weak
RGB Lagged Sum	16	0,34706515	Accepted
RGB Lagged Sum	17	0,22315068	Accepted
RGB Lagged Sum	18	0,60883384	Accepted
RGB Lagged Sum	19	0,03173787	Accepted
RGB Lagged Sum	20	0,61629974	Accepted
RGB Lagged Sum	21	0,13572091	Accepted
RGB Lagged Sum	22	0,09273275	Accepted
RGB Lagged Sum	23	0,00069544	Weak
RGB Lagged Sum	24	0,15634171	Accepted
RGB Lagged Sum	25	0,38814509	Accepted
RGB Lagged Sum	26	0,15555052	Accepted
RGB Lagged Sum	27	0,01361002	Accepted
RGB Lagged Sum	28	0,87320916	Accepted
RGB Lagged Sum	29	0,36916854	Accepted
RGB Lagged Sum	30	0,39509498	Accepted
RGB Lagged Sum	31	0	Failed
RGB Lagged Sum	32	0,67711933	Accepted
RGB Kolmogorov- Smirnov	0	0,16754083	Accepted
DAB Byte Distribution	0	0	Failed
DAB DCT	256	0,13238942	Accepted
DAB Fill Tree	32	0,11162289	Accepted
DAB Fill Tree	32	0,26502748	Accepted
DAB Fill Tree 2	0	0,00896917	Accepted
DAB Fill Tree 2	1	0,00237664	Weak
DAB Monobit 2	12	1	Failed

Table 3. Results of testing the sequences using DIEHARDER statistical package

Conclusion:

The random number generator under study passes a vast majority of the tests of the *DIEHARDER* package.

Remark

The results of NIST and DIEHARDER statistical tests demonstrate not only excellent statistical properties of the generator under study, but also its cryptographic strength. One of the common methods of hacking a generator or a message, which was masked with a gamma produced by this generator, is a frequency analysis of the generator's output sequence. However, according to the results of the tests, the output of the generator is extremely close to white noise and, therefore, an attacker cannot hope for a certain amount of information due to the presence of systematic deviations or repetitions in the gamma.

An Estimation of generator's period

Consider a linear feedback shift register with a length of L bits equipped with a multiplicative convolution transform. Let the feedback be given by the polynomial $P(x)$ primitive in the field $GF(2)$.

According to the principle of multiplicative convolution, the output bit does not participate in feedback mechnism. The calculation of feedback for a register, equipped with a multiplicative convolution transform, performs in the same way as for an ordinary register. Thus, a generator based on a shift register with a multiplicative convolution goes through exactly $2^L\text{-}1$ different internal states, after which the states begin to repeat, and hence the same happens with output bit. Therefore, there is a precise theoretical estimate of the generator's period at its upper bound:

$$T\_theoretical \leq 2^L - 1$$

Let us perform a series of computational experiments aimed at empirical assessment of the investigated generator period. Using a register of length equal to L bits, we generate a certain quantity of pseudorandom numbers, and each has the same length (L bits). For the small values of L (for example, such as $L= 6; 10$), it is possible to produce sequences with the length that coincides with the period of the generator. For big values of L (for example, such as $L = 66; 82$), we produce a limited sequence of numbers and also take into account the results of *NIST* and *DIEHARDER* statistical tests.

Here we note three important experimental results:

1) ***A sequence of $2^L–1$ numbers, each with a length of L bits, produced by a shift register of the same length (L bits), equipped with a multiplicative convolution transform, can contain repeating elements (that is the contrast from an ordinary shift register, where each of $2^L–1$***

different internal states will be directly extracted as an output). This conclusion follows from the analysis of the output sequence obtained by the register of small length (*L= 6;10*).

2) ***With an increase in the length of the register, equipped with a multiplicative convolution, the distribution of its generated numbers tends to uniform distribution.*** This conclusion follows from the analysis of fragments of the output sequences generated by longer registers (*L= 66; 82*), as well as from the results of statistical tests (the output sequences created by the registers of this length successfully passed all the tests of the *NIST* package and most of the tests of the *DIEHARDER* package).

