UNIVERSITY OF FRANCHE-COMTE

DOCTORAL SCHOOL SPIM

Fento-St

(DISC)

P H D T H E S I S

to obtain the title of

PhD of Science

of the University of Franche-Comte

Specialty : COMPUTER SCIENCE

Defended by

Xiaole FANG

USE OF CHAOTIC DYNAMICS FOR GENERATING PSEUDO-RANDOM NUMBERS IN DIFFERENT CONTEXTS

Thesis Advisor: Jacques BAHI and Laurent LARGER

defended on February 28, 2013

Jury :

Reviewers :	-
	-
Advisor :	-
President :	-
Examinators :	-
	-
	-
Invited :	-

Chapter 1

Introduction

Alone with the rapid development of Internet and universal application of multimedia technology, multimedia data including audio, image and video has been transmitted over insecure channels, it implies the need to protect data and privacy in digital world. This development has revealed new major security issues. For example, new security concerns have recently appeared because of the evolution of the Internet to support such activities as e-Voting, VoD, and digital rights management [66]. The random number generators (RNGs) are very important cryptographic primitive widely used in the Internet security, because they are fundamental in cryptosystems and information hiding schemes.

Random Number Generators (RNGs) are widely used in science and technology, it is a critical component in modern cryptographic systems, communication systems, statistical simulation systems and any scientific area incorporating Monte Carlo methods and many others [43, 50, 57]. The Random statistical quality of the generated bit sequence is measured by two aspects: the unpredictability of the bit stream and the speed at which the random bits can be produced. Other factors like system complexity, cost, reliability and so on, are also important for establishing successful RNGs. There are usually two methods for RNGs: one relates to deterministic algorithms implemented in hardware and software, the pseudo-random numbers are degenerated from a single 'seed', such generators are named pseudo-random number generators (PRNGs) [35]; another one counts on high entropy signals, whether from purely nondeterministic and stochastic physical phenomena, or from deterministic but chaotic dynamical systems (necessarily mixed with an unavoidable noisy and smaller compound) [60, 41]. A potential advantage of the latter physical high entropy signal, resides in its deterministic features which might be used to achieve chaos synchronization as it has been already demonstrated [46] and widely used for secure chaos communications [6]. However, synchronization possibility of the random binary sequence extracted from the chaotic physical signal is still an open problem, which resolution could lead to the efficient and practical use of the Vernam cypher.

For the PRNGs algorithms, it defined by a deterministic recurrent sequence in a finite state space, usually a finite field or ring, and an output function mapping each state to an input value. This is often either a real number in the interval $(0, 1)$ or an integer in some finite range [35]. It can be easily implemented in any computational platform, however, they suffer from the vulnerability that the future sequence can be deterministically computed if the seed or internal state of the algorithm is discovered. The main advantages of PRNGs is that no hardware cost is added and the speed is only counted on processing hardware. Its algorithms are developed to prevent guessing of the initial conditions, and the rate might be slowed down because of increased complexity of such algorithms.

Recently, some researchers have demonstrated the possibility to use chaotic dynamical systems as RNGs to reinforce the security of cryptographic algorithm, because the unpredictability and distort-like property of chaotic dynamical systems [27, 24, 47]. These attempts are due to the hypothesis that digital chaotic systems can possibly reinforce the security of cryptographic algorithms, because the behaviors of chaotic dynamical systems are very similar to those of physical noise sources [54]. Hence in [33], chaos has even been applied to strengthen some optical communications. Particularly, the random-like, unpredictable dynamics of chaotic systems, their inherent determinism and simplicity of realization suggest their potential for exploitation as RNGs.

In chaotic cryptography, there are two main design paradigms: in the first paradigm chaotic cryptosystems are realized in analog circuits (mainly based on chaos synchronization technique) [45], and in the second paradigm chaotic cryptosystems are realized in digital circuits or computers and do not depend on chaos synchronization technique. Generally speaking, synchronization based chaotic cryptosystems are generally designed for secure communications though noisy channels and cannot directly extended to design digital ciphers in pure cryptography. What's worse, many cryptanalytic works have shown that most synchronization based chaotic cryptosystems are not secure since it is possible to extract some information on secure chaotic parameters [18]. Therefore, although chaos synchronization is still actively studied in research of secure communications, the related ideas have less significance for conventional cryptographers. Since this dissertation is devoted to research lying between chaotic cryptography and traditional cryptography, only digital chaotic ciphers will be discussed in this dissertation.

