symmetry

Article

High Performance Parallel Pseudorandom Number Generator on Cellular Automata

Alla Levina [1,*], Daniyar Mukhamedjanov [1], Danil Bogaevskiy [1], Pavel Lyakhov [1,2], Maria Valueva [3] and Dmitrii Kaplun [1]

[1] Faculty of Computer Science and Technology, Saint Petersburg Electrotechnical University "LETI", 197022 St. Petersburg, Russia; dmukhamedjanov@etu.ru (D.M.); dvbogaevskiy@etu.ru (D.B.); pliakhov@ncfu.ru (P.L.); dikaplun@etu.ru (D.K.)
[2] Department of Applied Mathematics and Mathematical Modeling, North-Caucasus Federal University, 355000 Stavropol, Russia
[3] Department of Number Theoretic Systems, North-Caucasus Federal University, 355000 Stavropol, Russia; mvalueva@ncfu.ru
* Correspondence: ablevina@etu.ru

Abstract: Nowadays, the practice of developing algorithms to maintain the confidentiality of data shows that there is a lack of some features, such as velocity, predictability, etc. Generating pseudorandom numbers is one such problem that lies in the basement of many algorithms, even in hardware microprograms. An unreliable generator can cause cyberattacks on it, despite the security in the upper layers. At the same time, the algorithm should be fast enough to provide uninterrupted circuit work for the entire system. The paper presents a new algorithm generating pseudorandom numbers on cellular automata, which is not only fast and easy-repeating, but unpredictable enough and can be used in cryptographic systems. One of the main tasks of pseudorandom number generators (PRNG) is to present a high level of nonlinearity, or as it can also be named, asymmetry. Using the National Institute of Standards and Technology (NIST) statistical test suite for random number generators and pseudorandom number generators, it is shown that the presented algorithm is more than three times superior to the state-of-the-art methods and algorithms in terms of *p*-value. A high level of the presented algorithm's parallelization allows for implementation effectively on calculators with parallel structure. Central Processing Unit (CPU)-based architecture, Field-Programmable Gate Array (FPGA)-based architecture, Compute Unified Device Architecture (CUDA)-based architecture of PRNG and different PRNG implementations are presented to confirm the high performance of the proposed solution.

Keywords: cellular automata; pseudorandom number generation; parallel computing; hardware-based parallel implementation

check for updates

Citation: Levina, A.; Mukhamedjanov, D.; Bogaevskiy, D.; Lyakhov, P.; Valueva, M.; Kaplun, D. High Performance Parallel Pseudorandom Number Generator on Cellular Automata. *Symmetry* **2022**, *14*, 1869. https://doi.org/10.3390/sym14091869

Academic Editor: Sergei D. Odintsov

Received: 29 July 2022
Accepted: 2 September 2022
Published: 7 September 2022

1. Introduction

There is a large interest in the world research society in the security and ways of its realization in both hardware and software engineering. Nowadays, all areas of IT need to be protected from cyberattacks; some attacks are presented in [1–6]. Building secure software on many levels, such as network, application, etc., gave protection from variable attacks and increased level of confidence. To protect all layers of systems can be used ciphers [7–9] and/or error-detecting/error-correcting codes, some of them are based on random numbers. For an example, random numbers used in PIN and password generation (PIN Protection Principles, ANSI X9.8:1, Password Generation, FIPS 181-1993), and prime generation (DSA, ANSI X9.30, RSA, ANSI X9.31, Prime Number Generation, ANSI X9.80) random challenges for authentication (Entity Authentication using PKC, FIPS 196) and key confirmation (NIST Key Schemes Recommendation). Such algorithms need to use absolutely random numbers, which are generated by physical processes and then are handled with hardware and software to obtain the correct formal number [10,11].

Random numbers are needed in various applications, but finding good random number generators is a difficult task [11]. All practical methods for generating random numbers are based on deterministic algorithms, so such numbers are more correctly called pseudo-random numbers, since they differ from true random numbers obtained as a result of some natural physical process.

Many systems must have the ability to process the same numbers again under the same conditions and with data of the same origin. For these needs, a pseudorandom generator can be used that can repeat the same numbers as much as it will be needed. There are many examples of using various algorithms, such as linear shift or Yarrow (or Fortuna) [12], which are very good for their options, and some of them are even cryptographically secure [13].

Cryptographic applications use deterministic algorithms to generate random numbers, therefore generating a sequence of numbers that theoretically cannot be statistically random. At the same time, if you choose a good algorithm, the resulting numerical sequence—pseudorandom numbers—will pass most tests for randomness. One of the characteristics of such a sequence is a long repetition period [14].

Examples of well-known cryptographically strong PRNGs are RC4, ISAAC [15], SEAL [16], SNOW [17], the very slow theoretical Bloom–Blum–Shuba algorithm [18], as well as counters with cryptographic hash functions or cryptographically secure block ciphers instead of the output function.

In addition, cryptographically strong ciphers include generators with multiple shift registers, generators with nonlinear transformations, and majority encryption generators such as $A5/x$.

Many researchers are focusing now on the creation of PRNG based on physical processes, some latest results in this area can be found in [19–21], the authors present PRNG based on a discrete hyper-chaotic system with an embedded cross-coupled topological structure, Hopfield Neural Network, and memristive Hopfield neural network (MHNN).

In 2022, the first PRNG algorithm based on the absolute value function Itamaracá [22] was presented; it is also a simple and fast model that generates aperiodic sequences of random numbers.

In this paper, we present a new algorithm for generating pseudorandom numbers, based on cellular automata, and illustrate its implementation on FPGA and CUDA. Nowadays, there are a number of PRNG algorithms based on CA using one or more rules on the same grid with standard neighborhood patterns, using LFSR, gaps, delays, and, less commonly, nonlinear functions. In this work, the application of several neighborhood templates, several independent cellular automata, and implementation of transitions of various cellular automata according to various sets of rules and expansion of rule sets based on the complementarity of the number of ones and zeros for a statistically normal distribution will be used.

The paper describes in detail the principle of operation of the presented algorithm, its strictly formalized mathematical model, the properties used in the development and the mathematical models and concepts underlying the generator. The paper also contains a description of the tests performed from the ready-made statistical software package, the grounds for these tests, a description of each of them, and the results and performance of the algorithm on the selected tests.

Compared to the existing algorithm, the presented algorithm is different from the previous and similar ones in several respects:

1. better results in some NIST test;
2. work speed;
3. parallelization in hardware implementation;
4. flexibility;
5. simplicity of architecture.

The paper is organized as follows: Section 1 presents an overview of ideas used in random number generators, the mathematical structure of homogeneous structures (HS) and cellular automata, Section 2 illustrates the created algorithm, giving a mathematical explanation of the presented algorithm, Section 3 demonstrates test results, Section 4 illustrates the implementation of the created algorithm on FPGA, and Section 5 gives CPU and CUDA implementation of the algorithm. In conclusion, a short overview of the presented work is given.

2. Random Number Generators Overview

Random numbers are needed in a variety of applications, yet finding good random number generators (RNG) is a difficult task. In recent years, some new results of building pseudorandom number generators (PRNGs) based on chaos and chaotic map were presented [23–27] and in work [28].

