mathematics

MDPI

Article

Cryptographically Secured Pseudo-Random Number Generators: Analysis and Testing with NIST Statistical Test Suite

Elena Almaraz Luengo [1,*] and Javier Román Villaizán [2]

[1] Department of Statistics and Operational Research, Faculty of Mathematical Science, Complutense University of Madrid, 28040 Madrid, Spain
[2] Faculty of Mathematical Science, Complutense University of Madrid, 28040 Madrid, Spain; jaroman@ucm.es
* Correspondence: ealmaraz@ucm.es

Abstract: There are several areas of knowledge in which (pseudo-)random numbers are necessary, for example, in statistical–mathematical simulation or in cryptography and system security, among others. Depending on the area of application, it will be necessary that the sequences used meet certain requirements. In general, randomness and uniformity conditions are required in the generated sequences, which are checked with statistical tests, and conditions on sequence unpredictability if the application is in security. In the present work, a literature review on cryptographically secure pseudo-random number generators (CSPRNGs) is carried out, they are implemented, and a critical analysis of their statistical quality and computational efficiency is performed. For this purpose, different programming languages will be used, and the sequences obtained will be checked by means of the NIST Statistical Test Suite (NIST STS). In addition, a user's guide will be provided to allow the selection of one generator over another according to its statistical properties and computational implementation characteristics.

Keywords: cryptographically secured pseudo-random number generators; hypothesis testing; NIST STS; test battery; test suite

MSC: 65C10; 62G10

Citation: Almaraz Luengo, E.; Román Villaizán, J. Cryptographically Secured Pseudo-Random Number Generators: Analysis and Testing with NIST Statistical Test Suite. *Mathematics* **2023**, *11*, 4812. https://doi.org/10.3390/math11234812

Academic Editor: Antanas Cenys

Received: 28 October 2023
Revised: 21 November 2023
Accepted: 23 November 2023
Published: 28 November 2023

1. Introduction

The use of random sequence generators in various fields of knowledge such as information and communication technologies (see [1,2]), computer security (see [3–5]), mathematical simulation [6–8]), sampling ([9]), generation of random variates (see, for example, [10]), etc., is ubiquitous. They appear in telephone communication signals and GPSs. Most cryptographic protocols make use of random numbers as input at some points [11] or use them as a key in stream cipher systems [12].

However, the problem of obtaining random sequences is by no means trivial from a computational point of view. Since computers are deterministic, instruction-driven machines, it is difficult to generate algorithms that "produce" randomness. While performing operations can give the appearance of being random, there is a pre-determined pattern behind them. Therefore, we resort to physical phenomena that have intrinsic randomness to obtain "truly" random data: radioactive decay, thermal noise, atmospheric phenomena, etc.

Obtaining data from physical phenomena, however, is relatively expensive, so it is not possible to extract the required number of random numbers in a short time. Sometimes, either because computational capacity is quite low or because many numbers are needed in a very short time, as in stream ciphering, it is of interest to create fast and computationally simple generators that produce seemingly random sequences.

At this point, it is necessary to specify the concept of being random or having the appearance of being random. From a purely linguistic point of view, random is that which depends on chance. In a more formal context, a generally accepted definition

of a sequence of random numbers is Marsaglia's [13]: a sequence of random numbers is uniformly distributed over all possible values, each number in it being independent of those generated before it. Similarly, a random number generator can be defined as a system that creates random sequences, as defined above. However, due to the diversity of applications of random numbers, it will be necessary for the sequences to have some characteristics or others. In the case of applications in statistics, the requirements demanded will be different from those demanded in cryptography, for example, mainly because in the latter case the condition of unpredictability of the sequence is essential. In this paper, we will focus on the analysis of number generators that are suitable precisely for cryptographic applications.

The rest of the paper is organized as follows: in Section 2, a classification of random numbers is given, with a description of them. In Section 3, we will focus on pseudo-random numbers and their generators. In Section 4, a description of cryptographically secure pseudo-random number generators is given, and an in-depth analysis is performed. Section 5 focuses on the experimental design. A detailed explanation will be given of both the computational and statistical aspects that have arisen during the experimentation, which will allow conclusions to be drawn on the pros and cons of the analyzed generators. Finally, in Section 6, the conclusions of our study are given.

2. Classification of Random Numbers

A well-known classification of the different types of random numbers (see, for example, [14–16]) establishes three fundamental groups of numbers: the so-called "true" random numbers, pseudo-random numbers, and quasi-random numbers.

True Random Numbers are generated with an unpredictable input, using as a resource the entropy of a random phenomenon such as the computer clock or photoelectric fluctuations. As they are unpredictable since there is no rule that allows us to know the values that are generated through those already generated, they are very useful in encryption systems. Two factors are involved in the generation of the sequences of this kind of numbers: (i) an unpredictable resource in the sense of having high entropy. As a measure, min-entropy is used, defined as the maximum number k such that for each $x \in X$ $P[X = x] \leq 2^{-k}$ is verified, where X is the resource (min-entropy is a stricter measure than Shannon's entropy since the min-entropy of X is always less than or equal to Shannon's entropy for X [17]). This factor is critical, as it will determine the available entropy. Some resources may be biased, so it would be necessary to remove them or perform a post-processing step. (ii) A "harvesting" mechanism, which is a function that is applied on the high-entropy resource collecting the maximum possible entropy. The objective is in the second component since the aim is to find a mechanism that can be used with all types of high-entropy resources. Sometimes, it is necessary to introduce a post-processing component in the generation of these sequences to strengthen the algorithms, either by masking imperfections of the entropy resource or the mechanism or also to have a tolerance in contexts of changes and manipulation. When applying post-processing, it is necessary to verify that the statistical properties of the sequences have not been modified. The most commonly used post-processing techniques in practice are cryptographic hash functions, Von Neumann's corrector, extractor functions, and resilient functions. For a more detailed description of the post-processing techniques, see [18].

Pseudo-random numbers are generated through a mathematical algorithm that produces a periodic sequence in a deterministic way; this process is completely determined by the initial condition, called the seed. It is intended to resemble true random numbers. With a suitable formula and a satisfactorily chosen seed, a sequence of pseudo-random numbers can be obtained that can practically be regarded as truly random in terms of passing a series of statistical tests of randomness (test suites or test batteries). They are used in the field of statistics and cryptography. A paper that describes pseudo-random numbers and generation methods in-depth is [19].

Quasi-random numbers, also known as low-discrepancy points. The series of these numbers is not intended to be random but shares statistical properties with random se-

quences. They are generated in such a way as to cover the feasible region in an optimal way. They do not obtain numbers but a sequence of points in a desired dimension, satisfying an equidistributional criterion, and are used in the numerical evaluation of integrals. More formal definitions of these numbers can be found in [20].

In general, when working with sequences of (pseudo-)random numbers, a series of characteristics are sought, which can be grouped into four broad categories: structural, computational, statistical, and complexity [21]. The structural category refers to aspects such as the length of the period (a sequence $x_0, x_1, \ldots$ of elements of a nonempty set is periodic, with period T, if there exists a positive integer T such that $x_{n+T} = x_n$ for all $n \geq 0$) or the lattice structure desired to be as long as possible, while the computational category deals with aspects of computer implementation such as ease of code, computation time, or memory space required. The statistical category is concerned with the properties of the distribution and statistical independence. One possible formalization of the statistical properties of uniform random numbers is shown in Definition 1.

Definition 1. *A sequence $x_0, x_1, \ldots$ of numbers in $I = [0, 1]$ is completely uniformly distributed if for each integer $s \geq 1$, the sequence of points $(x_n, x_{n+1}, \ldots, x_{n+s-1}) \in I^s$, $n = 0, 1, \ldots$, is uniformly distributed in I^s.*

As for the complexity category, the definition of Chaitin's [22] for a finite sequence of random digits can be seen in Definition 2.

Definition 2. *If $b \geq 2$ is an integer and σ_N is a finite sequence of length N formed by the elements of $Z_b = \{0, 1, \ldots, b-1\}$, then its Kolmogorov's complexity is the minimum length of the program that generates the sequence σ_N on a Turing's machine.*

The sequence is considered random if it has the maximum Kolmogorov's complexity among those sequences of size N formed by the elements of Z_b. Although this definition is not very useful from a practical point of view, it is necessary that the sequences are sufficiently complex so that the algorithms cannot be discovered and the sequence of random numbers can be known, having dire consequences in applications such as cryptography.

There are two main types of generators (see, for example, [19,23]):

- Physical generators (true random number generators, TRNGs) are physical devices that use external sources to generate random numbers. The more widely generators used are based on electrical circuits equipped with a noise source that is amplified, sampled, and compared with a reference signal to produce sequences of bits. These random bits are joined together to form bytes, integers, or real numbers as required. The output sequences of TRNGs can be used directly as random sequences or can be used as input to a pseudo-random number generator.
- Arithmetic generators (pseudo-random number generators, PRNGs): these are deterministic algorithms that run on computers. There are two main sub-types, linear and nonlinear.

It is also possible to find the classification expressed in terms of (see [24]): deterministic (pseudo-random, PRNG) and truly non-deterministic (truly random, TRNG). The former is further divided into pure and hybrid, while the latter can be separated into physical (PTRNG) and non-physical (NPTRNG), each being either pure or hybrid. Hybrid generators are those that combine elements of the truly random and pseudo-random generators.

From a cryptographic point of view, it is essential that the generators used are secure; in this sense, the unpredictability property must be verified. This implies that the knowledge of any substring of a generated sequence does not imply that the complete sequence can be calculated or estimated and that the knowledge of any given internal state does not imply that the preceding or subsequent numbers can be calculated. In this sense, it is possible to find in the literature the name cryptographically secure pseudo-random number generator (CSPRNG). CSPRNGs are a special type of PRNG with the property of unpredictability.

This means that given n consecutive bits of the key, there is no algorithm in polynomial time that can predict the next bit with probability of success greater than 50%.

In this paper, we will focus on analyzing the class of cryptographically secure random number generators. Being a subgroup within the PRNGs, in the following section we will discuss the essential aspects of such generators.

3. Key Aspects of Pseudo-Random Number Generators

As mentioned above, pseudo-random numbers are characterized by the fact that they are generated based on a numerical algorithm and are not truly random, although they may appear to be so if the algorithm is not known. In particular, one can define random number generation algorithms as shown in Definition 3 (see [13]).

Definition 3. *Given a finite set X, a function $f : X \to X$ and an initial value called the seed (the seed can be obtained by a truly random number generator, or be proposed by the user; in any case, it is sought to be unpredictable). $x \in X$, the generated sequence by a generation algorithm is:*

$$x, f(x), f^2(x), f^3(x), \ldots \tag{1}$$

where $f^2(x) = f(f(x))$ and $f^n(x) = f(f^{n-1}(x))$, $\forall n \geq 3$.

The desirable properties required of pseudo-random numbers are having a sufficiently long period, having no structure, having good statistical properties, and computational efficiency.

Generally, a distinction is made between uniform and non-uniform pseudo-random numbers. A paper that surveys the methods used for the generation of pseudo-random numbers is [19]. Examples of the best-known methods include: congruential generators, lagged Fibonacci generators, and linear-feedback shift register (LFSR), among others.

As for non-uniform pseudo-random numbers, they are generally generated from a uniform pseudo-random number and subsequently transformed into the non-uniform target distribution. Among the best-known transformation methods are the inversion method, the rejection method, the composition method, or the ratio of uniforms method (ROU method), among others.

Once the (pseudo-)random values have been generated, they must be tested. Some of the best-known tests are the runs test, the gap test, the Kolmogorov–Smirnov test, etc.

Usually, the tests are grouped into sets of tests called batteries or suites, and some of the best-known are Diehard [25], Dieharder [26], NIST SP 800-22 [11], Practrand [27], ENT [28], FIPS 140-2 [29], etc. Modules validated as FIPS 140-2 compliant will continue to be accepted by federal agencies in the United States and Canada for the protection of sensitive information or designated information until 21st September 2026. The new version, FIPS 140-3 [30] (final draft: https://csrc.nist.gov/publications/detail/fips/140/3/final, accessed on 1 January 2022), does not include hypothesis tests, and therefore FIPS 140-2 is still currently in use (although there are some linear interdependencies between some of the tests that make it up; see [31]). ENT battery allows the quality of the PRNGs to be checked by comparing their output with that of a TRNG. However, some vulnerabilities have recently been discovered in its design, and therefore we will not use it in our empirical tests (see [32]). The most widely used standard is the NIST SP 800-22 battery, and that is why in the present work we make use of its tests, although it is true that many of them can be found in other suites, such as Dieharder and in particular in its version for R, RDieharder. An article that studies in detail the tests that are included in the most used batteries in the literature is [33]. In fact, there is currently a line of research that focuses on the development of new tests and batteries. Some of the most recent works in this direction are [34–37], among others. On the other hand, there are also various hardware-level tests that consist of physically examining the hardware used for generation to check if the generator is working properly and that there are no vulnerabilities or weak points that could compromise security [35]. An interesting paper that analyzes the nomenclature

of random number generators, the available test suites, and the methodology to test a generator implementation is [38].