3) ***If the output sequence contains repeating numbers or fragments, then their repetition is chaotic and their influence become insignificant with an increasing of the register length.*** The conclusion about the chaotic nature of repeating numbers and fragments follows from a direct analysis of the output sequences of the registers. The conclusion that the effect of repeating numbers or fragments become insignificant with an increasing the register length follows from the results of statistical tests. If the repetitions were systematic or even had a strict periodic structure, it would certainly worsen the results of the approximate entropy test, the spectral test and the linear complexity test as well as affect the results of other tests of the *NIST* and *DIEHARDER* packages.

Due to the results of the experiments, the following empirical estimation of the period of the developed generator is valid:

$$T\_empirical = 2^L - 1$$

The closure of M - sequences class

The empirical proof of the «closing» property of the pseudorandom sequences class produced by the random number generator is a weighty argument in favor of its using for cryptographic purposes.

Consider a register with a length of $L=82$ bits, equipped with a multiplicative convolution transform and the linear feedback is given by the polynomial $P(x)=x^{82}+x^{79}+x^{47}+x^{44}+1$ primitive in the field $GF(2)$.

Let us generate two samples of pseudorandom bits and perform bitwise addition of these samples. The result is one new sample, and each bit is the sum by modulo 2 of the two corresponding bits in the first and second samples.

Let us perform a study of the obtained sample using the *NIST* and *DIEHARDER* statistical packages. The summary of results is shown in the table:

	TEST	**RESULT**
1	Frequency	Passed
2	BlockFrequency	Passed
3	CumulativeSums	Passed
4	Runs	Passed
5	LongestRun	Passed
6	Rank	Passed
7	FFT	Passed
8	NonOverlappingTemplate	Passed
9	OverlappingTemplate	Passed
10	Universal	Passed
11	ApproximateEntropy	Passed
12	RandomExcursions	Passed
13	RandomExcursionsVariant	Passed
14	Serial	Passed
15	LinearComplexity	Passed

Table 4. Summary of NIST tests

	TEST	**RESULT**
1	Birthdays	Passed
2	OPERM5	Passed
3	32x32 Binary Rank	Passed
4	6x8 Binary Rank	Passed
5	Bitstream	Passed

6	OPSO	Passed
7	OQSO	Passed
8	DNA	Passed
9	Count the 1s (stream)	Passed
10	Count the 1s (byte)	Passed
11	Parking Lot	Passed
12	Minimum Distance (2d Circle)	Passed
13	Minimum Distance (3d Sphere)	Passed
14	Squeeze	Passed
15	Sums	Passed
16	Runs	Passed
17	Craps	Passed
18	Marsaglia and Tsang GCD Test	Passed
19	STS Monobit	Passed
20	STS Runs	Passed
21	STS Serial (Generalized)	Passed
22	RGB Bit Distribution	Passed
23	RGB Generalized Minimum Distance	Passed
24	RGB Permutations	Passed
25	RGB Lagged Sum	Passed
26	RGB Kolmogorov- Smirnov	Passed
27	DAB Byte Distribution	Failed
28	DAB DCT	Passed
29	DAB Fill Tree	Passed
30	DAB Fill Tree 2	Passed
31	DAB Monobit 2	Failed

Table 5. Summary of DIEHARDER tests

Conclusion 1:

The sequence successfully passes all the *NIST* statistical tests.

Conclusion 2:

The sequence successfully passes the most of the *DIEHARDER* package.

Remark

The obtained empirical estimates of the period of output sequences, as well as the closing property of their class, contribute to the cryptographic strength of the generator under study: a small generator period leads to masking different parts of the original message with the same gamma, which gives information about the message content. The closing property of the pseudorandom sequences class demonstrates both good statistical properties of the output sequence and the impossibility of indirect extracting information from an encrypted message by applying some arbitrary gamma created with the same generator.