Actually, 'true' random generation is mostly based on some kind of natural physical processes, or in another way, the 'noise' source features of some machine hardware may be used. Simplifying, there are many ways to determine some noises as "1" or "0", giving a sequence of bits, and as a result a number. This number is exactly what we need.

However, the difference between these and pseudorandom generators is in uniqueness, which means that there is no chance of obtaining the same number more than once. The great part of computations where a random number should be used is related to cryptography. For simplicity, there is a need to control the calculation of the speed, effectiveness, and repetition generated numbers (PRNG). Some examples to understand the general idea of such generators will be presented in the following:

There exist many ways to generate pseudorandom numbers on a computer; the most popular one is the Linear Congruential Generator (LCG). In 1951, Lehmer generator [29] was published, and LCG was published in 1958 [30,31]. Today, LCG can be called the method most commonly used for generating pseudorandom numbers. An advantage of LCGs is that the period is known and long with the appropriate parameters. Although not the only criterion, a short period is a fatal flaw in a PRNG [32]. LCGs based on the following recurrent formula:

$$X_{n+1} = (aX_n + c) \bmod m, n \geq 0, m > 0, 0 < a < m$$

The value m is the module, a is the multiplier, and c is an additive constant. The sequence has a maximum possible period m, after which it starts to repeat itself [32]. LCGs are very popular among researchers, and most mathematical software packages include them. So-called lagged Fibonacci generators are also widely used. They are of the form:

$$X_n = (X_n r \circ X_n p) \bmod m$$

The numbers r and p are called "lags", and there are methods to choose them appropriately. The operator $\circ$ can be one of the following binary operators: addition, subtraction, multiplication, or exclusive or. However, it should be noted that, from the point of view of hardware implementation, both congruential and lagged Fibonacci RNGs are not very suitable: they are inefficient in terms of silicon area and time when applied to fine-grained massively parallel machines, for built-in self-test or for other on-board applications [33].

There is a sufficiently large roadmap to achieve the PRNG objective, which has features such as simplicity, velocity, and unpredictability.

3. Homogeneous Structures (HS) and Cellular Automata (CA)

Homogeneous structures (HS) [34] may be formally described as $\sigma = (Z^k, E_n, V, \phi)$, where Z^k—set of k-dimensional vectors, $E_n = 0, 1, \ldots, n-1$—set of states of one cell in ϕ, $V = (\alpha_1, \ldots \alpha_{(h-1)})$—neighborhood template (ordered set of distinct k-dimensional vectors from Z^k), $\phi = \phi(x_0, x_1, \ldots, x_{(h-1)})$, $\phi : (E_n)^h \to E_n$—local transition function ($\phi(0, 0, \ldots, 0) = 0$).

From the definition of HS, it can be seen that HS can be compared to the set of ordinary Moore automata [35] if their states depended on the states of neighboring countries. Actually, the neighborhood scheme can differ from each other (Neuman's scheme is more like a symbol plus with the changeable central cell, Moore's scheme is more like the square 3×3). There are a great number of combinations that can be used.

In this paper, only one class of HS will be used. It can be represented like $S = (k, n, m)$, where k and n are described above, and m is the set of $m_i > 0, i = 1, \ldots, k$. Another value needs to be determined—$P = V_m(\alpha)$. However, with some restrictions for values $k, n, (k = n = 2)$, it is obvious that $|S| = 2^{(2^{(h-1)})}$.

Actually, as a result of some HS, without loss of generality for this research, this object can be replaced by CA. Cellular automata (CA) [34] are dynamical systems in which space and time are discrete. A cellular automata consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps, according to a local, identical interaction rule. Here, we will only consider Boolean automata for which the cellular state $s \in 0, 1$. The state of a cell at the next time step is determined by the current states of the surrounding neighborhood of cells. Cellular array (grid) is d-dimensional, where $d = 1, 2, 3$ is used in practice; as an example for simplicity of understanding, we shall concentrate on $d = 1$, i.e., one-dimensional grids.

Each rule in every single cell can be represented as a simple finite automata (or finite-state machine) with its input states, transition function, and output state, which is the result of using the rule function with input state as an entry [36,37]. However, in terms of cellular automata, there is an entry as a set of states, called neighborhood, when the next state is in formal dependence of states of the particular target cell and its surrounding cells' states [38,39]. For one-dimensional CA (Figure 1), the target cell is surrounded by r neighbors (cells) at a discrete moment on the appropriate side (by template) where r cells are called radius (each cell has $2r + 1$ cells, which impact its next state).

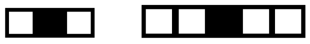


Figure 1. Neighborhood (one-dimensional). **left**: [r = 1]; **right**: [r = 2].

Besides one-dimensional CAs, there are two main templates for two-dimensional CAs (Figure 2): the first is the so-called von Neumann neighborhood [40] (when the target cell is surrounded by four nondiagonal cells) and the second is called the Moore neighborhood [41] (when the target cell is surrounded by eight cells).

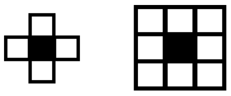


Figure 2. Neighborhood (two-dimensional). **left**: Neumann neighborhood; **right**: Moore neighborhood.

There are also two types of automata, which differ from each other by some kind of complexity in terms of rules. The fact is that one type, called uniform CA, determines one rule for all cells in the grid, but the second type, non-uniform [42] or inhomogenous may have different rules for cells. At the same time, they both have the same features of simplicity, locality, and parallelism with the difference in realization (inhomogenous ones require more memory sources for describing rules). Additionally, when we think of finite grid of CAs, it should be highlighted that one-dimensional CAs grids are represented by the circular structure, when two-dimensional ones have a grid in toroidal form.

According to Wolfram's notation [43], firstly, it should be determined that the configuration is the set of ones and zeros (two-state CA, where the sequence is considered as a random number) at a particular discrete moment of time. Wolfram also suggested additional rule numbers. Let us describe the most popular Wolfram's *Rule 30* as an example:

In Boolean form, *Rule 30* can be written as

$$f_0(t+1) = f_{(-1)}(t) \oplus (f_0(t) \vee f_1(t))$$

where the radius is $r = 1$ and $f(t)$ is the state of cell i at time t. As we can see from this formula, $f_0(t+1)$ is the next state of the cell i at $t+1$ step, which is a simple Boolean function of neighbor cells states and target cell state at the step t. That is how a random sequence of bits is obtained, counting every cell as a target cell in the same moment for all the cells. There are several ways to make the sequence more complex and unpredictable, e.g., time spacing and site spacing.

Time spacing is a method of generating configurations, when not every configuration is a part of the resulting sequence; this means that some configurations are generated to produce new ones only, but it will not be in the resulting sequence.

Site spacing is a method of generating sequences when only particular cells' states in the whole configuration (or a row) are bits of the resulting sequence. These methods are obviously empowered by unpredictability feature for whole sequences, but, of course, time or memory sources with these methods remain the same.

Another example in order to show how Wolfram's rules can be encoded. For an example: $f(111) = 1, f(110) = 0, f(101) = 1, f(100) = 1, f(011) = 1, f(010) = 0, f(001) = 0, f(000) = 0$, is denoted *Rule 184*.

As an example of non-uniform CA, two rules can be taken: *Rule 90* and *Rule 150*, which are rules for particular cells in the grid. In Boolean form, the *Rule 90* can be written as

$$f_0(t+1) = f_{(-1)}(t) \oplus f_1(t)$$

and *Rule 150*:

$$f_0(t+1) = f_{(-1)}(t) \oplus f_0(t) \oplus f_1(t)$$

These rules perform a sufficiently effective random number generator, in spite of the fact that they can be simply described as linear Boolean functions.