In general, some PRNGs are not suitable for cryptographic purposes, while there are designs that are theoretically proven to be secure ([39–41]). In practice, however, most of the generators implemented in operating systems and cryptographic libraries are not based on the main security models and present weaknesses that make them vulnerable against known attacks. Attacks on pseudo-random generators can be of three types: direct cryptanalytic attacks, entry-based attacks, and state compromise extension attacks (further subdivided into: backtracking, permanent compromise, and meet-in-the-middle attacks). For more details about these attacks, see [42].

Among the most common failures in pseudo-random generators that allow external attacks are [43] entropy overestimation and guessable starting points, incorrect handling of keys and seeds, implementation errors, cryptographic attacks on generation mechanisms, side-channel attacks, and chosen-input attacks.

4. Description of Cryptographically Secure Pseudo-Random Number Generators

Having defined the different types of existing random number generators, we now analyze those that can be used in the context of cryptography. In this field, it is necessary that a number cannot be predicted before it is generated. For example, if a random number in the range $0, \ldots, 2^n - 1$ is to be generated, an observer outside the generation may not be able to predict the number with a probability greater than $1/2^n$. Similarly, if a sequence with m random numbers has been generated, an observer who knows $m - 1$ of them cannot predict the m-th element of the sequence with a probability greater than $1/2^n$.

TRNGs can be used in cryptography with very good results; however, due to their high computational cost, it is necessary to look for computationally simpler alternatives. An exhaustive analysis of the application of these generators in cryptography can be found in [18].

The aim of this paper is the analysis of pseudo-random generators in cryptography, which are less expensive than the truly random ones. A simple definition of a cryptographic pseudo-random number generator could be the following: it is a cryptographic mechanism to process an unpredictable input by generating pseudo-random outputs. If it is well defined, implemented, and used properly, even an attacker with a huge amount of resources would not be able to distinguish between the generated output and a random sequence of bits.

The first cryptographically secure pseudo-random number generator was proposed by Shamir [44], who is one of the inventors of RSA [45]. In this case, the integer factorization problem is used as an essential element to ensure security. As for the design, it uses modular exponentiation of large numbers, which makes it very slow and, in sequence, not very useful from a computational point of view. An improved version of the RSA generator was proposed in [46]. This generator generates $\lfloor N(1 - 2/e) \rfloor$ bits per exponential by e, where e is encryption key for RSA. However, each exponentiation requires one modular square per bit. In [47], the BBS (Blum Blum Shub) generator is introduced, which requires only one modular square per bit instead of a modular exponentiation. It works with the concept of quadratic residue one-way function, and its security also depends on the fact of the intractability of the integer factorization of modulus N. In [48], a generation algorithm based on the transformation of sequences obtained with linear congruential generators, multiple recursive generators with large orders, fast multiple recursive generators, and dx generators ([49]) is proposed. The mechanism used makes it difficult to predict the sequence of the generator even if the partial sequence or parameters are known or there is knowledge of the algorithm used in the transformation of the generators. Subsequently, other models have been proposed, as in, for example, [50], where a CSPRNG based on a hybrid methodology including chaotic additional input is defined; in [51], where a CSPRNG based on controlling the distribution of generated random numbers with the chaotic henon congruential generator is proposed; and in [52,53], where a CSPRNG based

on the permutation of the internal state of the PRNG is proposed. The results are checked with Dieharder.

In the following, the most important generators used in practice, which can be used individually or in combination, will be discussed in detail. Their characteristics, mathematical foundations, and implementation will be presented, and then a critical analysis of them will be made, establishing the pros and cons of each one of them, making a comparison between them and a statistical validation by means of the suite of statistical tests provided by NIST.

4.1. Blum–Blum–Shub (BBS) Generator

It was proposed by L. Blum, M. Blum, and M. Shub in 1986 [47]. In order to define this generator, it is necessary to use the following notation (see [54]):

- $s \in_R S$ states that s has been randomly chosen from the set S.
- N is a Blum integer, (i.e., $N = pq$, where p and q are prime and verify $p \equiv q \equiv 3 \mod 4$).
- n is the size in bits of N.
- $\mathbb{Z}_N(+1)$ is the set of Jacobi's integers symbol +1 modulus N (The Jacobi's symbol is an arithmetic function that takes two arguments and returns an integer value from the interval $[-1,1]$. In [55], a more detailed definition and the way to obtain these integers can be seen).
- $\Lambda_N = \mathbb{Z}_N(+1) \cap \left(0, \frac{N}{2}\right)$.
- $\mid y \mid_N = y \mod N \in [0, N), \forall y \in \mathbb{Z}$.
- $y \underline{\mod} N \in \left(-\frac{N}{2}, \frac{N}{2}\right)$ denotes the smallest residue in absolute value of y modulus N.
- $l_i(y)$ denotes the i-th least significant bit of y, $i = 1, 2, \ldots$.
- $E_N(y) = \mid y^2 \underline{\mod} N \mid$, referred to as the absolute of Rabin's function (Rabin's function [56] is a public-key cryptosystem whose decryption is equivalent to factorization).

Definition 4. *Let $N = pq$ be the product of two prime numbers congruent to $3 \mod 4$ and let x_0 be the seed, with $x_0 \in_R \mathbb{Z}_N(+1) \mod N$. The sequence obtained with the BBS pseudo-random generator is the bit sequence $b_0, b_1, \ldots$ generated through the following steps:*

1. *Update x_i as follows: $x_{i+1} = E_N(x_i)$, where $E_N(x) = x^2 \mod N$.*
2. *Extract the bit $b_i = l_i(x_i)$.*

An alternative definition is given by Sidorenko and Schoenmakers [54]:

Definition 5. *Let k, j be two positive integers and let $x_1 \in_R \Lambda_N$ be the seed. We consider a deterministic algorithm that transforms the seed into a binary sequence of length $M = jk$ by repeating the following steps for $i = 1, \ldots, k$:*

1. *For $r = 1, \ldots, j$, consider $b_{(i-1)j+r} = l_{j-r+1}(x_i)$.*
2. *Update x_i as follows: $x_{i+1} = E_N(x_i)$.*

This algorithm is the BBS generator with j output bits per iteration.

An interesting property of the BBS generator is the possibility to compute every value x_i directly:

$$x_i = \left(x_0^{2^i \mod (p-1)(q-1)}\right) \mod M \tag{2}$$

Intuitively, the performance of the BBS generator can be improved in two ways, either by increasing the number of bits per iteration or by using a smaller module size. However, these improvements could cause the security of the generator to be weakened, which is why the parameters to be used as well as the optimal value of the number of extracted bits, j per iteration, must be carefully analyzed. The security of the BBS generator is proved by

reduction (security reduction is a type of mathematical proof to show whether a protocol is secure in the sense of being at least as hard to break as another problem that is believed to be complicated [57]). If the generator is insecure, then there exists an algorithm B that, given $E_N(x)$ for some $x \in \Lambda_N$ and the $j-1$-th least significant bits of x, can guess the j-th; that is, it can factorize the modulus and put the security of the generator at risk. For asymptotic security ($j > 1$), the reduction is sought to be in polynomial time, while for concrete security ($j = 1$), the reduction is desired to be as tight as possible.

In the case of $j = 1$, it has been widely studied in the literature by several authors, highlighting Fischlin and Schnorr's work [58] achieving a tight reduction in an efficient generator. This reduction is achieved by modifying the Rabin function used in the generator.

For the case $j > 1$, asymptotic security has also been analyzed in [59]. This paper shows that the BBS generator is secure if $j = O(\log \log N)$. Sidorenko and Schoenmakers [54] obtained a more efficient result than Vazirani and Vazirani. An auxiliary definition is necessary to understand the result.

Definition 6. *A pseudo-random generator is (T_A, ϵ)-secure if it passes all statistical tests with a tolerance ϵ in at most T_A time.*

Sidorenko and Schoemakers proved that the pseudo-random BBS generator is (T_A, ϵ)-is safe if:

$$T_A \leq \frac{L(n)}{36n(\log_2 n)\delta^{-2}} - 2^{2j+9} n \delta^{-4} \tag{3}$$

where $\delta = (2^j - 1)^{-1} M^{-1} \epsilon$, and $L(n)$ is the number of clock cycles required to factor an n-bit integer; this value is estimated to be $L(n) \approx 2.8 \cdot 10^{-3} \cdot \exp(1.9229(n \ln 2)^{1/3} \ln(n \ln 2))^{2/3})$.

4.2. LFSR Generators

This type of generators is widely used due to their simplicity, easy and efficient computational implementation, and good statistical properties. However, some of these characteristics make them vulnerable to cryptographic attacks. In this subsection, we analyze some of the generators proposed in the literature that show good results against attacks. An LFSR is a shift register in which the input is a bit that comes from applying a linear transformation to a previous state.

The first generator to be studied was proposed by D. Coopersmith, H. Krawczyk, and Y. Mansour [60], known as the *Shrinkage generator.* This method uses two pseudo-random bit resources to construct a third one with a better quality than the original resources, understanding quality as the difficulty of predicting the sequence. The generation process is as follows: let $\{a_i\}_{i=0}^{\infty}$ be the first sequence and $\{s_i\}_{i=0}^{\infty}$ the second; the third sequence $\{z_i\}_{i=0}^{\infty}$ is constructed with those bits a_i such that the corresponding s_i is a 1, and the remaining bits of the first sequence are discarded. Mathematically, $\forall k = 0, 1, \ldots, z_k = a_{i_k}$, where i_k is the position of the k-th 1 of the sequence $s_0, s_1, \ldots$, we denote A to the first sequence, S to the second, and Z to the third, which is the one returned by the generator.

In [60], the two bit sequences are generated with LFSR using as feedback function the linear combination in $\mathbb{Z}_2$, being able to be fixed or variable. This generator has the necessary statistical properties to be used in cryptography in a secure way.

Different attacks on this generator have been raised to check its security; if the linear combination function of A and S is known, it would be possible to make an exhaustive search for the seed of S in order to try to recover the seed of A through the resolution of a linear system of dimension $n/2$, being n the length of the sequence A. Once the seeds are obtained, it would be possible to discover the whole generation process. This process would have a complexity of $O(2^m \cdot n^3)$ with n being the length of A and m the length of the sequence S. Whereas if the linear combination is secret, which is more normal, the above procedure no longer works; being necessary to try multiple combinations of seed and linear combination function for S and with part of the Z sequence, it might be possible

to generate the complete sequence. In this case, the complexity is higher, $O(2^{2m} \cdot n^2 \cdot m)$, where n and m are the lengths of the sequences.

Another type of attack that can be performed on this generator is based on its linear complexity, but it requires knowledge of a large number of bits of the final sequence for the attack to be effective, namely $2^{m-2} \cdot n$, which makes it practically impossible to perform. The complexity of this attack would be $O(2^m \cdot n^2)$.

A modification of this Shrinkage generator is the one proposed by W. Meier and O. Staffelbach [61], known as the *Self-Shrinkage generator*. Unlike the Shrinkage generator that used two LFSRs to generate the sequence, only one LFSR is used in this proposal. The process of obtaining the sequence is as follows: we have a sequence $a = (a_0, a_1, \dots)$ generated from an LFSR, and it is considered a bit pair sequence $((a_0, a_1), (a_2, a_3), \dots)$. If the pair $(a_{2i}, a_)$ is equal to $(1, 0)$ or $(1, 1)$, then a pseudo-random bit 0 or 1, respectively, occurs. In case the bit pair was $(0, 1)$ or $(0, 0)$, they would be discarded. The final sequence will be $s = (s_0, s_1, \dots)$.

This Self-Shrinkage generator can be implemented as the generator with two LFSRs in the following way: starting from the sequence $a = (a_0, a_1, \dots)$, the sequences $(a_0, a_2, \dots)$ and $(a_1, a_3, \dots)$ are formed, and the methodology of the Shrinkage generator can be applied, obtaining the same sequence by applying the same feedback function. It is also possible to make the inverse implementation, starting from the sequences generated by two LFSRs, $(b_0, b_1, \dots)$ and $(c_0, c_1, \dots)$, and the sequence $a = (c_0, b_0, c_1, b_1, \dots)$ can be constructed and the Self-Shrinkage generator can be used. Different cryptographic attacks have also been executed on this generator; starting from a known subsequence of s, one tries to reconstruct the entire sequence from the different possible combinations of the sequence a. In this case, if the feedback function is known, the complexity of this process would be $O(2^{0.75N})$, where N is the length of the sequence a generated with an LFSR, while if the feedback function is unknown, the complexity increases to $O(2^{1.75N})$.