However, even though chaotic systems exhibit random-like behavior, they are not necessarily cryptographically secure in their discretized form, see e.g. [29, 19]. The reason partly being that discretized chaotic functions do not automatically yield sufficiently complex behavior of the corresponding binary functions, which is a prerequisite for cryptographic security. It is therefore essential that the complexity of the binary functions is considered in the design phase such that necessary modifications can be made. Moreover, many suggested PRNG based on chaos suffer from reproducibility problems of the keystream due to the different handling of floating-point numbers on various processors, see e.g. [39].

Chaotic dynamical systems are usually continuous and hence defined on the real numbers domain. The transformation from real numbers to integers may lead to the loss of the chaotic behavior. The conversion to integers needs a rigorous theoretical foundation.

In this paper, some new chaotic pseudo-random bit generator is presented, which can also be used to obtain numbers uniformly distributed between 0 and 1. Indeed, these bits can be grouped n by n, to obtain the floating part of $x \in [0, 1]$ represented in binary numeral system. These generators are based on discrete chaotic iterations which satisfy Devaney's definition of chaos [28]. A rigorous framework is introduced, where topological chaotic properties of the generator are shown.

The design goal of these generators was to take advantage of the random-like properties of realvalued chaotic maps and, at the same time, secure optimal cryptographic properties. More precisely, the design was initiated by constructing a chaotic system on the integers domain instead of the real numbers domain.

The quality of a PRNG is proven both by theoretical foundations and empirical validations. Various statistical tests are available in the literature to check empirically the statistical quality of a given sequence. The most famous and important batteries of tests for evaluating PRNGs are: TestU01 [56], NIST (National Institute of Standards and Technology of the U.S. Government) and DieHARD suites [53, 37], and Comparative test parameters [40]. For various reasons, a generator can behave randomly according to some of these tests, but it can fail to pass some other tests. So to pass a number of tests as large as possible is important to improve the confidence put in the randomness of a given generator [59].

1.1 Research Background and Significance

1.2 Related work

In [28, 13], it is proven that chaotic iterations (CIs), a suitable tool for fast computing iterative algorithms, satisfies the topological chaotic property, as it is defined by Devaney [26]. Indeed, we have obtained this PRNG by combining chaotic iterations and two generators based on the logistic map in [64]. The resulted PRNG shows better statistical properties than each individual component alone. Additionally, various chaos properties have been established. The advantage of having such chaotic dynamics for PRNGs lies, among other things, in their unpredictability character. These chaos properties, inherited from chaotic iterations, are not possessed by the two inputted generators. We have shown that, in addition of being chaotic, this generator can pass the NIST battery of tests, widely considered as a comprehensive and stringent battery of tests for cryptographic applications [53].

Then, in the papers [14, 15], we have achieved to improve the speed of the former PRNG by replacing the two logistic maps: we used two XORshifts in [14], and ISAAC with XORshift in [15]. Additionally, we have shown that the first generator is able to pass DieHARD tests [14], whereas the second one can pass TestU01 [15].

In [63, 11], which is an extension of [64], we have improved the speed, security, and evaluation of the former generator and of its application in information hiding. Then, a comparative study between various generators is carried out and statistical results are improved. Chaotic properties, statistical tests, and security analysis allow us to consider that this kind of generator has better characteristics and is capable to withstand attacks.

In prior literature, the iterate function is just the vectorial boolean negation. It is then judicious to investigate whether other functions may replace the the vectorial boolean negation function in the above approach. In [8], we combined its own function and its own PRNGs to provide a new PRNG instance. and propose a method using Graph with strongly connected components as a selection criterion for chaotic iterate function. The approach developed along these lines solves this issue by providing a class of functions whose iterations are chaotic according to Devaney and such that resulting PRNG success statistical tests.

Then we use the vectorial Boolean negation as a prototype and explain how to modify this iteration function without deflating the good properties of the associated generator in [12]. Simulation results and basic security analysis are then presented to evaluate the

randomness of this new family of generators.