4. Algorithm of Generating Pseudorandom Numbers on Cellular Automata

The main idea of the presented algorithm is to combine the best practices for generating pseudorandom numbers by cellular automata and a number of improvements to enforce the general features of cellular automata, saving the best. Figure 3 presents the flowchart of the proposed algorithm.

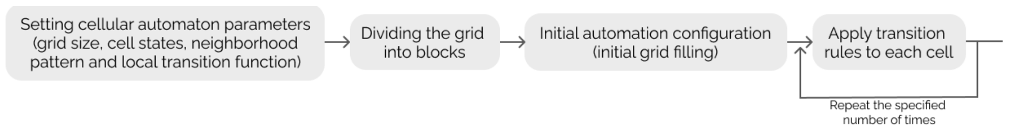

Setting cellular automaton parameters (grid size, cell states, neighborhood pattern and local transition function)
Dividing the grid into blocks
Initial automation configuration (initial grid filling)
Apply transition rules to each cell
Repeat the specified number of times

Figure 3. The flowchart of the proposed algorithm.

There are several subsections, which describe step-by-step all levels of algorithm evolution. In addition, there are some ideas for improving the presented algorithm with the help of genetic algorithms to expand the set of rules to be used to generate a pseudo-random number.

4.1. Grid

The main result of the presented algorithm is that it can generate pseudo-random sequences of numbers. Despite the fact that CA had already been used in such a way, we improved some characteristics by adding new conditions and ideas. There is a grid, where each cell only has two states: 0 and 1, and states are changed by some function. The result of steps in algorithm will be a binary performed number, which can be formed from the grid (or part of it). Thus, we have a grid with its sizes: p and q, which are both prime numbers (to improve periodic features). Changes start right from this stage: dividing all the grid into $m \geq 2$ blocks, which are formed as rectangles b_i with sizes l_{b_i} and w_{b_i}, and each block consists of $l_{b_i} \times w_{b_i}$ cells (Figure 4). Thus, we obtain the equation:

$$p \times q = \sum_{2 \leq i \leq m} (l_{b_i} \times w_{b_i})$$

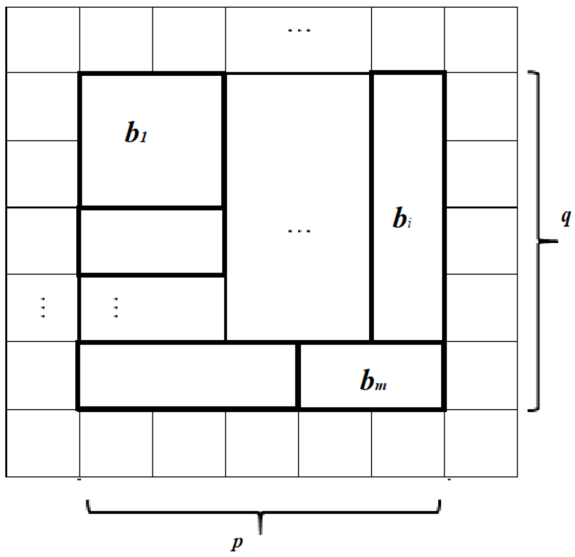


Figure 4. Whole grid view.

This is quite a simple method, where each block needs to be placed in the grid, called the compass (NESW—North, East, South, West) (Figure 5). The matter is that there will be a rule by which blocks will be placed from the lower left corner and move to the North (up) till the reaching of edge of the grid or to the next block, then to the East (right), South (down) and West (left), according to their size values. This rule may vary for more flexibility, i.e., "snake" method of filling the grid (filling rows from the bottom to the top with switching to the next row from opposite sides).

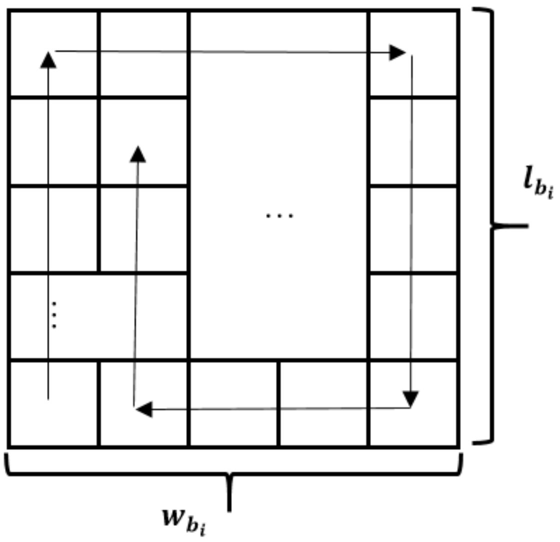


Figure 5. b_i block grid view with the NESW scheme of filling.

Here is an example of using the NESW method of rule. Let us take rule 15 (00001111 in binary representation) and fill the grid with sizes 4×5 (Figure 6).

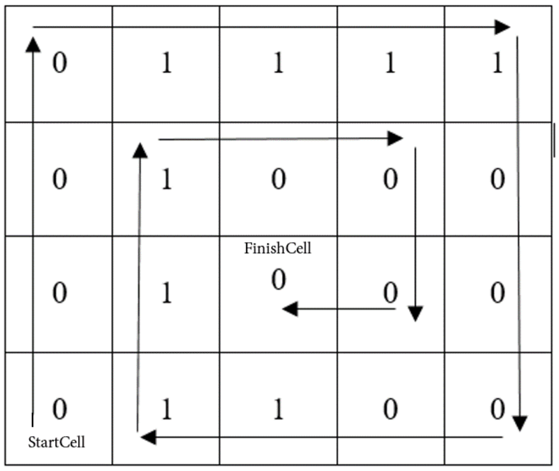


Figure 6. NESW method using an example.

4.2. Rules and Neighborhood

We will work with only two-dimensional space; this means that we take $d = 2, n = 2$, (Z^2, E_2). Firstly, start configuration will be formed by the rule NESW, the same as blocks. It means that number $l_{b_i} \times w_{b_i}$ in binary form will fulfill the blockgrid along the narrowing edges, etc. What about the rules? The solution of this question may be very flexible because the algorithm is provided by two kinds of rules: numbered rules (like the rule 30 at the beginning of this article) and simple operations (their combination). Each block can have its own rule, which makes the whole generating more non-uniform, which leads to unpredictability and better periodic features. Here comes an idea which involves the set of rules in the table, where each rule (both kinds) is encrypted in some combinations, depending on the number of using rules.

Thus, the rule for each block may be chosen by the simple *mod* operation from the $l_{b_i} \times w_{b_i}$ in case of using every possible rule in the block. However, for the best results, there should be chosen several rules with an equal number of ones and zeros (it may be used for the whole grid—if we have two blocks and three ones in the first block rule, so the second block should have num-3 ones, where num is a length of rule-vector if both vectors are equal). Additionally, to avoid bad generations, we should determine certain rules for the grid. In addition, we could fill the "space" between the actual size of the grid and formal. In order to avoid collisions of values, we should make our grid less than its actual size for one cell (or two, depending on the neighborhood template) on each side.