Concluding all the ramarks above it is also worth noting the following:
the combination of excellent results of statistical tests and good empirical estimates of the generator period and the closing property of the output sequences gives a strong argument that there is no obvious dependence between the sequence of internal states of the generator and its output sequence, which in essence means cryptographic strength.

FURTHER DISCUSSION AND CONCLUSION

The proposed idea of equipping the shift register with multiplicative convolution transform is completely new and can be developed in several directions at once.

Firstly, there is the possibility of creating a random number generator, which design includes several registers at once and each is equipped with a multiplicative convolution transform. So, for example, we can be guided by the idea of the $A5$ algorithm and combine the outputs of several registers with the XOR operation and each certain register can be equipped with a multiplicative convolution transform. This generator is expected to have a good resistance to correlation attacks, which are actively used by cryptographers to test generators consisting of two or more shift registers. Indeed, most of the known correlation attacks are aimed at determining the relationship between the output of the generator and the output of one of its constituent registers. However, if each register is equipped with a multiplicative convolution transform, then the output of the generator will not be the bitwise sum of the rightmost bits of the registers, but the sum of the multiplicative convolution transform results and this fact can significantly complicate the hacking procedure.

Secondly, it should be noted that the current level of development of computer technology allows working with fairly long shift registers. We can imagine a register divided into k segments with corresponding lengths $p_1 - 1, p_2 - 1, ..., p_k - 1$, where $p_1, p_2, ..., p_k$ are primes of the form $4t + 3$. A single tact of such a register can consist of simultaneous applying k transformations of the multiplicative convolution to the indicated segments of the register with the corresponding prime modules. As a result of this procedure $2k$ register cells will be selected and their contents can be added modulo 2 or immediately sent to the output.

Thirdly, the idea of using cell numbers to enhance the cryptographic strength of the register and improve its statistical properties deserves attention by itself.

Summing up the work done, we will review its most important achievements:

- A new method for generating pseudorandom sequences was invented - that is the generation using a multiplicative convolution transform. This new method is based on well-developed theory of linear feedback shift registers, however, not only the bits of the register, but also their order numbers are involved in the formation of the output sequence, which leads to the absence of linearity among the bits of the output sequence.
- Based on the proposed transformation, the concept of new random number generator was built. The structure of the developed generator have a partial similarity with an ordinary shift register: for example, the feedback calculation mechanism of the developed generator is completely identical to the similar scheme in the ordinary shift register with linear feedback.

The fundamental difference of the presented generator is its method of generating the output bits: the generator's output is formed as a function of all bits of the current state of the register, and all the operations with bits and their order numbers are carried out in the Galois field formed by prime modulus.

- The concept of the proposed random number generator was implemented using modern C / C ++ language standards. Several successive improvements were shown and the final version of the program demostrated its high performance. So, the speed of shift registers equipped with multiplicative convolution varies from 80 to 100 megabits per second depending on their length, and the performance loss compared to an ordinary register of the same length is only about five percent.
- In the final section of the work, it was demonstrated that the proposed random number generator has excellent cryptographic properties: the statistical properties of the generated gamma correspond to the NIST standard for pseudorandom sequences. In addition, the generator passes the most of the tests of the DIEHARDER package, which further convinces of the excellent quality of the generated pseudorandom sequences. The proposed generator also demonstrates its cryptographic resistance to the gamma frequency analysis and shows the high linear complexity, therefore, it is resistant to direct attempts of cracking the initial state using the basic methods of linear algebra and the Berlekamp-Massey algorithm.

Thus, as a result of the work done, the created random number generator is resistant to basic algebraic attacks and can generate high-quality pseudorandom sequences maintaining the high performance of their production.

BIBLIOGRAPHY

[1] J.A. Reeds, "Cracking Random Number Generator", Cryptologia, v. 1, n. 1, Jan 1977, pp.20-26.