1.3 Thesis Goals

1.4 Thesis Organization

1.5 Abbreviations

Abbreviation	Definition
RNGs	Random Number Generators
TRNGs	True Random Number Generators
PRNG	Pseudo Random Number Generator
CSPRNG	Cryptographically Secure Pseudo Random Number Generator
NIST	National Institute of Standards and Technology
VOD	Video on Demand

1.6 Mathematical Symbols

Symbol	Meaning
$[\![1;\mathsf{N}]\!]$	$\rightarrow \{1, 2, \ldots, N\}$
S^n	$\rightarrow$ the n^{th} term of a sequence $S = (S^1, S^2, \ldots)$
v_i	$\rightarrow$ the i^{th} component of a vector: $v = (v_1, v_2, \ldots, v_n)$
f^k	$\rightarrow k^{th}$ composition of a function f
strategy	$\rightarrow$ a sequence which elements belong in $[\![1;\mathsf{N}]\!]$
mod	$\rightarrow$ a modulo or remainder operator
$\mathbb{S}$	$\rightarrow$ the set of all strategies
$\mathbf{C}_n^k$	$\rightarrow$ the binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
$\oplus$	$\rightarrow$ bitwise exclusive or
$+$	$\rightarrow$ the integer addition
$\ll$ and $\gg$	$\rightarrow$ the usual shift operators
$(\mathcal{X}, \mathrm{d})$	$\rightarrow$ a metric space
$\lfloor x \rfloor$	$\rightarrow$ returns the highest integer smaller than x
$n!$	$\rightarrow$ the factorial $n! = n \times (n-1) \times \cdots \times 1$
$\mathbb{N}^*$	$\rightarrow$ the set of positive integers {1,2,3,...}

Part I

First Part: Pseudo Random Number Generator Based on Chaotic Iteration

CHAPTER 2

General Notions

This chapter, serving as the background of this thesis, is devoted to basic notations and terminologies in the fields of random number and its classification.

2.1 Randomness

A random bit sequence could be interpreted as the result of the flips of an unbiased "fair" coin with sides that are labeled "0" and "1," with each flip having a probability of exactly $1/2$ of producing a "0" or "1." Furthermore, the flips are independent of each other: the result of any previous coin flip does not affect future coin flips. The unbiased "fair" coin is thus the perfect random bit stream generator, since the "0" and "1" values will be randomly distributed (and [0,1] uniformly distributed). All elements of the sequence are generated independently of each other, and the value of the next element in the sequence cannot be predicted, regardless of how many elements have already been produced. Obviously, the use of unbiased coins for cryptographic purposes is impractical. Nonetheless, the hypothetical output of such an idealized generator of a true random sequence serves as a benchmark for the evaluation of random and pseudorandom number generators. [53]

2.2 Types of Random Number Generators (RNGs)

A RNG is a computational or physical device designed to generate a sequence of numbers or symbols that lack any pattern, i.e. appear random.

It is always a difficult task to generate good random number/sequence. Although it is accepted that rolling a dice or drawing cards is random these mechanical methods are not practical. in the past, random numbers are usually generated offline based on some dedicated setup or devices, and the sequences are stored in table ready for use. These random tables are still available in the world-wide-web or some data CDROMs.

However, due to the online requirements and the security issues, random table becomes inappropriate, and hence different RNGs have been proposed, especially after the introduction of computer.

In general, RNGs can be grouped into two classes, namely true random number gernerators and pseudo random number generators, depending on their sources of randomness. RNGs can be classified as:

2.2.1 True Random Number Generators (TRNGs)

A TRNG is one which generates statistically independent and unbiased bits. These are also called as non-deterministic RNGs. In computing, a True random number generator is an apparatus that generates random numbers from a physical process. Such devices are often based on microscopic phenomena that generate a low-level, statistically random "noise" signal, such as thermal noise or the photoelectric effect or other quantum phenomena. These processes are, in theory, completely unpredictable, and the theory's assertions of unpredictability are subject to experimental test. A quantum-based hardware random number generator typically consists of a transducer to convert some aspect of the physical phenomena to an electrical signal, an amplifier and other electronic circuitry to bring the output of the transducer into the macroscopic realm, and some type of analog to digital converter to convert the output into a digital number, often a simple binary digit 0 or 1. By repeatedly sampling the randomly varying signal, a series of random numbers is obtained.