The last question in generating our pseudorandom numbers is a neighborhood. The fact is that, as far as we have blocks, we can choose several samples for some blocks and its number and variety depending on the number of blocks. We can not use a number of rules more than the number of blocks, but we can use similar ones for several blocks.

4.3. "Repeat Please"

Pseudorandom number generator must have the option of repeating the algorithm in order to obtain a similar number after the same operations with the same conditions and configurations. Actually, not to pass the result number, we can only send the key K—the sequence of our data in order to repeat operations on the other end, if needed. We use symbol | to mark concatenation:

$$K = p|q|l_{b_1}|w_{b_1}|\ldots|l_{b_m}|w_{b_m}|V_1|\ldots|V_m|T$$

where V_i is sent like the sequence of 9 bits where each bit, if it is 1, then cell is in the sample, and, when it is 0, it is out of sample.

$(-1, 1)$	$(0, 1)$	$(1, 1)$
$(-1, 0)$	$(0, 0)$	$(1, 0)$
$(-1, -1)$	$(-1, 0)$	$(1, -1)$

Therefore, this neighborhood mapping table becomes such a string

$$-1, -1; 0, -1; 1, -1; -1, 0; 0, 0; 1, 0; -1, 1; 0, 1; 1, 1$$

Here are the numbers of coordinates, instead of which, it should be placed 1 or 0. T is a table, where each using rule may be encrypted, for example, a basic binary number. Thus, in spite of the fact that we must send such a long-sized key, we can put some data in the boundary cells, placed around our main grid in order to save correct transition of state. We can save $|V_1|\dots|V_m|T$ part of the key in these cells, depending on its complexity and size.

4.4. Additional Function

To be sure that our algorithm will generate high-quality random number sequences, we decided to use some nonlinear functions at this level of the development algorithm. The reason is in the CA rules. The fact is that only part of the rules has good random features (the ability to generate complex structures). However, if we take only this part of the rules, we obtain too small a set of needed ones, and there are two ways to improve it. The first one is to add a nonlinear function for XOR with the subset of the grid for making it more complex. The second is that we can choose "good" (having randomness features) packs of rules with the help of genetic algorithms or some kind of tutor algorithm. Therefore, the evolution of CAs will show us only those rules which can be used in practice.

5. Test Results

Our algorithm was tested with the help of the NIST Test Suite, which was developed to test RNG and PRNG [44]. The process of the test involved: generation sequences of bits by our algorithm (around 1,000,000 bits), testing.txt file with a sequence with an NIST test suite. The 10,000 variations of the rules pack and ways of filling the grid were tested, as sizes of grids. As a result, the number of the most "productive" packs of rules-grid size-neighborhood were highlighted.

Thus, we used the grid with sizes 307×53, divided by six blocks of approximately equal lengths, while the width of each block was 53. Each block was filled with bits, generated by results of multiplication of blocks' sizes, with the NESW method. The neighborhood template was simple and even classical enough; it was a 3B rule (center cell change state in dependence of its previous state and left and right neighbors' ones). The package of rules was: $\{45, 75, 89, 101, 135, 86\}$.

Rule 45:

000	001	010	011	100	101	110	111
1	0	1	1	0	1	0	0

Rule 75:

000	001	010	011	100	101	110	111
1	1	0	1	0	0	1	0

Rule 89:

000	001	010	011	100	101	110	111
1	0	0	1	1	0	1	0

Rule 101:

000	001	010	011	100	101	110	111
1	0	1	0	0	1	1	0

Rule 135:

000	001	010	011	100	101	110	111
1	1	1	0	0	0	0	1

Rule 86:

000	001	010	011	100	101	110	111
0	1	1	0	1	0	1	0

5.1. Short Description of Criteria with Results

The test statistic is used to calculate a p-value that summarizes the strength of the evidence against the null hypothesis. For these tests, each p-value is the probability that a perfect random number generator would have produced a sequence less random than the sequence that was tested, given the kind of non-randomness assessed by the test. If a p-value for a test is determined to be equal to 1, then the sequence appears to have perfect randomness. A p-value of zero indicates that the sequence appears to be completely non-random. A significance level (α) can be chosen for the tests. If p-value $\geq \alpha$, then the null hypothesis is accepted; i.e., the sequence appears to be random. If p-value $< \alpha$, then the null hypothesis is rejected; i.e., the sequence appears to be non-random. The parameter α denotes the probability of the $Type I$ error (if the data are, in truth, random, then a conclusion to reject the null hypothesis (i.e., conclude that the data are non-random) will occur a small percentage of the time) [44], α is equal to 0.01 [44].

5.1.1. Frequency (Monobit) Test

The focus of the test is the proportion of zeros and ones for the entire sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence is approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to 0.5, that is, the number of ones and zeros in a sequence should be about the same. All subsequent tests depend on the passing of this test [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.744146	99.6%	Frequency

5.1.2. Frequency Test within a Block

The focus of the test is the proportion of the ones within M-bit blocks. The purpose of this test is to determine whether the frequency of ones in an M-bit block is approximately $M/2$, as would be expected under an assumption of randomness. For block size $M = 1$, this test degenerates to the Frequency (monobit) test [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.380537	99.4%	BlockFrequency

5.1.3. Runs Test

The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits. A run of lengths k consists of exactly k identical bits and is bounded before and after with a bit of opposite value. The purpose of the run test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.428244	98.5%	Runs

5.1.4. Longest Run of Ones in a Block

The focus of the test is the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. Note that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Therefore, only a test for ones is necessary [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.383827	99.1%	LongestRun

5.1.5. Rank Test

The focus of the test is the rank of disjoint sub-matrices of the entire sequence. The purpose of the test is to check for linear dependence among fixed length substrings of the original sequence [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.702458	99.0%	Rank

5.1.6. Discrete Fourier Transform (Spectral) Test

The focus of this test is the peak heights in the Discrete Fourier transform of the sequence. The purpose of this test is to detect periodic features (i.e., repetitive patterns that are close to each other) in the sequence tested that would indicate a deviation from the assumption of randomness. The intention is to detect whether the number of peaks exceeding the 95% threshold is significantly different than 5% [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.650637	99.5%	FFT

5.1.7. Non-Overlapping Template Matching Test

The focus of this test is the number of occurrences of prespecified target strings. The purpose of this test is to detect generators that produce too many occurrences of a given non-periodic (aperiodic) pattern. For this test and for the Overlapping Template Matching test, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides into a bit position. If the pattern is found, the window is reset to the bit after the found pattern and the search is resumed [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.433358	98.7%	NonOverlappingTemplate

5.1.8. Overlapping Template Matching Test

The focus of the Overlapping Template Matching test is the number of occurrences of prespecified target strings. Both this test and the Non-overlapping Template Matching test use an m-bit window to search for a specific m-bit pattern. As in the previous test, if the pattern is not found, the window slides one bit in the position. The difference between this test and the previous test is that; when the pattern is found, the window slides only one bit before resuming the search [44].