Another interesting LFSR-based generator is the one proposed by C. G. Günther [62], known as the *Alternating Step generator (ASG)*, consisting of three LFSR sub-generators, K, M, and $\bar{M}$, which are interconnected. The output of generator K controls the clock of the other two; i.e., the output of generators M and $\bar{M}$ is repeated whenever K produces a 1 or a 0, respectively. This generator can be described as follows: let $\kappa, \mu, \bar{\mu}$ be the sequences generated by $K, M, \bar{M}$, and let $f_t = \sum_{s=0}^{t-1} \kappa_s$ and $\bar{f}_t = t - f_t$, and then the output ω_t is described by:

$$\omega_t = \mu_{f_t} \oplus \bar{\mu}_{\bar{f}_t} \tag{4}$$

This generator passes all the necessary tests to be used in cryptography and also allows a cascade structure in which each generator can be an ASG, which makes it more secure and efficient.

In recent years, different generators based on LFSRs have been proposed, such as the one proposed by J. Melià-Seguí, J. García-Alfaro, and J. Herrera-Joancomartí [63]. In this generator, a physical TRNG is combined with an LFSR. It consists of four blocks: an LFSR with a multi-polynomial architecture, a logic decoder, a polynomial selector, and the TRNG. Basically, the generator works as follows: the output of the TRNG feeds the decoder that drives the polynomial selector, which rotates depending on the value of the TRNG, thus avoiding the linearity that the LFSR may have. Another interesting generator is the one proposed by H. Zhang, Y. Wang, B. Wang, and X. Wu [64], which combines LFSR generators together with genetic algorithms with the objective of having nonlinearity in the generator, improving the period of the sequence and its efficiency.

In [65], a thermal noise-based oscillator physical TRNG is combined with an LFSR, such that each cycle of the LFSR is modified by an XOR operation involving the TRNG. The introduction of this operation makes the output sequence of the LFSR unpredictable and unrepeatable, making it a secure generator for cryptography. The authors also propose several methods to reduce the computational cost of the TRNG.

4.3. /dev/random Generator

This generator was implemented by Theodore T'so in the Linux operating system in 1994, and since then, it has been the pseudo-random number generator of Linux. This generator uses the external entropy of different resources, such as user input or unpredictable elements. Once this entropy is collected, it is used to generate random bits.

The structure of the generator consists of two sets (*pools*) (see [66]); the first one $\mathcal{P}_1$ is used to store the entropy collected from the external events $\mathcal{E}$, while the second pool $\mathcal{P}_2$ is used to obtain the random bits. In addition, there are two functions that interact with the pools, the mixing function m and the generating function gen. The former mixes the input to the pool, while the latter handles the generation. Entropy extraction occurs in two steps. First, the extractor function $e : \mathcal{E} \to \mathcal{D}$ converts the events of $\mathcal{E}$ into data from the pool $\mathcal{D}$ with the appropriate format. Subsequently, these data are combined into the pool $\mathcal{P}_1$ by the mixing function m, being of the hash type. To generate the random bits from $\mathcal{P}_2$, the gen function is applied. If $\mathcal{P}_2$ contains sufficient entropy, then the bits are generated using only $\mathcal{P}_2$. In the case of insufficient entropy, bits are shifted from $\mathcal{P}_1$ to $\mathcal{P}_2$ to increase the entropy. The generator checks that the amount of entropy in $\mathcal{P}_1$ is sufficient for generation but does not check in $\mathcal{P}_2$. This shift is performed through the function $s = gen \circ m$. First, bits are generated from $\mathcal{P}_1$ and combined with the m function in $\mathcal{P}_2$.

The transition function T is responsible for processing the new input, moving the information from the first to the second pool and mixing $\mathcal{P}_2$ after generation. The latter is performed by the t function: $\mathcal{P}_2 \to \mathcal{P}_2$, being of the hash type. So, the function T is formed by $e : \mathcal{E} \to \mathcal{D}$, $m : \mathcal{D} \to \mathcal{P}_1$, $s : \mathcal{P}_1 \to \mathcal{P}_2$, and $t : \mathcal{P}_2 \to \mathcal{P}_2$.

Regarding the security of this generator, the partitioning into two pools prevents iterative attacks since the input does not have a direct influence on the generator output. In addition, by using information from the system itself, input attacks are complicated to carry out. Employing hash functions in both the mix for $\mathcal{P}_1$ and the post-generation mix makes the generator not vulnerable to the various direct attacks. This generator is very dependent on the entropy used; if a large amount of unpredictable resources is available, good pseudo-random numbers will be generated, with good properties to be applied in cryptography and passing the randomness tests. However, if few entropy resources are available, the generator will slow down, and the generated numbers will be more vulnerable to different cryptographic attacks.

A problem with this generator is the speed; if a large set of pseudo-random numbers is needed, its use is not recommended, being suitable for short sequences, as for example, for obtaining the key in encryption algorithms [66].

4.4. AES Generator

The AES (Advanced Encryption Standard) generator is a block cipher scheme that can be used to encrypt or decrypt information. It was announced by the National Institute of Standards and Technology (NIST) in 2001 [67], representing limited instances of the Rijndael algorithm proposed by Daemen and Rijmen [68]. It is currently one of the most widely used algorithms in the field of symmetric cryptography.

The AES algorithm can be used in two different ways in the generation of pseudo-random numbers, the counter mode and the PRNG mode. In the former, a 128-bit $\mathcal{C}$ counter is used, which starts at c_0 and generates the sequence $\{c_i\}$, where $c_i = c_{i-1} + 1 \mod 2^{128}$. The output of this generator at time i is $E_{\mathcal{K}}(c_i)$, where $E_{\mathcal{K}}(x)$ represents the encryption of x using AES with the key $\mathcal{K}$. In the second mode, the AES algorithm is applied iteratively on the internal state of the generator. The sequence of pseudo-random numbers $\{x_i\}$ is produced successively, where $x_i = E_{\mathcal{K}}(x_{i-1})$, and the output of the generator is x_i itself.

Regarding the AES encryption algorithm itself, a two-dimensional array of bytes (a byte consists of 8 bits) called state is used, which consists of four rows of bytes, where each row contains Nb bytes, where Nb is the length of the input sequence divided by 32. In the array state, we denote by s each of the bytes with its two indices, where the row is

represented by r in the range $0 \leq r < 4$ and the columns by c with $0 \leq c < Nb$, so that each byte is represented as $s_{r,c}$.

For the AES algorithm, it is usual to use 128-bit sequences, although it is also possible to work with 192 and 256 bits. The encryptor scheme is as follows: first, the information of the input bytes is copied to the status array, then a series of operations are performed, which are detailed below, to finally generate the output bytes.

To copy the information from the input array to the status array, this scheme is followed: $s[r,c] = in[r+4c]$ with $0 \leq r < 4$ and $0 \leq c < Nb$, while for passing the array state to the output, the following is performed: $out[r+4c] = s[r,c]$ with $0 \leq r < 4$ and $0 \leq c < Nb$. (see [69]).

The operations performed on the array state to obtain the output of the encryptor are as follows:

- **SubBytes transformation:** it is a byte substitution using a two-dimensional, nonlinear, invertible matrix, called S-box. The construction of the matrix is made in two steps:
  1. The inverse of each byte is obtained, the element $\{00\}$ is assigned to itself.
  2. The following transformation is applied in $\mathbb{Z}_2$:

$$b_i^{'} = b_i \oplus b_{(i+4) \mod 8} \oplus b_{(i+5) \mod 8} \oplus b_{(i+6) \mod 8} \oplus b_{(i+7) \mod 8} \oplus c_i \quad (5)$$

  with $0 \leq i < 8$, where b_i represents the i-th bit of the byte being transformed, $b_i^{'}$ is the i-th bit of the transformed byte, c_i is the i-th bit of the byte $d = \{63\}$, and $\oplus$ is the XOR operator.

  Once the matrix is obtained, the byte of the status array is replaced by the corresponding byte in the S-box matrix.
- **ShiftRows transformation:** the last three status rows are moved, and the first row remains unchanged. The transformation is:

$$s_{r,c}^{'} = s_{r,(c+\ shift(r,Nb))\ mod\ Nb)} \text{ for } 0 < r < 4 \text{ and } 0 \leq c < Nb \quad (6)$$

  where $shift(r, Nb)$ depends on the file number, r, for the case $Nb = 4$:

$$shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3 \quad (7)$$

- **MixColumns transformation:** in this step, the state columns are multiplied by an invertible polynomial $a(x)$ of the form:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \quad (8)$$

  Specifically, the polynomial used in the AES algorithm is:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \quad (9)$$

- **AddRoundKey transformation:** a RoundKey is added to the state. RoundKeys are extracted from the $\mathcal{K}$ key and stored in a linear array of dimension (4×11). The first four words (a word is a group of 32 bits that are treated as a single element or as a 4-byte array) of the array are equal to that of the key, and the remaining ones are generated by the S-box matrix and multiplying by x^{i-1}, where $i^i \geq 1$ and $x = \{02\}$.

  A more detailed explanation of the processes can be found in [67,68].

The AES algorithm was developed to be secure against cryptographic attacks, so both generation modes will be secure as well. Part of the security of this algorithm is based on fuzziness and diffusion, which make it virtually impossible to guess the internal state of the generator from the output or to guess the output if part of the internal state is known to the attacker. If the seed of the algorithm is modified from a pool, and the attacker could

introduce some regularity into the pool through an input attack, this would have no effect on the structure of the output, and the generated numbers would be safe.

One of the weaknesses of this generator occurs when the internal state is compromised, which can make it possible to calculate past and future numbers. This makes the PRNG mode less suitable for cryptographic applications since by using part of the internal state in the process, it becomes weaker to attacks. However, the counter mode has a higher strength to these attacks and would be a good choice in these applications.

Both generation modes pass randomness tests. The advantage of AES generators lies in the combination of speed with cryptographic security. If you can couple the counter mode together with a random resource and a good key update mechanism, you will have a fast and highly secure generator for cryptographic applications.

4.5. Yarrow Generator

Proposed by J. Kelsey, B. Schneier, and N. Ferguson [43], it has four major components that attempt to stand alone: the entropy accumulator (collects samples of entropy resources and stores them in a pool), the reseed mechanism (periodically the key is modified with a new resource from the pool), the generator mechanism (generates outputs from seed), and the reseed control (determines when the generator key needs to be updated).

In a more detailed way, the entropy accumulator is the process in which the PRNG obtains the new internal state, being non-guessable. This step is critical both at the time of generator initialization and seed update. To avoid iterative attacks and to update the generator regularly, it is critical that the amount of entropy accumulated is correctly estimated. The entropy accumulation mechanism must also resist chosen input attacks. In the Yarrow generator, the entropy of the samples is collected in two pools, each in a hashed context. The fast pool provides the key updates, seeking to make the keys as short-lived as possible with accurate entropy estimates, while the slow pool provides rare, but conservative, key updates, so that you always have a secure key update. As for the key update mechanism, in the Yarrow generator a new key K is generated from the pool entropy. The steps of this mechanism are:

1. The entropy accumulator computes the hash function of the inputs to the fast pool; let v_0 be this result.
2. Determine $v_i := h(v_{i-1} \mid v_0 \mid i)$ for $i = 1, \ldots, t$, where h is the hash function.
3. Update the key: $K \longleftarrow h'(h(v_{p_t} \mid K), k)$, where $h'(m, k) :=$ first k bits of $(s_0 \mid s_1 \mid \ldots)$.
4. $C \longleftarrow E_K(0)$ is updated, where E_K is a triple-DES (data encryption standard) algorithm (algorithms are methods for encrypting information, and the triple-DES variant is more secure to attacks). A construction of these algorithms and how they differ from the basic DES can be seen in [70]).

The generating mechanism works as follows: an n-bit counter, C, is available; to generate the next output block, C is incremented and the block is encrypted with the triple-DES algorithm using the K key. To generate the next block, the values are updated as follows (see [43]):

$$\begin{aligned} C &\leftarrow (C + 1) \mod 2^n \\ R &\leftarrow E_K(C) \end{aligned} \tag{10}$$

where R is the next output block and K is the key.

Finally, the update control occurs automatically. The fast pool is used to update whenever any of its sources has an entropy estimate above some threshold value; in the Yarrow generator this threshold is 100 bits. The slow pool is used to update whenever at least two of its sources have entropy estimates above some other threshold value; in this case, this threshold is 160 bits.

This generator is secure against iterative attacks since it has pool separation as well as key update, which means that the input does not have a direct influence on the generator output. In addition, the slow pool prevents damage caused by entropy overestimation,

protecting against cryptographic attacks. Due to the generator mechanism, the number of bits of the output that can be known from a backtracking attack is limited.

4.6. Fortuna Generator

Proposed by B. Schneier and N. Ferguson [71] as an improvement to Yarrow generator, this generator solves the problem of having to define entropy estimators since it dispenses with them.