[2] J.A. Reeds, "Cracking a Multiplicative Congruential Encryption Algorithm", in Information Linkage Between Applied Mathematics and Industry, P.C.C. Wang, ed., Academic Press, 1979, pp. 467 - 472.

[3] J.A. Reeds, "Solution of Challenge Cipher", Cryptologia, v. 3, n. 2, Apr 1979, pp. 83-95.

[4] J.B. Plumstead, "Inferring a Sequence Generated by a Linear Congruence", Proceedings of the 23rd IEEE Symposium on the Foundations of Computer Science, 1982, pp. 153-159.

[5] A.M. Frieze, J. Hastad, R. Kannan, J.C. Lagarias, and A. Shamir, "Reconstructing Truncated Integer Variables Satisfying Linear Congruenccs", SIAM Journal on Computing, v. 17, n. 2, Apr 1988, pp. 262-280.

[6] A.M. Frieze, R. Kannan, and J.C. Lagarias, "Linear Congruential Generators loo not Produce Random Sequences", Proceedings of the 25th IEEE Symposium on Foundations of Computer Science, 1984, pp. 480-484.

[7] J.Hastad and A.Shamir, "The Cryptographic Secunty of Truncated Linearly Related Variables", Proceedings of the 1 7th Annual ACM Symposium on the Theory of Computing, 1985, pp. 356-362.

[8] E.L. Key, "An Analysis of the Structure and Complexity of Nonlinear Binary Sequence Generators", IEEE Transactions on Information Theory v. IT-22, n. 6, Nov 1976, pp. 732-736.

[9] K.C. Zeng, C.-H. Yang, and T.R.N. Rao, "On the Linear Consistency Test ILCTl in Cryptanalysis with Applications", Advances in Cryptology CRYPT 0 89 Proceedings, Springer-Verlag, 1990, pp. 164-174.

[10] K.C. Zeng, C.-H. Yang, L. Wei, and T. R.N. Rao, "Pseudorandom Bit Generators in Stream-Cipher Cryptography", IEEE Computer, v. 24, n. 2, Feb libel, pp. 5-17.

[11] J.O. Bruer, "On Pseudo Random Sequences as Crypto Generators", Proceedings of the International Zurich Seminar on Digital Communication, Switzerland, 1984.

[12] D. Gollmann, "Kaskadenschaltungen takt gesteuerter Schicberegister als Pseudozufallszahlengencratoren", Ph.D. disserta tion, Universitat Linz, 1983. (In German).

[13] W.G.Chambers and D.Gollmann, "Lock-In Effect in Cascades of Clock-Controlled Shift Registers", Advances in Cryptology EUROCRYPT '88 Proceedings, Springer-Verlag, 1988, pp.333-343.

[14] D. Gollmann, "Pseudo Random Properties of Cascade Connections of Clock Controlled

Shift Registers", Advances in Cryptology: Proceedings of EUROCRYPT 84, Springer- Verlag, 1985, pp. 93-98.

[15] D. Gollmann, "Correlation Analysis of Cascaded Sequences", Cryptography and Coding, H.J.Beker and F.C. Piper, eds., Oxford: Clarendon Press, 1989, pp. 289-297.

[16] D. Gollmann, "Transformation Matrices of Clock-Controlled Shift Registers", Cryptography and Coding 111, M.J. Ganley, e d., Oxford: Clarendon Press, 1993, pp. 197-210.

[17] Bruce Schneier, "Applied Cryptography
Protocols, Algorithms, Source Code in C", Triumph, Moscow, 2002.

[18] S.B. Xu, INK. He, and X.M. Wang, "An Implementation of the GSM General Data Encryption Algorithm A5", CHINACRYPT 94, Xidian, China, 11-1S Nov 1994, pp. 287-291.(In Chinese)

[19] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Special Publication 800-22 Revision 1a. National Institute of Standards and Technology