***p*-Value**	**Proportion**	**Statistical Test**
0.632191	100%	OverlappingTemplate

5.1.9. Maurer's "Universal Statistical Test"

The focus of this test is the number of bits between the matching patterns (a measure that is related to the length of a compressed sequence). The purpose of the test is to detect whether the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random [44].

p-Value	Proportion	Statistical Test
0.414862	98.3%	Universal

5.1.10. Linear Complexity Test

The focus of this test is the length of a linear feedback shift register (LFSR). The purpose of this test is to determine whether the sequence is complex enough to be considered random. Random sequences are characterized by longer LFSRs. An LFSR that is too short implies non-randomness [44].

p-Value	Proportion	Statistical Test
0.730485	100%	LinearComplexity

5.1.11. Serial Test

The focus of this test is the frequency of all possible overlapping m-bit patterns throughout the sequence. The purpose of this test is to determine whether the number of occurrences of the 2mm-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity; that is, every m-bit pattern has the same chance of appearing as every other m-bit pattern [44].

p-Value	Proportion	Statistical Test
0.651956	99.3%	Serial

5.1.12. Approximate Entropy Test

The focus of this test is the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of the test is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and $m + 1$) against the expected result for a random sequence [44]:

p-Value	Proportion	Statistical Test
0.778903	99.1%	Approximate Entropy

5.1.13. Cumulative Sums (Cusums) Test

The focus of this test is the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted (-1, +1) digits in the sequence. The purpose of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. This cumulative sum may be considered as a random walk. For a random sequence, the excursions of the random walk should be near zero. For certain types of non-random sequences, the excursions of this random walk from zero will be large [44].

p-Value	Proportion	Statistical Test
0.734146	99.9%	CumulativeSums

5.1.14. Random Excursions Test

The focus of this test is the number of cycles that have exactly K visits in a cumulative sum random walk. The cumulative sum random walk is derived from partial sums after the (0, 1) sequence is transferred to the appropriate (-1, +1) sequence. A cycle of a random walk consists of a sequence of steps of unit length taken at random that begin at and return to the origin. The purpose of this test is to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence. This test is actually a series of eight tests (and conclusions), one test (conclusion) for each of the states -4, -3, -2, -1, +1, +2, +3, +4 [44].

p-Value	Value
0.673779	−4
0.640660	−3
0.636757	−2
0.906842	−1
0.614216	+1
0.042225	+2
0.570447	+3
0.802887	+4

5.1.15. Random Excursions Variant Test

The focus of this test is the total number of times a particular state is visited (that is, occurs) in a cumulative sum random walk. The purpose of this test is to detect deviations from the expected number of visits to various states in the random walk. This test is actually a series of 18 tests (and conclusions), one test (conclusion) for each of the states: −9. . . −1 and +1. . . +9 [44].

p-Value	Value
0.148224	−9
0.293802	−8
0.799213	−7
0.810598	−6
0.589138	−5
0.525096	−4
0.584469	−3
0.659030	−2
0.878513	−1
0.878513	+1
0.469278	+2
0.502912	+3
0.711568	+4
0.910749	+5
0.860976	+6
0.702797	+7
0.532906	+8
0.543193	+9

The choice of rules is determined by good randomness and an equal number of ones and zeros in each of these particular rules. Table 1 presents the results of the NIST test suite package [44] with the input of the file, generated by the developed algorithm.

Table 1. Results for the uniformity of *p*-values and the proportion of passing sequences.

p-Value	Proportion	Statistical Test
0.744146	99.6%	Frequency
0.380537	99.4%	BlockFrequency
0.734146	99.9%	CumulativeSums
0.428244	98.5%	Runs
0.383827	99.1%	LongestRun
0.702458	99.0%	Rank
0.650637	99.5%	FFT
0.433358	98.7%	NonOverlappingTemplate
0.632191	100%	OverlappingTemplate
0.414862	98.3%	Universal
0.778903	99.1%	ApproximateEntropy
0.651956	99.3%	Serial
0.730485	100%	LinearComplexity

All p-values in this table are averaged from 10–100 rounds of tests.

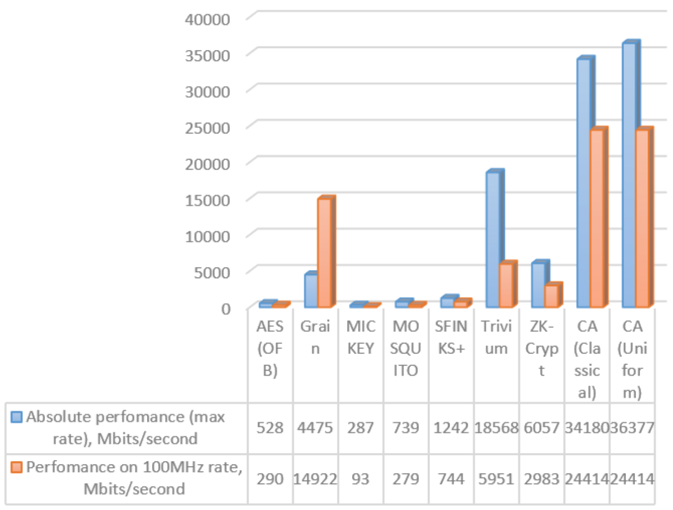
In addition, there is a comparison (Table 2) between the measurements of the NIST test suite of the presented method and the Linear Congruential method, and it is observed that the presented method obtains better results in some tests, and the p-value in some tests is much higher.

Table 2. Comparison of the algorithm on CA and the linear congruential method.

CA		**Linear Congruential**		**Statistical Test**
***p*-Value**	**Proportion**	***p*-Value**	**Proportion**	
0.744146	99.6%	0.739918	99.8%	Frequency
0.380537	99.4%	0.122325	99.0%	BlockFrequency
0.734146	99.9%	0.689918	100%	CumulativeSums
0.428244	98.5%	0.213309	98.7%	Runs
0.383827	99.1%	0.122325	98.1%	LongestRun
0.702458	99.0%	0.213309	99.0%	Rank
0.650637	99.5%	0.639918	100%	FFT
0.433358	98.7%	0.578346	98.4%	NonOverlappingTemplate
0.632191	100%	0.350485	100%	OverlappingTemplate
0.414862	98.3%	0.213309	98.3%	Universal
0.778903	99.1%	0.791468	100%	ApproximateEntropy
0.610977	98.8%	0.534146	99.2%	RandomExcursions
0.633388	99.2%	0.468312	100%	RandomExcursionsVariant
0.651956	99.3%	0.615983	100%	Serial
0.730485	100%	0.213309	100%	LinearComplexity

6. FPGA Implementation

The presented algorithm has two implementations, one on FPGA and one on CUDA. Here, there is a chart (Figure 7) from the paper [45], which shows the velocity of a pseudo-random numbers' generator, based on a cellular automaton implemented on FPGA. There are several algorithms, which were measured on the same FPGA platform from eSTREAM (as pseudorandom generators, which were considered strictly and detailed enough for their process velocity and statistical features) [46,47].



	AES (OFB)	Grain	MICKEY	MOSQUITO	SFINKS+	Trivium	ZK-Crypt	CA (Classical)	CA (Uniform)
🟦 Absolute perfomance (max rate), Mbits/second	528	4475	287	739	1242	18568	6057	34180	36377
🟧 Perfomance on 100MHz rate, Mbits/second	290	14922	93	279	744	5951	2983	24414	24414

Figure 7. Different generating pseudorandom numbers' algorithm performance.