The generator consists of three parts: the first part is a generator, which takes a seed with a prefixed size giving as output an arbitrary amount of pseudo-random data. Secondly, there is the accumulator, which is responsible for collecting and pooling the entropy of various resources, occasionally also performing an update of the generator seed. Finally, the seed control file ensures that the generator can output good pseudo-random numbers.

The generator part uses the AES encryption algorithm in counter mode with a 256-bit encryption key and a 128-bit counter. The key and counter form the secret internal state of the generator. After each generation of pseudo-random data, a new 256-bit block is generated, which will serve as the key for the next generation, forgetting the one used. This eliminates any possibility of information leakage about previous generations since it would be impossible for an attacker to obtain the previous outputs despite knowing the internal state of the generator because the key would be different. The operations performed in this generator are:

- **Initialization:** both the key and the counter are set to 0, indicating that the generator has not yet been updated.
- **Update:** in this operation, the internal state is updated with a new string. A hash function is used to update the key, performing an exhaustive mix of the input string together with the existing key. The counter is also incremented by one unit, in this case a 16-byte integer.
- **Block generation:** a number of blocks with random output are generated using AES encryption with the key and the counter. This operation has an initial condition in which it is determined if the counter is non-zero to use the encryption algorithm. The output of this part is 16-byte blocks.
- **Generate random data:** a pseudo-random byte string with the length required by the user is obtained. To reduce the statistical deviation with respect to a truly random output, the length of the output string is limited to 2^{20}. Once the output is obtained, the key is modified with the block generating operation, and a 32-byte key is obtained.

In the second part of this generator, it is necessary to use different entropy resources, such as mouse movements, number of clicks, or internal computer data, such as memory or clock. These resources must be unpredictable to prevent the generator from being vulnerable to attacks.

To update the generator, it is necessary to group the resources into a pool large enough so that the attacker cannot enumerate the possible values in the pool. Unlike the Yarrow generator, where entropy estimators and various heuristic rules are used, in the Fortuna generator, this is solved in a simpler way.

There are 32 pools: P_i, $i = 0, \ldots, 31$, and each one contains a number of bytes coming from the entropy resources; this distribution of events is performed cyclically. An update of the generator is performed each time the pool P_0 has a sufficient amount of information. These updates are numbered $1, 2, 3, \ldots$. Depending on the update number, r, one or more pools are included in the update. To determine which pools are included, the following rule is followed: pool P_i is included in the update if 2^i is a divisor of r. So P_0 is always included, P_1 is included in even updates, P_2 is included every fourth update, and so on with the rest of the pools. After having been employed in an update, the corresponding pool is emptied. This technique makes the generator more secure against attacks since in the case where the attacker can know a significant amount of the P_0 pool, he will be able to reconstruct the new state of the generator. However, when P_1 is used in the update, it will

contain twice as much unpredictable information for the attacker, and P_2 will contain four times as much unpredictable information. So, no matter how many false random events the attacker creates or how many events he knows about, if there is at least one source of random events that he cannot predict, there will always be a set that gathers enough entropy to defeat him.

The third part of the generator, the seed control file, serves as data storage with high entropy; specifically, it consists of 64 bytes of data. This preserves the internal state of the generator, ensuring that after restarting the generator, good pseudo-random numbers are produced. This file would only be available to the generator, and once it is used, it is rewritten with new data.

Despite the simplicity of this idea, the implementation is complicated and highly dependent on the programming environment used. A more detailed explanation of the problems associated with this part of the generator, as well as the other parts of the generator, can be found in [71].

4.7. Trifork Generator

Proposed by A. B. Orúe, F. Montoya, and L. Hernández [72], it is a method based on the combination of three sequences generated from three coupled, delayed Fibonacci generators, with all of them being mutually perturbed. The perturbed, delayed Fibonacci generator is a modification of the conventional method. This transformation consists of perturbing the smallest and largest bit of the samples, defined by:

$$\begin{aligned} x_n &= ((x_{n-r} \oplus x'_{n-s}) + ((x_{n-s} \oplus x'_{n-r})) \mod m \\ x'_{n-s} &= (x_{n-s} \gg d) \\ x'_{n-r} &= (x_{n-r} \ll d) \end{aligned} \tag{11}$$

where r and s are the delays, N is the size, m is the basis verifying $m = 2^N$, d is a constant, $2 \leq d \leq 0.7N$, $\oplus$ is the XOR operator, $\gg$ and $\ll$ are the operators to shift right and left, $\gg d$ is equivalent to multiplying by 2^{-d} followed by a *floor* operation, while $\ll d$ is equivalent to multiplying by 2^d followed by the $\mod m$ operation.

This modification employs three types of operations of different natures. On the one hand, there is the $\mod 2^N$ operation used in algebraic generators; secondly, the XOR sum operator and then the shift operations to the right and left that are peculiar to shift register methods are used.

The Trifork generator has three branches, each consisting of a perturbed Fibonacci delayed generator. The three branches are interconnected with each other cyclically by the perturbation. Two branches are combined with the sum XOR operator to form the output of the joint generator, while the third generator remains hidden; thus, analyzing the output of the generator to discover the parameters would be a waste of time. This is one of the advantages of the Trifork generator over other generators.

The Trifork generator is defined as:

$$\begin{aligned} w_n &= x_n \oplus z_n \\ x_n &= (x_{n-r1} + x_{n-s1}) \mod m \oplus z'_n \\ y_n &= (y_{n-r2} + y_{n-s2}) \mod m \oplus x'_n \\ z_n &= (z_{n-r3} + z_{n-s3}) \mod m \oplus y'_n \\ x'_n &= ((x_{n-r1} + x_{n-s1}) \mod m) \gg d \\ y'_n &= ((y_{n-r2} + y_{n-s2}) \mod m) \gg d \\ z'_n &= ((z_{n-r3} + z_{n-s3}) \mod m) \gg d \end{aligned} \tag{12}$$

where w_n is the output of the generator at instant n; x_n, y_n, z_n are the outputs of the three generators; $r_1, s_1, r_2, s_2, r_3, s_3$ are the values of the delays; and r_1, r_2, r_3 should be different to ensure different lengths in the sequences.

The seeds of this generator are obtained as follows:

$$\begin{aligned} X_{k+1} &= aX_k + c \\ Y_{k+1} &= aY_k + c \\ Z_{k+1} &= aZ_k + c \end{aligned} \tag{13}$$

where X_k, Y_k and Z_k are the sets of the seeds of each individual generator.

4.8. Trit Generators

Among the new quantum cryptographic methods, we can highlight those that employ quantum secure direct communication (QSDC), which allows the transmission of information directly through an open channel without the need for encryption, eliminating the problem of key discovery. Several works related to this methodology require the generation of trits of pseudo-random numbers to be efficient. In [73], a method for generating secure pseudo-random numbers based on trits is proposed. The process consists of two steps: first, the internal state is initialized to subsequently generate the pseudo-random numbers. In more detail:

1. The internal state of the vector, U, is initialized. Using vector initialization $VI \in V_e$ and in the secret key $K \in V_n$, it is considered:

$$U = (x_1, x_2, x_3, x_4, x_5, x_6, y_1, y_2, y_3, y_4, k_1, k_2, k_3, k_4),$$

   where $x_i \in V_l$, $y_j \in V_l$ and $k_j \in V_l$ are parts of the vector U, with $i = 1, \ldots, 6$ and $j = 1, 2, 3, 4$. Vector VI is $VI = (VI_1, VI_2, \ldots, VI_{10})$, while the key is made up of four parts $K = (K_1, K_2, K_3, K_4)$. Then, vector U can be initialized as follows: $x_i = VI_i$, $y_j = VI_{6+j}$, $k_j = K_j$, with $i = 1, \ldots, 6; j = 1, ..., 4$.
2. The sequence is generated progressively, $M = (M_1, \ldots, M_b)$, $M \in V_m$, where M_q are the parts of the generated sequence.
   (a) To generate each of the M_q, the following steps are executed:
      i. New values for x_1, x_2, x_3.
      ii. New values for k_1, k_2, y_1, y_2.
      iii. New values for x_4, x_5, x_6.
      iv. New values for k_3, k_4, y_3, y_4.
   (b) The vectors y are concatenated, obtaining $M_q = (y_1 \mid y_2 \mid y_3 \mid y_4)$.

Based on this process, the authors computationally implemented this generator (*TriGenv.2.0*), whose pseudo-code is presented in Algorithm 1.

In [73], two comparisons are made in terms of the efficiency of this generator with the basic C++ language pseudo-random generator. The first one is compared with NIST Statistical Test Suite (NIST STS) [11], and the second one is compared with a modification of the NIST STS for pseudo-random number terns. The results are briefly detailed below:

1. For the case of the NIST STS, 100 sequences consisting of $6 \cdot 10^7$ trits from each generator were simulated. Subsequently, these terns are converted to bits and the test is applied. In this case, the TritGen generator failed the test most of the time. The authors conclude that the standard bit sequence tests do not work properly in evaluating the trit sequences.
2. In the test modified by the authors, five sequences of $1.5 \cdot 10^7$ trits were obtained in each generator. In this case, the TritGen generator showed better results than the C++ generator.

Algorithm 1 Pseudo-code TriGen v.2.0 [73]

Input: Vector VI, key K, $VI \in V_{240}$, $K \in V_{96}$ and parameter b
Output: Sequence $M = (M_1, \ldots, M_b)$, $M \in V_{96b}$, $M_q \in V_{96}$, $q \in \overline{1,b}$
1. $x_i \in VI_i$,$y_j \in VI_{6+j}$, $k_j \in K_j$, $i \in \overline{1,6}$, $j \in \overline{1,4}$.
2.
for q = k **step** 1 **until** $q \leq b$ **do**
  for j = 0 **step** 1 **until** $j < 4$ **do**
    $x_1 = (Sbox(x_1 + k_1) \oplus x_4) \lll k_4$; $x_2 = (Sbox(x_2 + k_2) \oplus x_5) \ggg k_3$;
    $x_3 = (Mix((x_3 + x_6) \oplus y_3) \lll x_1$;
    $k_1 = Sbox((Sbox(x_1 \oplus k_1) + x_5) \oplus y_1)$; $k_2 = Sbox(Mix(x_2 + k_2 + x_6) \oplus y_2)$;
    $y_1 = Sbox(((k_1 + y_1) \lll x_2) \oplus k_3)$; $y_2 = Mix(Sbox(((k_2 + y_2) \ggg x_3) \oplus k_4))$;
    $x_4 = (Sbox(x_4 + k_3) \oplus x_1) \lll k_2$; $x_5 = (Sbox(x_5 + k_4) + x_2) \ggg k_1$;
    $x_6 = Mix((x_6 + x_3) \oplus y_1) \lll x_4$;
    $k_3 = Sbox((Sbox(x_4 \oplus k_3) + x_2) \oplus y_3)$; $k_4 = Sbox(Mix(x_5 + k_4 + x_3) \oplus y_4)$;
    $y_3 = Sbox(((k_3 + y_3) \lll x_5) \oplus k_1)$; $y_4 = Mix(Sbox(((k_4 + y_4) \ggg x_6) \oplus k_2))$
  end
  $M_q = (y_1 \mid y_2 \mid y_3 \mid y_4)$
end

More detailed results of these simulations as well as details of the modified test can be found in [73].

Another pseudo-random number tern generator is proposed in [74], in which a primitive polynomial, the trace function, and Legendre's symbol in an odd feature field are employed. Before introducing the generator algorithm, several definitions are necessary (see Definitions 7 and 8).

Definition 7. *Legendre's symbol* (a/p) *for an element in a field* F_p *is defined as:*

$$(a/p) = \begin{cases} 0, & \text{if } a = 0 \\ 1, & \text{if } a \text{ is a quadratic residual different from } 0 \\ -1, & \text{if } a \text{ is not a quadratic residual different from } 0 \end{cases} \tag{14}$$

where an integer x *is a quadratic residual if,* $x^2 \equiv q \mod n$.

Legendre's symbol is calculated as $(a/p) = a^{(p-1)/2} \mod p$.

Definition 8. *The trace function performs a mapping from an element of the extension field* $X \in F_{p^m}$ *to an element of the prime field* $x \in F_p$ *as follows:*

$$x = Tr(X) = \sum_{i=0}^{m-1} X^{p^i} \tag{15}$$

Now, it is possible to define the algorithm. Let p be an odd characteristic prime, and let m be the degree of the primitive polynomial $f(x)$ over the field F_p. Using this polynomial, one can generate a vector of maximum length over F_{p^m}. Let ω be the primitive element of this field, and then the sequence of ternaries:

$$T = \{t_i\},\ t_i = \left(\frac{Tr(\omega^i) + A}{p}\right)\ i = 0, 1, 2, \ldots, p^m - 2 \tag{16}$$

is the longest length having period $p^m - 1$. With $A \in F_p$ not equal to 0.

In Table 1, we show in each category of the generator their characteristics, statistical properties, and security parameters.

Table 1. Overview of described generators.