Hardware modeling was performed on Artix 7 xc7a200tfbg676-3 in Vivado 2016.3. The synthesis parameters are shown in Table 3. The modeling results are presented in Table 4. There are not enough resources of the device for size 1024 on 1024 and 4096 on 4096. Absolute performance of the proposed device is up to 1540 times greater than that of the known devices shown in Figure 7, and performance at the 100 MHz rate of the proposed device is up to 1074 times higher than that of the known ones.

Table 3. Synthesis parameters.

Parameters	**Results**
Flatten hierarchy during LUT mapping	rebuilt
Convert clock gating logic to flop enable	off
Max number of global clock buffers used by synthesis	12
Fanout limit	400
Synthesis directive	Default
Retiming	Not checked
FSM Extraction Encoding	One_hot
Keep equivalent registers	Checked
Resource sharing	off
Control set optimization threshold	Auto
Disable LUT Combining	Checked
Min length for chain of registers to be mapped onto SRL	5
Max number of block RAM allowed in design	Max number allowed for the part in question
Max number of Ultra RAM blocks allowed in design	Max number allowed for the part in question
Max number of block DSP allowed in design	Max number allowed for the part in question
Max number of BRAM that can be cascaded by the tool	Max number allowed for the part in question
Max number of URAM that can be cascaded by the tool	Max number allowed for the part in question
Cascade DSP	Auto
Enable VHDL assert statements to be evaluated	Not checked

Table 4. Hardware modeling results.

Size	53×307	256×256	512×512
Blocks	4	16	16
Delay, ns	4.275	4.681	4.681
LUTs	35,334	142,848	580,608
Frequency, MHz	234	214	214
Absolute performance, Mbit/s	3,806,081	14,000,427	56,001,709
Performance on 100 MHz rate, Mbit/s	1,627,100	6,553,600	26,214,400

We can compare the results of the FPGA-based implementation with the results presented in [48,49].

7. CUDA Implementation

Hardware used:

1. CPU: Intel Core i5-4670
2. GPU: GeForce GTX 750 Ti

As noted above, one of the main advantages of the method is the possibility of parallelization of the calculations. Here, we will describe the implementation of the algorithm on CUDA (GPU) with the main differences from the CPU variant.

There are two approaches:

1. Sequential calculation of blocks, parallel calculation of cells in blocks.
   The cycle on the CPU is created in which each block will be calculated. To calculate a block on the GPU for each cell, a stream is created and the final state of the cell is calculated.
2. Parallel computation of blocks, parallel computation of cells in blocks.
   Each cell downstream will be run; in each stream, the block to which the cell belongs is determined, and then the final state of the cell is calculated.

The most difficult part in this step is to calculate the index of the cell in the block when it is circled clockwise (NESW). Since the program is executed in parallel, there is no sequential round-trip in a clockwise direction, and the index is calculated using a formula that uses the perimeter of the rectangle.

The difference between the three hardware-based methods of the described PRNG implementation can be seen in Figures 8 and 9 and Table 5 presents the comparison between the results of the tests of these three methods. The difference chart is presented in Figure 10.

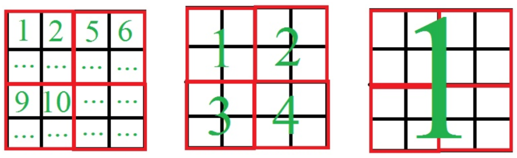
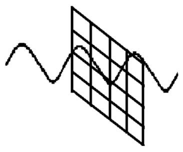
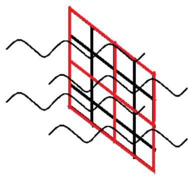


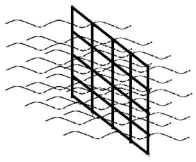
Figure 8. Calculations queue for CPU and GPU. **Left**: sequential calculations for both cells and blocks (CPU); **center**: parallel calculations for cells in each block (GPU1); **right**: parallel calculations for whole field (GPU2).



One stream for whole field (CPU)



The stream for each block (GPU1)



The stream for each cell (GPU2)

Figure 9. Using streams for calculations on CPU and GPU.

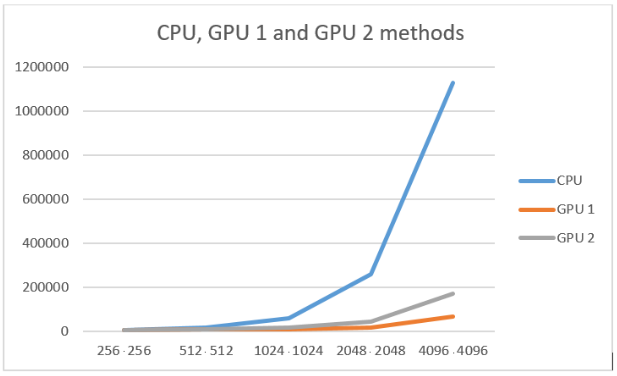


Figure 10. Number of cells (X) to execution time (Y) chart.

Table 5. Comparison between tests results of three hardware-based methods.

Size	Blocks	CPU (µs)	GPU1 (µs)	GPU2 (µs)
53 × 307	4	1639	1948	4584
256 × 256	16	3816	5813	5786
512 × 512	16	17,264	6792	7485
1024 × 1024	16	57,986	8182	14,583
2048 × 2048	16	257,453	16,887	43,932
4096 × 4096	16	1,128,957	64,226	171,278

After CUDA research, there were several tests with variable block size (Table 6).

Table 6. Block size variety output.

Field	Blocksize	Blocks	CPU (µs)	GPU1 (µs)	GPU2 (µs)
256 × 256	16 × 16	256	43,910	121,671	19,135
256 × 256	32 × 32	64	13,683	30,743	9567
256 × 256	64 × 64	16	6778	8157	7859
256 × 256	128 × 128	4	4873	2570	7001
512 × 512	32 × 32	256	53,994	125,554	52,134
512 × 512	64 × 64	64	25,109	36,101	18,919
512 × 512	128 × 128	16	17,327	8826	10,303
512 × 512	256 × 256	4	15,189	2958	8167
1024 × 1024	64 × 64	256	95,915	127,036	175,194
1024 × 1024	128 × 128	64	65,996	38,352	46,781
1024 × 1024	256 × 256	16	58,411	11,874	20,757
1024 × 1024	512 × 512	4	55,890	5414	12,556

It is obvious that all the advantages and disadvantages of CPU- and GPU-based implementations are concluded. However, it is not in such a way for GPU1 and GPU2 implementations difference.

The main differences between GPU1 and GPU2 implementations are that, in the GPU2 implementation, the calculation of blocks and cells occurs in parallel, and in GPU1—the cells in the block are calculated in parallel, and the blocks are sequential. In the second case, additional calculations are needed, such as the calculation of the cell belonging to the unit. The second implementation will give an increase in performance with a large number of blocks and a large volume of cells.

It can be seen from the tables, on small data, that two implementations on the GPU lose their pros to the CPU. This is because the amount of computation is so small that copying the device into memory and additional computations in each GPU stream take a lot of time for this amount of data.

However, on large amounts of data, implementation on GPU wins due to parallel computing. The use of additional calculations is due to the fact that they are performed in parallel and allow each cell and block to be calculated in parallel.

8. Discussion

It should be noted that, from the point of view of hardware implementation, both linear congruent and retarded Fibonacci generators are not very suitable [50–52]: they are not efficient in terms of microprocessors and time, when it is necessary to apply PRNG for distributed machines and parallel computing, for embedded testers, or for other applications at this level.