Generator	Characteristics	Statistical Properties	Security Parameters
BBS	• Congruential model • Extracts least significant b_i bits from the sequence x_i	• Criptographically secure • Deterministic algorithm	• $N = pq$, where p and q are prime an verify $p \equiv q \equiv 3 \, mod \, 4$ • $y \, \underline{mod} \, N \in \left(-\frac{N}{2}, \frac{N}{2}\right)$ • $E_N(y) = \lvert y^2 \, \underline{mod} \, N \rvert$, absolute of Rabin's function
LFSR	• Uses bit sequences to obtain a criptographically secure sequence • Common in the literature	• Random sequences • Scalable algorithm	• Secure based on its linear complexity • Resources used are pseudo-random
/dev/random	• Linux random number generator • Two pools and two functions	• Unpredictable sequences • External entropy	• Pools partitions increase the security • Information from the system itself
AES	• Block cipher scheme • Widely used in symmetric cryptography • Two ways of generation	• Encryption algorithm • Different operations using bytes	• Fuzziness and diffusion • No effect if the seed is modified externally
Yarrow	• Four components • Uses hash functions	• Encrypted with triple-DES algorithm • No entropy overestimation	• Input does not have direct influence on the output • Key updates with accurate entropy estimators • Pool separation
Fortuna	• Improvement to Yarrow generator • Three parts: generator, accumulator, and reseed control file	• Encrypted with AES algorithm • Unpredictable resources	• New key every generation • 32 pools gather enough entropy
Trifork	• Combination of three sequences generated with delayed Fibonacci generators, mutually perturbed • Modification of conventional method	• Unpredictable output • Employs three operations: mod 2^N, sum XOR, and shift operations	• Third generator remains hidden • Parameters are chosen as appropriate

5. Experimental Analysis

In this section, we will analyze the randomness of the PRNGs described in the previous section. For this purpose, we will apply the NIST Statistical Test Suite (NIST STS) [11] to the output sequence in binary format of these algorithms.

It is reasonable to expect that all the generators analyzed will pass the NIST STS with satisfactory results since these algorithms have been proposed following a correct statistical design. However, this analysis will be carried out with the objectives of empirically showing the statistical goodness of these generators and checking their correct implementation.

On the other hand, a study will be conducted on the computational speed of data generation with the described generators to later establish a series of recommendations for the user when deciding on one generator or another.

In order to consider whether the proportion of sequences passing a test is acceptable or not, we are working with the following confidence interval, recommended in [11]:

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}} \tag{17}$$

where $\hat{p} = 1 - \alpha$ and m is the sample size.

5.1. Materials and Methodology

The computer used for the simulations is an HP Pavilion laptop with 16 GB of RAM and an Intel core i7-10750H CPU @ 2.60GHz. For the application of the tests, the Linux console was used, specifically the STS package that is available on the official NIST website (https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software, accessed on 1 January 2020).

In the cases studied, the generator is considered to pass the test if 96 or more sequences pass the test. For the random excursion test and its variant, this ratio is approximately 0.95, which implies that 60 sequences have to pass the test.

To carry out this analysis, 100 sequences of 10^6 bits each will be generated. This number of bits is because some tests require this amount in order to be executed, while the number of sequences used is the number recommended in the publication for the parameters to be used in this work. Each sequence is generated from a randomly chosen seed, and in the case of generators that require additional parameters, they are also randomly generated. In the case of the random excursion test and its variant, due to the construction of both tests, only 63 of the 100 sequences generated are used.

Finally, the parameters used in the applications of the tests suite, following the recommendations of [11], are:

- Significance level (α): 0.01;
- Size of the block in the frequency test within a block: 128;
- Size of the length in bits of each template in the non-overlapping template matching test: 9;
- Length in bits of the template in the overlapping template matching test: 9;
- The length of each block in the approximate entropy test: 10;
- The length in bits of each block in the serial test: 16;
- The length in bits of a block in the linear complexity test: 500;
- Maximum number of templates: 40.

The results obtained with the different generators are analyzed next.

5.2. Results

We will begin by describing how each of the generators described in Section 4 was implemented and indicating the computational time involved in generating the sequences.

The R program and the Linux system were used to program the codes. In particular, the R language was used for the implementation of the BBS, Shrinkage, Self-Shrinkage, ASG, AES, and Trifork generators and the Linux system for the implementation of the /dev/random, Sober-128, Yarrow, and Fortuna generators. The codes can be found in Appendix A.

In particular, for the implementation of the BBS generator, we used the function *auxiliary* to obtain the value of the parameters and the seed necessary for this algorithm, and the function *bbs*, which is in charge of the generation of the pseudo-random numbers. To perform each of the 100 simulations, both functions were executed, obtaining different and independent values in each simulation. The average computation time for a simulation of one million bits was 37.15 s.

For the Shrinkage generator, the function *lfsr* was implemented to obtain the sequences that feed the generator, and subsequently, the function *shrinkage* was used for the generation of the binary sequence. In order to obtain each sequence, each of the components necessary

for the execution of the functions was chosen randomly. In the polynomials, a size of 16 has been chosen, with the elements of these being random. With this size, we tried to have as much variability as possible in the polynomials without using a lot of computing time. The average computation time for the generation of a sequence of one million bits was 88.7 s.

In the case of the Self-Shrinkage generator, the function *lfsr* was implemented to obtain the sequence to be used in the generator, and subsequently the function $self\_shrinkage$ was used for the generation of the binary sequence. To obtain each sequence, each of the components necessary for the execution of the functions was chosen randomly; in the case of the polynomial, size 16 was chosen with its random components. An average time of 85.62 s was used to generate a sequence of one million bits.

For the ASG generator, the *lfsr* function was used to obtain the three sequences to be used in the generator, and then the *asg* function was used for the generation of the binary sequence. In order to obtain each sequence, each of the components necessary for the execution of the functions was chosen randomly. As in the two previous generators, the size of the polynomials was 16. An average time of 61.42 s was used to generate a sequence of one million bits.

In the case of the AES generator, the function *AES* was implemented. The generation mode used was the counter due to its greater security against attacks. Between each sequence of one million bits, the counter and the key used are updated, so that the sequences obtained are independent. The average time used for the generation of each of the one million bit sequences is 5.03 s.

The generation of the sequences with the Trifork algorithm was carried out by means of the code of the functions *fibonacci* for obtaining the sequences of the delayed Fibonacci algorithm and the function *trifork* for the generation of the binary sequences. Different random parameters were used in each sequence, as these are independent. It should be noted that a power of 2 was used as a modulus, and the recommendations of the original paper were followed for the d parameter. An average time of 58.42 s was used to generate a sequence of one million bits.

As for the generators programmed in Linux, in the case of /dev/random, it was not necessary to load any package since it is the generator of this operating system. The binary sequences were obtained directly without any additional transformation of the data. The sequences were simulated independently in such a way that between each sequence, the entropy resource of the generator is updated. The average computation time to obtain a sequence of one million bits was 2.88 s.

In the case of the Sober-128 generator, the CryptX package was used, which can be downloaded free of charge from the Ubuntu repository (http://manpages.ubuntu.com/manpages/impish/man3/CryptX.3pm.html, accessed on 1 January 2022) Between the simulation of each million bit sequence, the seed is updated and the entropy is renewed. When using this package, the output is in hexadecimal format, so it is necessary to make the transformation to binary to apply the tests. The average computation time to obtain each of these sequences is 3.84 s.

In the case of the Yarrow generator, it was also simulated through the CryptX package in the Linux console. Between the simulation of each bit sequence an update of the seed is performed and the entropy of the generator is renewed. The output of the function used is again in hexadecimal format, with the transformation to binary being necessary to apply the battery tests. The average computation time for the generation of a sequence of one million bits was 3.56 s.

Like the Yarrow generator, of which it is an improved version, for the Fortuna generator, the simulation was performed through the Linux console with the CryptX package. An update of the seed and entropy was also performed after each binary sequence. As with the two previous generators, hexadecimal sequences were obtained, so it was necessary to transform them to binary in order to apply the different tests. The average computation time was 3.16 s for a sequence of one million bits.

As for the results obtained after the application of the NIST tests on the sequences generated from each and every one of the generators, these were satisfactory in general and

allow us to affirm that the generators passed the requirements of the NIST STS. Next, we will detail the results broken down by generator and we will show, for illustrative purposes and without loss of generality, some graphic examples of the histograms of the *p*-values resulting from some tests.

The results obtained with the BBS generator were satisfactory. The lowest acceptance rate for the tests, except for the random excursion test and its variant, is 97 out of 100 binary sequences. In the case of the random excursion test and its variant, this minimum rate was 61 out of 63 binary sequences.

Analyzing uniformity, the null hypothesis of uniformity is not rejected for a significance level of 1%. As an illustration, Figure 1 shows the distribution of these *p*-values for the case of the linear complexity test.

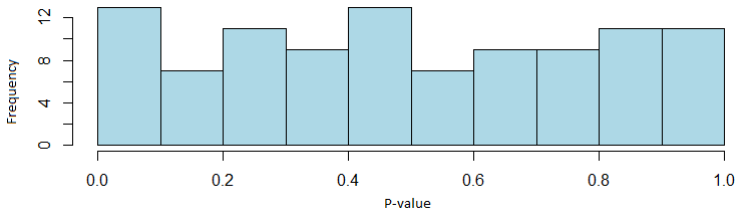

Figure 1. Histogram of the *p*-values of the linear complexity test (BBS case).

With these results, it can be concluded that since the proportions of sequences passing the test are within the confidence interval previously defined, the BBS generator passes the NIST test battery, and therefore, random sequences are obtained with this algorithm that can be used in cryptography. Furthermore, through the distribution of the *p*-values, it can also be determined that the sequences are uniformly distributed.

The Shrinkage generator passes all the tests with solvency except for a non-overlapping template matching test, which was passed for 95 out of 100 sequences; however, this acceptance rate is not included within the confidence interval defined above. Removing this test, the lowest number of sequences passing a test is 96 out of 100 sequences for all but the random excursion test and its variant, where the lowest number of binary sequences passing the test is 61 out of 63.

As for uniformity, there is no evidence of the rejection of the null hypothesis of uniformity at 1% in any of the tests performed. In Figure 2, the distribution of the *p*-values for the case of the discrete Fourier transform test is shown.

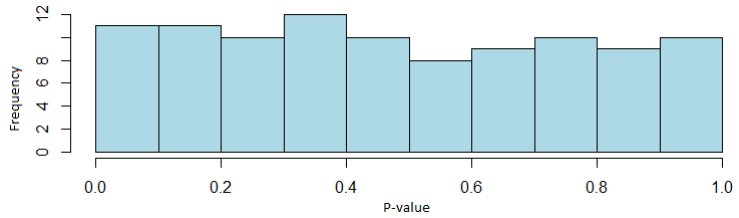

Figure 2. Histogram of the *p*-values of the discrete Fourier transform test (Shrinkage generator case).

According to the obtained results, it can be concluded that since the proportions of sequences passing the test are within the confidence interval previously defined, the Shrinkage generator passes the NIST test battery, and therefore, random sequences are obtained with this algorithm. Although a non-overlapping template matching test is not passed, since there are several such tests with acceptance ratios within the confidence interval, it is considered that this test will not cause the randomness of the generator to be lost. It is possible that with other sequences and with a greater number of bits in them, it is obtained that this generator passes all the tests with solvency. In addition, through the distribution of the *p*-values it can also be determined that the sequences are uniformly distributed.

Regarding the results obtained by using the Self-Shrinkage generator, it should be noted that there are two non-overlapping template matching tests in which the proportion of sequences passing these tests is not within the defined confidence interval; in both cases, 94 out of 100 binary sequences pass the test. Except for these two situations, the other tests are passed with confidence, with the minimum acceptance rate being 61 out of 63 sequences for the random excursion test and its variant and 96 out of 100 for the rest of the tests.

In relation to the distribution of the *p*-values, it should be noted that there is one non-overlapping template matching test that rejects the null hypothesis of uniformity; in the rest of the cases, there is no evidence to reject this hypothesis and therefore uniformity in the sequences. Figure 3 shows the distribution of the *p*-values of one of the non-overlapping template matching tests, showing the uniform distribution of the *p*-values.

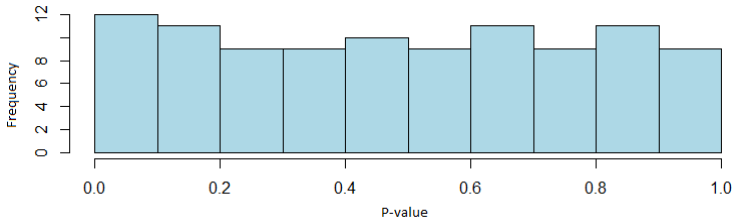

Figure 3. Histogram of the *p*-values of the non-overlapping template test (Self-Shrinkage generator case).