A third widely used type of generator is the so-called linear feedback generator (LFSR). Linear feedback shift registers are common among physicists and computer engineers. There are forms of LFSR that are suitable for hardware implementation.

However, it turns out that, compared to equivalent CA-based generators, they are less efficient; in addition, they are less applicable in terms of the possibility of implementation and debugging, although the area required for the CA cell is slightly larger than for the LFSR [38]. In [53], comparing LFSR and CA, was also illustrated, the benefits of using CA in PRNG were shown.

Moreover, different sequences generated by the same CA are much less correlated than similar sequences generated by LFSR. This means that CA-generated bit sequences can be used in parallel, which implies clear advantages in using them in applications.

PRNG based on CA outperforms similar algorithms in terms of performance characteristics, flexibility in development, scalability and parallelization properties.

9. Conclusions

This paper presents a pseudorandom generating algorithm on CA in a detailed form with new ideas to improve some features of an algorithm, which existed before. Testing results show that the described generator has good perspectives because of its effectiveness, simplicity, velocity, and improved unpredictability.

The proposed PRNG algorithm combines the advantages of the speed of cellular automata on a two-dimensional basis and ease of implementation, as a step-by-step series of Moore automata with initiated initial states. Compared to the classical algorithms generating pseudo-random numbers, the cellular automaton presented algorithm has such changes:

- application of several neighborhood templates;
- the use of several independent cellular automata;
- implementation of transitions of various cellular automata according to various sets of rules;
- expansion of rule sets based on the complementarity of the number of ones and zeros for a statistically normal distribution.

The changes mentioned helped our algorithm to obtain better results in some NIST tests compared to existing algorithms. The developed algorithm can be successfully used and implemented in many cryptographic primitives, given the ease of implementation, speed, and compliance with modern development models (parallelization of processes).

Future research will be focusing on an implementation presented generator in different cryptographic applications such as lightweight cryptography or coding theory.

Author Contributions: Conceptualization, A.L, D.M., P.L., D.B. and D.K.; methodology, A.L., D.M., P.L. and D.K.; software, D.B. and M.V.; validation, M.V., D.B. and A.L.; formal analysis, D.K. and A.L.; investigation, D.M., D.B. and M.V.; resources, P.L. and D.K.; data curation, D.B., D.K. and P.L.; writing—original draft preparation, D.M., D.K. and P.L.; writing—review and editing, D.K., A.L and P.L.; visualization, M.V. and D.M.; supervision, A.L.; project administration, D.K.; funding acquisition, A.L. and D.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CA	Cellular Automata
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FPGA	Field-Programmable Gate Array
HS	Homogeneous Structures
LCG	Linear Congruential Generator
LFSR	Linear Feedback Shift Register
NESW	North, East, South, West
NIST	National Institute of Standards and Technology
PRNG	Pseudorandom Number Generator
RNG	Random Number Generator

References

1. Genkin, D.; Shamir, A.; Tromer, E. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. *IACR Cryptol. Eprint Arch.* **2013**, *857*, 2013.
2. Genkin, D.; Pipman, I.; Tromer, E. Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. *J. Cryptogr. Eng.* **2014**, *5*, 242–260.
3. Genkin, D.; Pachmanov, L.; Pipman, I.; Tromer, E. Stealing Keys from PCs by Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. *IACR Cryptol. Eprint Arch.* **2015**, *170*, 2015.
4. Levina, A.; Borisenko, P.; Mostovoy, R.; Orsino, A.; Ometov, A.; Andreev, S. Mobile Social Networking under Side-Channel Attacks: Practical Security Challenges. *IEEE Access* **2017**, *5*, 2591–2601.
5. Levina, A.; Sleptsova, D.; Zaitsev, O. Side-channel attacks and machine learning approach. In Proceedings of the Conference of Open Innovation Association, FRUCT, St. Petersburg, Russia, 5–7 April 2016; pp. 181–186.
6. Levina, A.; Mostovoi, R.; Sleptsova, D.; Tsvetkov, L. Physical model of sensitive data leakage from PC-based cryptographic systems. *J. Cryptogr. Eng.* **2019**, *9*, 393–400. [CrossRef]
7. Sasaki, T.; Togo, H.; Tanidaa, J.; Ichiokab, Y. Stream cipher based on pseudo-random number generation using optical affine transformation. *Appl. Opt.* **2000**, *39*, 2340–2346. [CrossRef] [PubMed]
8. Biryukov, A.; Shamir, A. Cryptanalytic time/memory/data trade-offs for stream ciphers. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Singapore, 6–10 December 2000; Springer: Berlin/Heidelberg, Germany, 2000; pp. 1–13.
9. Cunsheng, D. *The Stability Theory of Stream Ciphers*; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2011; Volume 561.
10. Haahr, M. True Random Number Service. 1998. Available online: www.random.org (accessed on 19 November 2021).
11. Brent, R. Uniform Random Number Generators for Supercomputers. 1992. Available online: https://www.semanticscholar.org/paper/Uniform-random-number-generators-for-supercomputers-Brent/e67e46d2b5581c9d6300138155de3dc8197fd9bb (accessed on 23 January 2021).
12. Niels, F.; Bruce, S.; Tadayoshi, K. Chapter 9: Generating Randomness. In *Cryptography Engineering: Design Principles and Practical Applications*; Wiley Publishing, Inc.: Hoboken, NJ, USA, 2010; ISBN 978-0-470-47424-2.
13. Kelsey, J.; Schneier, B.; Wagner, D.; Hall, C. *Cryptanalytic Attacks on Pseudorandom Number Generators*; Fast Software Encryption; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1998; Volume 1372.
14. L'Ecuyer, P. Chapter 4: Random Number Generation. In *Springer Handbooks of Computational Statistics*; Springer: Berlin/Heidelberg, Germany, 2007.
15. Jenkins, R.J. ISAAC. In *Fast Software Encryption*; Gollmann, D., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1039.
16. Chen, H.; Laine, K.; Player, R. Simple Encrypted Arithmetic Library—SEAL v2.1. In Proceedings of the International Conference on Financial Cryptography and Data Security, Sliema, Malta, 3–7 April 2017.
17. Kircanski, A.; Youssef, A.M. *On the Sliding Property of SNOW 3G and SNOW 2.0*; Information Security; IET: London, UK, 2010; Volume 5, pp. 199–206.
18. Blum, L.; Blum, M.; Shub, M. A Simple Unpredictable Pseudo-Random Number Generator. *SIAM J. Comput.* **1986**, *15*, 364–383. [CrossRef]
19. Li, S.; Liu, Y.; Ren, F.; Yang, Z. Design of a high throughput pseudo-random number generator based on discrete hyper-chaotic system. *IEEE Trans. Circuits Syst. II Express Briefs* **2022**. [CrossRef]
20. Yu, F.; Zhang, Z.; Shen, H.; Huang, Y.; Cai, S.; Du, S. FPGA implementation and image encryption application of a new PRNG based on a memristive Hopfield neural network with a special activation gradient. *Chin. Phys. Soc. IOP Publ. Chin. Phys. B* **2022**, *31*, 020505. [CrossRef]
21. Yu, F.; Zhang, Z.; Shen, H.; Huang, Y.; Cai, S.; Jin, J.; Du, S. Design and FPGA Implementation of a Pseudo-random Number Generator Based on a Hopfield Neural Network Under Electromagnetic Radiation. *Front. Phys.* **2021**, *9*, 690651. [CrossRef]
22. Pereira, D.H. *Itamaracá: A Novel Simple Way to Generate Pseudo-Random Numbers*; Cambridge University: Cambridge, UK, 2022. [CrossRef]
23. Moysis, L.; Rajagopal, K.; Tutueva, A.V.; Volos, C.; Teka, B.; Butusov, D.N. Chaotic Path Planning for 3D Area Coverage Using a Pseudo-Random Bit Generator from a 1D Chaotic Map. *Mathematics* **2021**, *9*, 1821. [CrossRef]
24. Tutueva, A.V.; Karimov, T.I.; Moysis, L.; Nepomuceno, E.G.; Volos, C.; Butusov, D.N. Improving chaos-based pseudo-random generators in finite-precision arithmetic. *Nonlinear Dyn.* **2021**, *104*, 727–737. [CrossRef]
25. Palacios-Luengas, L.; Pichardo-Méndez, J.L.; Díaz-Méndez, J.A.; Rodríguez-Santos, F.; Vázquez-Medina, R. PRNG based on skew tent map. *Arab. J. Sci. Eng.* **2019**, *44*, 3817–3830. [CrossRef]
26. Podstrigaev, A.S.; Smolyakov, A.V.; Maslov, I.V. Probability of Pulse Overlap as a Quantitative Indicator of Signal Environment Complexity. *J. Russ. Univ. Radioelectron.* **2020**, *23*, 37–45. [CrossRef]
27. Datcu, O.; Macovei, C.; Hobincu, R. Chaos based cryptographic pseudo-random number generator template with dynamic state change. *Appl. Sci.* **2020**, *10*, 451. [CrossRef]