After analyzing the performed tests, the Self-Shrinkage generator passes the NIST STS, since most of the tests have an acceptance ratio within the defined confidence interval. Despite the failure of two non-overlapping template matching tests, since there are several such tests with acceptance ratios within the confidence interval, it is considered that these tests will not cause the randomness of the generator to be lost. Furthermore, through the distribution of the *p*-values, it can also be determined that the sequences are uniformly distributed, except for a non-overlapping template matching test. It is possible that with other sequences and with a larger number of bits in the sequences, this generator may pass all tests with flying colors, both for the sequences and for the distribution of the *p*-values.

In the case of the ASG generator, the proportion of sequences that passed each of the tests was found to be within the confidence interval. The minimum proportion for the tests with 100 sequences was 96 out of 100, while for the random excursion tests and its variant, this proportion was 61 out of 63 binary sequences.

Regarding the distribution of the *p*-values, it was obtained that none of the tests has evidence to reject the null hypothesis of uniformity in the χ^2 test for a significance level of 1%. Figure 4 shows the histogram of the *p*-values of the linear complexity test, showing their uniform distribution.

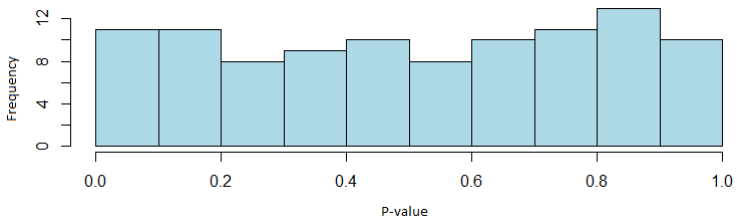

Figure 4. Histogram of the *p*-values of the linear complexity test (ASG generator case).

With these results, it can be concluded that since the proportions of sequences passing the test are within the confidence interval previously defined, the Alternating Step generator (ASG) passes the NIST test battery, and therefore random sequences are obtained with this algorithm. Furthermore, through the distribution of the *p*-values, it could also be determined that the sequences are uniformly distributed. Concluding that this generator is suitable for the field of cryptography.

The /dev/random generator passes all the tests since the proportion of sequences that pass each test are within the defined confidence interval. For the case of the random excursion test and its variant, the minimum number of sequences passing the test is 61 out of 63 binary sequences, while for the rest of the tests, it is 96 out of 100 binary sequences.

Observing the uniformity of the p-values, it is obtained that they all pass the χ^2 test at 1%. Figure 5 shows the distribution of the p-values for the cumulative sums test.

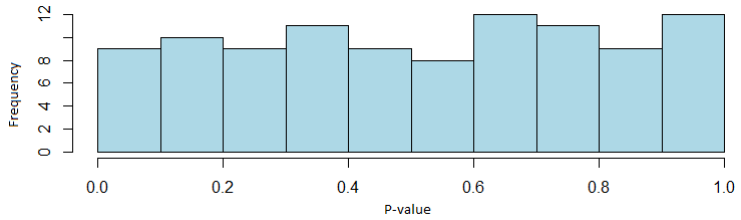

Figure 5. Histogram of the p-values of the cumulative sums test (/dev/random generator case).

Once the different tests were performed, it could be concluded that the /dev/random generator produces random bit sequences, also being uniform, so this algorithm can be used in the generation of random numbers for cryptography.

The results obtained by using the AES generator are also satisfactory. It can be seen that the generated sequences pass the tests with flying colors. For the random excursion test and its variant, the lowest acceptance rate is 61 out of 63 sequences, while for the rest of the tests, this proportion is 96 out of 100 binary sequences, both values being within the confidence interval.

In terms of uniformity, it can also be seen that the different tests pass the χ^2 test. This can be seen in Figure 6, which represents the distribution of the p-values for the case of the non-overlapping template matching test.

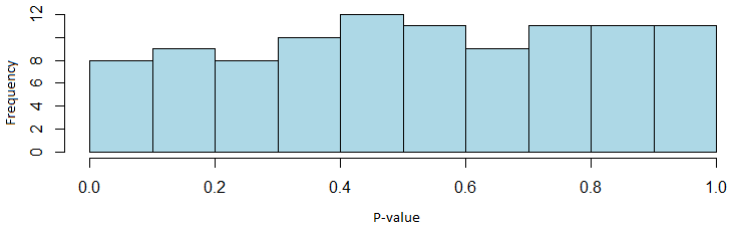

Figure 6. Histogram of the p-values of the non-overlapping template test (AES generator case).

Once the obtained results were analyzed, it could be concluded that since the proportions of sequences that pass each of the tests are within the confidence interval previously defined, the AES generator passes the NIST STS, and therefore random sequences are obtained that can be used in the field of cryptography. Furthermore, through the distribution of the p-values, it has been determined that the generated sequences are uniformly distributed.

The results of the test battery show that the Sober-128 generator passes the tests successfully. In the random excursion tests and its variant, the minimum acceptance rate is 62 out of 63 binary sequences, while for the rest of the tests this proportion is 96 out of 100 sequences. In both cases, they are within the confidence interval.

Regarding the distribution of the p-values, it can be observed that none of them finds evidence to reject the null hypothesis of uniformity in the χ^2 test. Figure 7 shows the distribution of the p-values of one of the random excursion tests to exemplify uniformity.

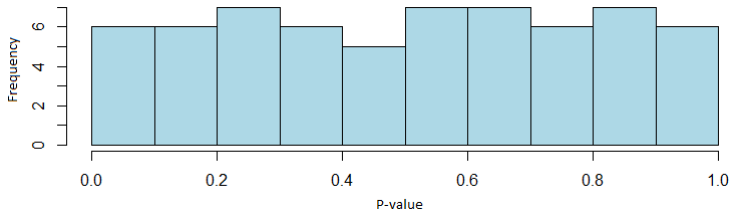

Figure 7. Histogram of the *p*-values of the random excursions test (Sober-128 generator case).

In view of the obtained results, it can be concluded that the Sober-128 generator passes the NIST STS, and therefore random sequences are obtained with this algorithm that can be used in the field of cryptography. This is because the acceptance rates of the tests are within the confidence interval that has been defined to accept the generator or reject it. Furthermore, through the distribution of the *p*-values, it was determined that the sequences are uniformly distributed.

Analyzing the results obtained by using the Yarrow generator, it can be observed that this generator passes all the tests with solvency except for the non-overlapping template coincidence test, in which 94 sequences pass it; however, this acceptance rate is not included within the confidence interval defined above. Removing this test, the lowest proportion of sequences passing the random excursion test and its variant is 61 out of 63 binary sequences. While for the rest of the tests, this proportion is 96 out of 100.

As for the uniformity of the *p*-values, it can be seen that it is obtained in all tests. As an example, the graph for the frequency test, whose values pass the uniformity test for a confidence level of 1%, is shown in Figure 8.

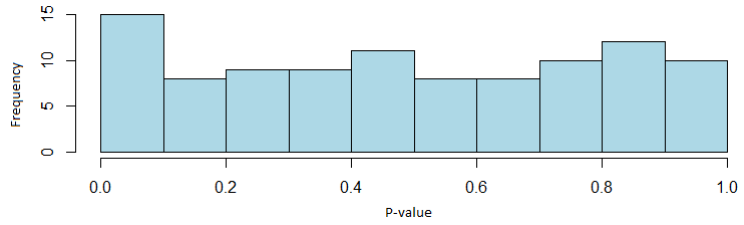

Figure 8. Histogram of the *p*-values of the frequency test (Yarrow generator case).

With these results, it can be concluded that since the proportions of sequences passing the tests are within the confidence interval previously defined, the Yarrow generator passes the NIST test suite, and therefore, random sequences are obtained with this algorithm. Although a non-overlapping template matching test is not passed, since there are several such tests with acceptance ratios within the confidence interval, it is considered that this test will not cause the randomness of the generator to be lost. It is possible that with other sequences and in greater number, it is obtained that this generator passes all the tests with solvency. In addition, through the distribution of the *p*-values, it can also be determined that the sequences are uniformly distributed.

The results obtained by using the Fortuna generator show that this generator passes all the tests since the lowest number of sequences passing a test is 97 out of 100, while for the random excursion test and its variant, this proportion is 62 sequences out of 63. In both cases, the proportions of sequences passing the test are within the confidence interval.

As with the other generators, there is uniformity in the *p*-values. Figure 9 shows the distribution for the case of Maurer's universal test. The different tests pass the χ^2 test for a significance level of 1%.

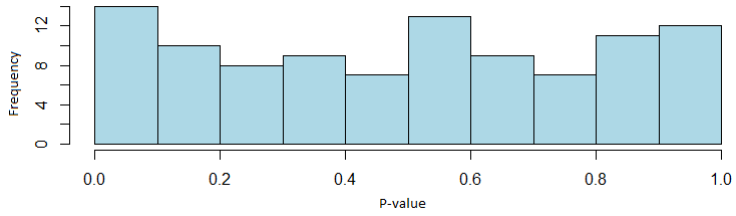

Figure 9. Histogram of the *p*-values of the Maurer's universal test (Fortuna generator case).

In view of the results obtained with the different tests, it can be concluded that random sequences are obtained with the Fortuna generator, since it has been verified that the proportion of sequences that passed the tests is within the confidence interval that has been defined to consider that a test has been passed. It was also possible to verify the uniformity of the sequences through the distribution of the *p*-values, so this generator is suitable to be applied in cryptography.

In relation to the results obtained by using the Trifork generator, it can be seen that the binary sequences pass the tests with solvency, since, as can be seen, the lowest proportion in the acceptance rate of the tests is within the defined confidence interval. Specifically, in the case of the random excursion tests and its variant, this minimum proportion is 61 out of 63 sequences, while for the rest of the tests it is 96 out of 100.

As for the distribution of the *p*-values, in no case is there evidence to reject the null hypothesis of uniformity, from which it is inferred that the sequences are going to be uniform. The Figure 10 shows the distribution of the streak test, exemplifying the uniform distribution of the *p*-values.

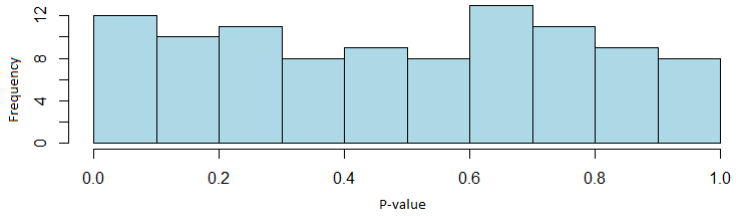

Figure 10. Histogram of the *p*-values of the runs test (Trifork generator case).

With these results, it can be concluded that, as the proportions of sequences passing each test are within the confidence interval previously defined, the Trifork generator passes the NIST STS, and therefore, random sequences are obtained with this algorithm. Furthermore, through the distribution of the *p*-values, it has also been determined that the sequences are uniformly distributed, making this generator suitable for the field of cryptography.

6. Conclusions

The applications of cryptography are diverse, as highlighted in the Introduction, so it is necessary that the elements that compose it, in particular the pseudo-random numbers, are as secure as possible. For this reason, in this work we have analyzed different generators of these numbers, paying special attention to those that verify a series of special properties for the field of cryptography, such as randomness and unpredictability. In addition, we have tried to find the generator that, fulfilling the necessary conditions, has a lower computational cost. The analysis of these generators has been carried out in two parts, in the first one, each generator has been exposed examining the algorithm used, emphasizing the possible security flaws that may occur in the process of obtaining the sequences. In the second part, a simulation of binary sequences has been carried out using the generators previously analyzed. Once these bit strings were obtained, the NIST STS

was applied to analyze their randomness and uniformity, and the computational cost of obtaining the sequences with each generator was also calculated.

In the simulation of the sequences, both the R code package and the packages already implemented in the Linux console were used, obtaining better results in computational cost in the latter as they are better optimized.

A number of relevant conclusions can be drawn from the study of the different generators:

- The Blum–Blum–Shub generator passed the NIST test suite with solvency, both in randomness and in the uniform distribution. In computational terms, it performed well, being a good pseudo-random number generator for cryptographic applications.
- Generators based on linear feedback shift register (LFSR), such as the Shrinkage generator, Self-Shrinkage generator and Alternating Step generator (ASG) were the slowest in computational terms among the generators analyzed. As for the application of the test battery, they have managed to pass the tests, although in the case of the Shrinkage and Self-Shrinkage generators, some tests were not passed due to the proportion of sequences necessary to be considered acceptable. Even so, it is concluded that with these generators, random sequences with uniform distribution are obtained, and these generators are suitable for the field of cryptography.
- The /dev/random generator, the one used in the Linux system, is the fastest in computational terms. This may be due to a higher optimization in its processes. In the test results, it was able to pass all of them with flying colors, both in terms of randomness and in terms of the uniform distribution. All this makes this generator suitable for the generation of cryptographically secure pseudo-random numbers.
- The AES generator obtained good results in both tests and computational cost. In the latter, it is the fastest of the algorithms implemented in R. Randomness and uniform distribution have been verified, making this generator suitable for cryptography.
- In the case of the Sober-128 generator, the execution time was reduced. Even so, the results of the test battery were satisfactory in both randomness and uniform distribution of the generated sequences. This makes this algorithm suitable for the generation of cryptographically secure pseudo-random numbers.
- Both the Yarrow generator and its predecessor, Fortuna, performed well in computational terms, the latter being slightly better. As for the results of the battery of tests, both algorithms managed to pass them with solvency; however, in the case of the Yarrow generator, it is the case that in a comparison test of non-overlapping templates, the proportion of accepted sequences was not sufficient. In spite of this, it is concluded that both generators yield random sequences with uniform distribution. In view of the results, it would be better to use the Fortuna generator for the generation of random numbers in cryptography.
- The Trifork generator, based on the delayed Fibonacci generators, obtained good results in the battery of tests, having in all of them an acceptance rate within the defined confidence interval. In computational terms, the result is satisfactory, so it can be concluded that this generator is suitable for the generation of secure pseudo-random numbers for cryptography.