28. L'Ecuyer, P.; Nadeau-Chamard, O.; Chen, Y.-F.; Lebar, J. Multiple Streams with Recurrence-Based, Counter-Based, and Splittable Random Number Generators. In Proceedings of the 2021 Winter Simulation Conference, Phoenix, AZ, USA, 12–15 December 2021.
29. Lehmer, D.H. Mathematical methods in large-scale computing units. In Proceedings of the Second Symposium on Large-Scale Digital Calculating Machinery, Oak Ridge, TN, USA, April 1949; Harvard University Press: Cambridge, MA, USA, 1951; pp. 141–146.
30. Thomson, W.E. A Modified Congruence Method of Generating Pseudo-random Numbers. *Comput. J.* **1958**, *1*, 83. [CrossRef]
31. Rotenberg, A. A New Pseudo-Random Number Generator. *J. ACM* **1960**, *7*, 75–77. [CrossRef]
32. L'Ecuyer, P. History of uniform random number generation. In Proceedings of the WSC 2017—Winter Simulation Conference, Las Vegas, NV, USA, 3–6 December 2017.
33. Tomassini, M. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time;* Springer: Berlin/Heidelberg, Germany, 2005.
34. Kudryavtsev, V.B.; Podkolzin, A.S. Cellular automata. *Intellect. Syst.* **2006**, *4*, 657–692.
35. Moore, E.F. Gedanken-experiments on Sequential Machines. *Autom. Stud.* **1956**, *34*, 129–153.
36. Wolfram, S. Random sequence generation by cellular automata. *Adv. Appl. Math.* **1986**, *7*, 123–169. [CrossRef]
37. Ilachinski, A. *Cellular Automata: A Discrete Universe*; World Scientific: Singapore, 2001.
38. Tomassini, M.; Sipper, M.; Perrenoud, M. On the generation of high-quality random numbers by two-dimensional cellular automata. *IEEE Trans. Comput.* **2000**, *49*, 1146–1151.
39. Tomassini, M.; Perrenoud, M. Cryptography with cellular automata. *Appl. Soft Comput.* **2001**, *1*, 151–160. [CrossRef]
40. Weisstein, E.W. Von Neumann Neighborhood. MathWorld—A Wolfram Web Resource. Available online: http://mathworld.wolfram.com/vonNeumannNeighborhood.html (accessed on 31 July 2019).
41. Weisstein, E.W. Moore Neighborhood. MathWorld—A Wolfram Web Resource. Available online: http://mathworld.wolfram.com/MooreNeighborhood.html (accessed on 8 August 2019).
42. Dennunzio, A.; Formenti, E.; Provillard, J. Non-uniform cellular automata: Classes, dynamics, and decidability. *J. Inf. Comput.* **2012**, *215*, 32–46. [CrossRef]
43. Wolfram, S. Cellular Automat. *Los Alamos Sci.* **1983**, *9*, 2–21.
44. Bassham, L.; Rukhin, A.; Soto, J.; Nechvatal, J.; Smid, M.; Leigh, S.; Levenson, M.; Vangel, M.; Heckert, N.; Banks, D. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*; Special Publication (NIST SP); National Institute of Standards and Technology: Gaithersburg, MD, USA, 2010.
45. Zhukov, A.E. Cellular Automata in Cryptography. Part 2. *Voprosy kiberbezopasnosti* **2017**, *4*, 47–66. [CrossRef]
46. Rogawski, M. Hardware Evaluation of eSTREAM Candidates: Grain, Lex, Mickey128, Salsa20 and Trivium. The eSTREAM Project. 2007; 10p. Available online: https://www.ecrypt.eu.org/stream/papersdir/2007/025.pdf (accessed on 6 March 2022).
47. Gurkaynak, F.; Luethi, P.; Bernold, N.; Blattmann, R.; Goode, V.; Marghitola, M.; Kaeslin, H.; Felber, N.; Fichtner, W. Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt. The eSTREAM Project. 2006; 12p. Available online: https://www.ecrypt.eu.org/stream/papersdir/2006/015.pdf (accessed on 2 February 2022).
48. Bakiri, M.; Guyeux, C.; Couchot, J.; Marangio, L.; Galatolo, S. A Hardware and Secure Pseudorandom Generator for Constrained Devices. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3754–3765. [CrossRef]
49. Bakiri, M.; Couchot, J.; Guyeux, C. CIPRNG: A VLSI Family of Chaotic Iterations Post-Processings for $\mathbb{F}_2$—Linear Pseudorandom Number Generation Based on Zynq MPSoC. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2018**, *65*, 1628–1641. [CrossRef]
50. Gutierrez, J. Attacking the linear congruential generator on ellipticcurves via lattice techniques. *Cryptogr. Commun.* **2021**, *14*, 505–525. [CrossRef]
51. Nannipieri, P.; Di Matteo, S.; Baldanzi, L.; Crocetti, L.; Belli, J.; Fanucci, L.; Saponara, S. True Random Number Generator Based on Fibonacci-Galois Ring Oscillators for FPGA. *Appl. Sci.* **2021**, *11*, 3330. [CrossRef]
52. Badra, M.; Guillet, T.; Serhrouchni, A. Random values, nonce and challenges: Semantic meaning versus opaque and strings of data. In Proceedings of the 2009 IEEE 70th Vehicular Technology Conference Fall, Anchorage, AK, USA, 20–23 September 2009; pp. 1–5.
53. Sachin, D. Comparison of LFSR and CA for BIST. *Comput. Sci.* **2005**. Available online: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.146.4514 (accessed on 31 July 2019).