Generally speaking, the generators analyzed have been found to be suitable for producing pseudo-random numbers that can be used in cryptography. However, some have shown better performance both in computation time and in passing the tests. The LFSR-based generators were the worst performers in both situations, making them less suitable for pseudo-random number generation in cryptography. The rest of the generators showed similar behavior in terms of test acceptance rates, but there are differences in computation time, making some of them better for the generation task.

In order to compare the different generators, Table 2 shows the computation times used in the simulation of a sequence of one million bits, and, on the other hand, Table 3 shows the advantages and disadvantages of each generator.

Table 2. Average time in seconds taken to generate a sequence of one million bits.

Generator	Time (In Seconds)
BBS	37.15
Shrinkage	88.7
Self-Shrinkage	85.62
ASG	61.42
/dev/random	2.88
AES	5.03
Sober-128	3.84
Yarrow	3.56
Fortuna	3.16
Trifork	58.42

Table 3. Pros and cons of each generator.

Generator	Pros	Cons
BBS	Conceptually simple Easy implementation in any environment	Special conditions are required for the parameters Slow running If the seed is guessed, all the outputs are known
LFSR	Simple and easy to understand design Fast implementation in computational terms Easy to include in other generators	High computational cost Weak linear algorithm against initial state recovery attacks Vulnerable to some attacks if not combined with other generators
/dev/random	Computationally efficient Optimally implemented in Linux Secure against various cryptographic attacks	Slow in low entropy contexts Dependent on entropy quality
AES	Simple implementation if the necessary packages are available in the programming language Fast execution Secure against several cryptographic attacks	Complex design Weak if internal condition is compromised
Sober-128	Implemented in Linux Fixes shortcomings of LFSR generators Not vulnerable to algebraic attacks and key attacks	Conceptually complex Vulnerable to attack if LFSR generator is known
Yarrow	Secure against iterative and backtracking attacks Implemented in Linux	Complex design Entropy estimators need to be defined
Fortuna	Implemented in Linux Solves the problem of entropy estimators Computationally efficient	Complex implementation depending on the programming environment The seed control file can cause memory problems
Trifork	Fast and easy implementation Corrects deficiencies of delayed Fibonacci generators Can be used in other generators	Conceptually complicated if no previous knowledge is present Slow running

The aim of this work was the analysis of different cryptographically secure pseudo-random number generators, looking for those that passed the NIST Statistical Test Suite and had the lowest possible computational cost. In this sense, it can be concluded that the Fortuna, /dev/random, Sober-128, and AES generators are the most suitable for obtaining these numbers due to their performance in the tests performed and their low computational cost.

Author Contributions: Conceptualization, E.A.L. and J.R.V.; methodology, E.A.L. and J.R.V.; software, E.A.L. and J.R.V.; validation, E.A.L. and J.R.V.; formal analysis, E.A.L. and J.R.V.; investigation, E.A.L. and J.R.V.; resources, E.A.L. and J.R.V.; data curation, E.A.L. and J.R.V.; writing—original draft preparation, E.A.L. and J.R.V.; writing—review and editing, E.A.L. and J.R.V.; visualization, E.A.L. and J.R.V.; supervision, E.A.L. and J.R.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: We thank the Editorial Office for their support for the dissemination of this work. We also thank the reviewers for their suggestions on how to improve our work.

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

Appendix A. Codes

```
library(primes)
library(digest)
library(dplyr)

###BBS

auxiliar<-function(){

  aux=c()
  p=sample(primes,1)
  q=sample(primes,1)
  while(p %% 4 !=3){
    p=sample(primes,1)
  }
  while(q %% 4 !=3 & q!=p){
    q=sample(primes,1)
  }
  n=p*q
  s=sample(n,1)
  while (gcd(s,n)!=1){
    s=sample(n,1)
  }
  aux=append(aux,c(p,q,s))
  return(aux)
}

bbs<-function(m){

  pri=auxiliar()
  p=pri[1]
  q=pri[2]
  x=pri[3]

  N=p*q
  secuencia=c()
  for (i in 1:m){
    x=x**2 %% N
    secuencia=append(secuencia,x %% 2)
  }
  return (secuencia)
}
```

```
###LFSR

lfsr<-function(seed,pol,m){

  x=seed
  secuencia=c()
  for (i in 1:m){
    secuencia=append(secuencia,tail(x,1))
    nuevo=(sum(x[pol])%% 2)
    x=head(x,-1)
    x=append(nuevo,x)
  }

  return(list(sec=secuencia,seed=x))
}


##Shrinkage

shrinkage<-function(pol_1,pol_2,m,n){

  secuencia=c()

  z=c(0,1)
  seed_1=sample(z,n,replace=TRUE)
  seed_2=sample(z,n,replace=TRUE)

  sec_1=lfsr(seed_1,pol_1,3*m)$sec
  sec_2=lfsr(seed_2,pol_2,3*m)$sec

  for (i in 1:(3*m)){
      if (sec_2[i]==1 & length(secuencia)<m) {
        secuencia=append(secuencia,sec_1[i])}

  }
  return (secuencia)
}

##Self-shrinkage

self_shrinkage<-function(pol,m,n){

  secuencia=c()

  z=c(0,1)
  seed=sample(z,n,replace=TRUE)

  sec=lfsr(seed,pol,4*m)$sec

  for(i in 1:(2*m)){
    if (sec[2*i-1]==1 & sec[2*i]==0 & length(secuencia)<m){
    secuencia=append(secuencia,0)}
    else if (sec[2*i-1]==1 & sec[2*i]==1  & length(secuencia)<m){
      secuencia=append(secuencia,1)}
  }

  return(secuencia)
}


##Alternating Step Generator
asg<-function(pol_1,pol_2,pol_3,m,n){

  secuencia=c()
  z=c(0,1)
  seed_1=sample(z,n,replace=TRUE)
  seed_2=sample(z,n,replace=TRUE)
  seed_3=sample(z,n,replace=TRUE)
```

```
  sec_1=lfsr(seed_1,pol_1,m+1)
  sec_2=lfsr(seed_2,pol_2,1)
  sec_3=lfsr(seed_3,pol_3,1)

  for(i in 1:m+1){
    if (sec_1$sec[i]==1){
      sec_2=lfsr(sec_2$seed,pol_2,1)
      secuencia=append(secuencia,as.integer(xor(sec_2$sec,sec_3$sec)))
    }
    else if (sec_1$sec[i]==0){
      sec_3=lfsr(sec_3$seed,pol_3,1)
      secuencia=append(secuencia,as.integer(xor(sec_2$sec,sec_3$sec)))
    }
  }

  return(secuencia)
}


##Lagged Fibonacci

fibonacci<-function(lag_1,lag_2,m,n,mod){

  secuencia=c()

  z=c(0,1)
  q=max(lag_1,lag_2)
  seed=sample(z,n,replace=TRUE)

  if (n+1>q){
    while(length(secuencia)<m){
      valor=(seed[(n+1)-lag_1]+seed[(n+1)-lag_2])%% mod
      secuencia=append(secuencia,valor)

      seed=append(seed,valor)
      seed=seed[-1]

    }
  }
  else print(''La semilla tiene que ser mas grande'')

  return(secuencia)
}


##Trifork
trifork<-function(r1,s1,r2,s2,r3,s3,m,n,mod,d){

  secuencia=c()

  fib_1=fibonacci(r1,s1,m,n,mod)
  fib_2=fibonacci(r2,s2,m,n,mod)
  fib_3=fibonacci(r3,s3,m,n,mod)

  x_prima=floor(fib_1*2^(-d))
  y_prima=floor(fib_2*2^(-d))
  z_prima=floor(fib_3*2^(-d))

  x=as.integer(xor(fib_1,z_prima))
  y=as.integer(xor(fib_2,x_prima))
  z=as.integer(xor(fib_3,y_prima))

  secuencia=as.integer(x,z)

  return(secuencia)
}
```

```
##AES
AES_128<-function(v,K){

  cont=as.raw(c(v:(v+15))%%2**128)
  key=as.raw(K)

  aes <- AES(key, mode=''CTR'',IV=01)
  bloq<-aes$encrypt(cont)

  out<-bloq%>%
    rawToBits()%>%
    as.integer()%>%
    paste(collapse ='')


  return(out)
}
```

References

1. Chen, I.T.; Tsai, J.M.; Tzeng, J. Audio random number generator and its application. In Proceedings of the 2011 International Conference on Machine Learning and Cybernetics, Guilin, China, 10–13 July 2011; Volume 4, pp. 1678–1683.
2. Dhaou, I.B.; Skhiri, H.; Tenhunen, H. Study and Implementation of a Secure Random Number Generator for DSRC Devices. In Proceedings of the 2017 9th IEEE-GCC Conference and Exhibition (GCCCE), Manama, Bahrain, 8–11 May 2017; pp. 1–9.
3. Nguyen-Duc, A.; Viet Do, M.; Quan, L.; Nguyen Khac, K.; Nguyen Quang, A. On the adoption of static analysis for software security assessment-A case study of an open-source e-government project. *Comput. Secur.* **2021**, *111*, 102470. [CrossRef]
4. Choi, J. Physical Layer Security for Channel-Aware Random Access with Opportunistic Jamming. *IEEE Trans. Inf. Forensics Secur.* **2017**, *12*, 2699–2711. [CrossRef]
5. Tang, J.; Jiao, L.; Zeng, K.; Wen, H.; Qin, K.Y. Physical Layer Secure MIMO Communications Against Eavesdroppers with Arbitrary Number of Antennas. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 466–481. [CrossRef]
6. Gedam, S.; Beaudet, S. Monte Carlo simulation using Excel(R) spreadsheet for predicting reliability of a complex system. In Proceedings of the Annual Reliability and Maintainability Symposium, 2000 Proceedings, International Symposium on Product Quality and Integrity (Cat. No.00CH37055), Los Angeles, CA, USA, 24–27 January 2000; pp. 188–193.
7. Gergely, A.M.; Crainicu, B. A succinct survey on (Pseudo)-random number generators from a cryptographic perspective. In Proceedings of the 2017 5th International Symposium on Digital Forensic and Security (ISDFS), Tirgu Mures, Romania, 26–28 April 2017; Volume 42, pp. 1–6.
8. Wang, P.; You, F.; He, S. Design of Broadband Compressed Sampling Receiver Based on Concurrent Alternate Random Sequences. *IEEE Access* **2019**, *7*, 135525–135538. [CrossRef]
9. Benedetti, R.; Andreano, M.S.; Piersimoni, F. Sample selection when a multivariate set of size measures is available. *Stat. Methods Appl.* **2019**, *28*, 1–25. [CrossRef]
10. D'Ovidio, M.; Polito, F. Discussion on the paper "On simulation and properties of the stable law" by L. Devroye and L. James. *Stat. Methods Appl.* **2014**, *23*, 359–363. [CrossRef]
11. Bassham, L.E.; Rukhin, A.L.; Soto, J.; Nechvatal, J.R.; Smid, M.E.; Barker, E.B.; Leigh, S.D.; Levenson, M.; Vangel, M.; Banks, D.L.; et al. *SP 800-22 Rev. 1a*; A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications; National Institute of Standards & Technology: Gaithersburg, MD, USA, 2010.
12. Tuncer, T.; Avaroglu, E. Random number generation with LFSR based stream cipher algorithms. In Proceedings of the 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 22–26 May 2017; Volume 42, pp. 171–175.
13. Marsaglia, G.; Zaman, A. A new class of random number generators. *Ann. Appl. Probab.* **1991**, *1*, 462–480. [CrossRef]
14. Warnock, T. Random-number generators. *Los Alamos Sci.* **1987**, *15*, 137–141.
15. Rubinstein, R.Y.; Kroese, D.P. *Simulation and the Monte Carlo Method*, 3rd ed.; John Wiley & Sons: Hoboken, NJ, USA, 2016.
16. Altiok, T.; Melamed, B. *Simulation Modeling and Analysis with ARENA*; Elsevier: Amsterdam, The Netherlands, 2007.
17. Barak, B.; Shaltiel, R.; Tromer, E. True Random Number Generators Secure in a Changing Environment. In *Cryptographic Hardware and Embedded Systems, Proceedings of the Cryptographic Hardware and Embedded Systems-CHES 2003, Cologne, Germany, 8–10 September 2003*; Walter, C.D., Koç, Ç.K., Paar, C., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2779, pp. 166–180.
18. Sunar, B. True random number generators for cryptography. In *Cryptographic Engineering*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 55–73.
19. Almaraz Luengo, E. A brief and understandable guide to pseudo-random number generators and specific models for security. *Stat. Surv.* **2022**, *16*, 137–181. [CrossRef]

20. Santha, M.; Vazirani, U.M. Generating quasi-random sequences from semi-random sources. *J. Comput. Syst. Sci.* **1986**, *33*, 75–87. [CrossRef]
21. Niederreiter, H. *Random Number Generation and Quasi-Monte Carlo Methods*; SIAM: Philadelphia, PA, USA, 1992.
22. Chaitin, G.J. On the length of programs for computing finite binary sequences. *J. ACM* **1966**, *13*, 547–569. [CrossRef]
23. Schindler, W. Random number generators for cryptographic applications. In *Cryptographic Engineering*; Koç, K.E., Ed.; Springer: Boston, MA, USA, 2009; pp. 5–23.
24. *ISO/IEC 18031:2011*; Information Technology. International Organization for Standardization: Geneva, Switzerland, 2011. Available online: https://www.iso.org/standard/54945.html (accessed on 1 January 2022).
25. Marsaglia, G. The Marsaglia Random Number CDROM Including the Diehard Battery of Tests of Randomness. 1995. Available online: https://web.archive.org/web/20160220101002/http://stat.fsu.edu/pub/diehard/ (accessed on 1 January 2022).
26. Brown, R.G.; Eddelbuettel, D.; Bauer, D. Dieharder: A Random Number Test Suite (Version 3.31.1). 2014. Available online: https://webhome.phy.duke.edu/~rgb/General/dieharder.php (accessed on 1 January 2022).
27. Practically Random: C++ Library of Statistical Tests for Rngs. 2010. Available online: https://sourceforge.net/projects/pracrand (accessed on 1 January 2022).
28. Walker, J. ENT: A Pseudorandom Number Sequence Test Program. 2008. Available online: https://www.fourmilab.ch/random/ (accessed on 1 January 2022).
29. *FIPS PUB 140-2*; Security Requirements for Cryptographic Modules Share to Facebook. National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2001. Available online: https://csrc.nist.gov/publications/detail/fips/140/2/final (accessed on 1 January 2022).
30. *FIPS 140-3*; Security Requirements for Cryptographic Modules. National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2019.
31. Almaraz Luengo, E.; Leiva, M.; García Villalba, L.J.; Hernandez-Castro, J.; Hurley-Smith, D. Critical Analysis of Hypothesis Tests in Federal Information Processing Standard (140-2). *Entropy* **2022**, *24*, 613. [CrossRef]
32. Almaraz Luengo, E.; Alaña, B.; García Villalba, L.J.; Hernandez-Castro, J. Weaknesses in ENT Battery Design. *Appl. Sci.* **2022**, *12*, 4230. [CrossRef]
33. Almaraz Luengo, E.; García Villalba, L.J. Recommendations on Statistical Randomness Test Batteries for Cryptographic Purposes. *ACM Comput. Surv.* **2021**, *54*, 1–34. [CrossRef]
34. Almaraz Luengo, E.; Leiva, M.; García Villalba, L.J.; Hurley-Smith, D.; Hernandez-Castro, J. Sensitivity and uniformity in statistical randomness tests. *J. Inf. Secur. Appl.* **2022**, *70*, 103322. [CrossRef]
35. Ryabko, B. Time-adaptive statistical test for random number generators. *Entropy* **2020**, *22*, 630. [CrossRef]
36. Simion, E. Entropy and Randomness: From Analogic to Quantum World. *IEEE Access* **2020**, *8*, 74553–74561. [CrossRef]
37. Demirhan, H.; Bitirim, N. Statistical Testing of Cryptographic Randomness. *J. Stat. Stat. Actuar. Sci.* **2016**, *9*, 1–11.
38. Crocetti, L.; Nannipieri, P.; Di Matteo, S.; Fanucci, L.; Saponara, S. Review of Methodologies and Metrics for Assessing the Quality of Random Number Generators. *Electronics* **2023**, *12*, 723. [CrossRef]
39. Dodis, Y.; Pointcheval, D.; Ruhault, S.; Vergniaud, D.; Wichs, D. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *CCS'13, Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, Berlin, Germany, 4–8 November 2013*; Association for Computing Machinery: New York, NY, USA, 2013; Volume 42, pp. 647–658.
40. Dodis, Y.; Shamir, A.; Stephens-Davidowitz, N.; Wichs, D. How to Eat Your Entropy and Have It Too-Optimal Recovery Strategies for Compromised RNGs. In *Advances in Cryptology-CRYPTO*; Lecture Notes in Computer Science; Garay, J.A., Gennaro, R.E., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; pp. 37–54.
41. Abdalla, M.; Belaïd, S.; Pointcheval, D.; Ruhault, S.; Vergnaud, D. Robust Pseudo-Random Number Generators with Input Secure Against Side-Channel Attacks, 2015. Cryptology ePrint Archive, Report 2015/1219. Available online: https://eprint.iacr.org/2015/1219 (accessed on 1 January 2022).
42. Kelsey, J.; Schneier, B.; Wagner, D.; Hall, C. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*; Lecture Notes in Computer Science; Vaudenay, S.E., Ed.; Springer: Berlin/Heidelberg, Germany, 1998; Volume 1372, pp. 168–188.
43. Kelsey, K.; Schneier, B.; Ferguson, N. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*; Lecture Notes in Computer Science; Heys, H., Adams, C.E., Eds.; Springer: Berlin/Heidelberg, Germany 1999; Volume 1758, pp. 13–33.
44. Shamir, A. On the generation of Cryptographically Strong Pseudorandom Sequences. *ACM Trans. Comput. Syst.* **1983**, *1*, 38–44. [CrossRef]
45. Rivest, R.; Shamir, A.; Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
46. Micali, S.; Schnorr, C.P. Efficient, perfect polynomial random number generator. *J. Cryptol.* **1991**, *3*, 157–172. [CrossRef]
47. Blum, L.; Blum, M.; Shub, M. A simple unpredictablepseudorandom numbergenerator. *SIAM J. Comput.* **1986**, *15*, 364–383. [CrossRef]
48. Anyanwu, N.; Deng, L.Y.; Dasgupta Dipankar, D. Design of Cryptographically Strong generator By Transforming Linearly Generated Sequences. *Int. J. Comput. Sci. Secur.* **2009**, *3*, 186–200.

49. Deng, L.Y.; Xu, H.Q. A System of High-dimensional, Efficient, Long-cycle and Portable Uniform Random Number Generators. *ACM Trans. Model. Comput. Simul.* **2003**, *13*, 299–309. [CrossRef]
50. Ãzkaynak, F. Cryptographically secure random number generator with chaotic additional input. *Nonlinear Dyn.* **2014**, *78*, 2015–2020. [CrossRef]
51. Vajargah, B.F.; Asghari, R. A Novel Pseudo-Random Number Generator for Cryptographic Applications. *Indian J. Sci. Technol.* **2016**, *9*, 1–5. [CrossRef]
52. Vajargah, B.F.; Asghari, R. A pseudo random number generator based on chaotic henon map (CHCG). *Int. J. Mechatron. Electr. Comput. Technol.* **2015**, *5*, 2120–2129.
53. Williams, B.; Hiromoto, R.E.; Carlson, A. A Design for a Cryptographically Secure Pseudo Random Number Generator. In Proceedings of the 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 18–21 September 2019; Volume 2, pp. 864–869. [CrossRef]
54. Sidorenko, A.; Schoenmakers, B. Concrete security of the Blum-Blum-Shub pseudorandom generator. In *Cryptography and Coding*; Lecture Notes in Computer Science; Smart, N.P., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3796, pp. 355–375.
55. Rousseau, G. On the Jacobi symbol. *J. Number Theory* **1994**, *48*, 109–111. [CrossRef]
56. Rabin, M.O. *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*; Technical Report; Massachusetts Institute of Technology: Cambridge, MA, USA, 1979.
57. Tibouchi, M. Security Reduction. *Encyclopedia of Cryptography and Security*; van Tilborg, H.C.A., Jajodia, S.E., Eds.; Springer: Boston, MA, USA, 2011; Volume 42, pp. 1167–1168.
58. Fischlin, R.; Schnorr, C.P. Stronger security proofs for RSA and Rabin bits. *J. Cryptol.* **2000**, *13*, 221–244. [CrossRef]
59. Vazirani, U.V.; Vazirani, V.V. Efficient and secure pseudo-random number generation. In *Advances in Cryptology—CRYPTO 1984, Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 19–22 August 1984*; Lecture Notes in Computer Science; Blakley, G.R., Chaum, D.E., Eds.; Springer: Berlin/Heidelberg, Germany, 1984; Volume 196, pp. 193–202.
60. Coppersmith, D.; Krawczyk, H.; Mansour, Y. The shrinking generator. In *Advances in Cryptology—CRYPTO'93, Proceedings of the 13th Annual International Cryptology Conference, Santa Barbara, CA, USA, 22–26 August 1993*; Stinson, D.R., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1994; Volume 773, pp. 22–39.
61. Meier, W.; Staffelbach, O. The self-shrinking generator. *Advances in Cryptology—EUROCRYPT'94, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, 9–12 May 1994*; Lecture Notes in Computer Science; De Santis, A.E., Ed.; Springer: Berlin/Heidelberg, Germany 1994; Volume 950, pp. 287–295.
62. Ganther, C.G. Alternating step generators controlled by de Bruijn sequences. In *Advances in Cryptology—EUROCRYPT'87, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques, Amsterdam, The Netherlands, 13–15 April 1987*; Lecture Notes in Computer Science; Chaum, D., Price, W.L.E., Eds.; Springer: Berlin/Heidelberg, Germany, 1988; Volume 304, pp. 5–14.
63. Melia-Segua, J.; Garca-Alfaro, J.; Herrera-Joancomarta, J. J3Gen: A PRNG for low-cost passive RFID. *Sensors* **2013**, *13*, 3816–3830. [CrossRef] [PubMed]
64. Zhang, H.; Wang, Y.; Wang, B.; Wu, X. Evolutionary random sequence generators based on LFSR. *Wuhan Univ. J. Nat. Sci.* **2007**, *12*, 75–78. [CrossRef]
65. Che, W.; Deng, H.; Tan, W.; Wang, J. A random number generator for application in RFID tags. In *Networked RFID Systems and Lightweight Cryptography: Raising Barriers to Product Counterfeiting*; Cole, P., Ranasinghe, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 279–287.
66. Röck, A. Pseudorandom Number Generators for Cryptographic Applications. Master's Thesis, Faculty of Natural Sciences, Paris-Lodron University, Salzburg, France, 2005.
67. *197*; Advanced Encryption Standard (AES). Federal Information Processing Standards Publication: Gaithersburg, MD, USA, 2001.
68. Daemen, J.; Rijmen, V. *The Design of Rijndael*; Springer: New York, NY, USA, 2002; Volume 2.
69. Rhee, M.Y. *Internet Security Cryptographic Principles, Algorithms and Protocols*; John Wiley & Sons: Hoboken, NJ, USA, 2003.
70. Coppersmith, D.; Johnson, D.B.; Matyas, S.M. A proposed mode for triple-DES encryption. *IBM J. Res. Dev.* **1995**, *40*, 253–262. [CrossRef]
71. Ferguson, N.; Schneier, B. *Practical Cryptography*; Practical Cryptography: Indianapolis, IN, USA, 2003; Volume 141, pp. 161–184.
72. Orue, A.B.; Montoya, F.; Hernandez Encinas, L. Trifork, a new pseudorandom number generator based on lagged Fibonacci maps. *J. Comput. Sci. Eng.* **2010**, *2*, 46–51.
73. Hu, Z.; Gnatyuk, S.; Okhrimenko, T.; Tynymbayev, S.; Iavich, M. High-Speed and Secure PRNG for Cryptographic Applications. *Int. J. Comput. Netw. Inf. Secur.* **2020**, *12*, 1–10. [CrossRef]
74. Ali, A.; Ali, E.; Ahsan Habib, M.; Nadim, M.; Kusaka, T.; Nogami, Y. Pseudo random ternary sequence and its autocorrelation property over finite field. *Int. J. Comput. Netw. Inf. Secur.* **2017**, *11*, 54–63. [CrossRef]