2.2.2 Pseudo Random Number Generators (PRNGs)

A pseudorandom number generator (PRNG), also known as a deterministic random bit generator (DRBG), [17] is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the PRNG's state. Although sequences that are closer to truly random can be generated using hardware random number generators, pseudorandom numbers are important in practice for simulations (e.g., of physical systems with the Monte Carlo method), and are central in the practice of cryptography and procedural generation. Common classes of these algorithms are linear congruential generators, Lagged Fibonacci generators, linear feedback shift registers, feedback with carry shift registers, and generalised feedback shift registers. Recent instances of pseudorandom algorithms include Blum Blum Shub, Fortuna, and the Mersenne twister.

2.3 Cryptographically secure pseudo random number generators

A PRNG suitable for cryptographic applications is called a cryptographically secure PRNG (CSPRNG). A requirement for a CSPRNG is that an adversary not knowing the seed has only negligible advantage in distinguishing the generator's output sequence from a random sequence. In other words, while a PRNG is only required to pass certain statistical tests, a CSPRNG must pass all statistical tests that are restricted to polynomial time in the size of the seed. Though such property cannot be proven, strong evidence may be provided by reducing the CSPRNG to a known hard problem in mathematics (e.g., integer factorization). In general, years of review may be required before an algorithm can be certified as a CSPRNG.

Some classes of CSPRNGs include the following:

- Stream ciphers

- Block ciphers running in counter or output feedback mode.
- PRNGs that have been designed specifically to be cryptographically secure, such as Microsoft's Cryptographic Application Programming Interface function CryptGenRandom, the Yarrow algorithm (incorporated in Mac OS X and FreeBSD), and Fortuna.
- Combination PRNGs which attempt to combine several PRNG primitive algorithms with the goal of removing any non-randomness.
- Special designs based on mathematical hardness assumptions. Examples include Micali-Schnorr and the Blum Blum Shub algorithm, which provide a strong security proof. Such algorithms are rather slow compared to traditional constructions, and impractical for many applications.

A stream cipher is a cryptographic technique that encrypts binary digits individually, using a transformation that changes with time. This is contrasted to a block cipher, where a block of binary data is encrypted simultaneously, with the transformation usually being constant for each block.

In specific applications, stream ciphers are more appropriate than block ciphers [48, 52]:

1. Stream ciphers are generally faster than block ciphers, especially in hardware.
2. Stream ciphers have less hardware complexity and less memory requirements for both hardware and software.
3. Stream ciphers process the plaintext character by character, so no buffering is required to accumulate a full plaintext block (unlike block ciphers).
4. Synchronous stream ciphers have no error propagation.

2.4 Stream Cipher

A stream cipher generates successive elements of the keystream based on an internal state. This state is updated in essentially two ways: if the state changes independently of the plaintext or ciphertext messages, the cipher is classified as a synchronous stream cipher. By contrast, self-synchronising stream ciphers update their state based on previous ciphertext digits.

2.4.1 One-Time Pad (Vernam Cipher)

In modern terminology, a Vernam cipher is a stream cipher in which the plaintext is XORed with a random or pseudorandom stream of data (the keystream) of the same length to generate the ciphertext. If the keystream is truly random and used only once, this is effectively a one-time pad.

Shannon [55] showed that the one-time pad provides perfect security. This means that the conditional entropy of the message M knowing the ciphertext C is the same as the entropy of the original message, i.e. $H(M|C) = H(M)$. He also showed that the one-time pad is optimal in the sense that the previous conditions cannot be achieved with a key of size smaller than the message.

The problem of the one-time pad is that we first have to agree on a key of the same length as the message. For most applications this is not practical. The next two schemes try to produce a "random looking" keystream from a short key and IV. By random looking, we mean that we cannot distinguish the keystream from a random sequence in a complexity less than trying all possible keys.

2.4.2 Synchronous stream ciphers

In a synchronous stream cipher a stream of pseudo-random digits is generated independently of the plaintext and ciphertext messages, and then combined with the plaintext (to encrypt) or the ciphertext (to decrypt). In the most common form, binary digits are used (bits), and the keystream is combined with the plaintext using the exclusive or operation (XOR). This is termed a binary additive stream cipher.

In a synchronous stream cipher, the sender and receiver must be exactly in step for decryption to be successful. If digits are added or removed from the message during transmission, synchronisation is lost. To restore synchronisation, various offsets can be tried systematically to obtain the correct decryption. Another approach is to tag the ciphertext with markers at regular points in the output.

If, however, a digit is corrupted in transmission, rather than added or lost, only a single digit in the plaintext is affected and the error does not propagate to other parts of the message. This property is useful when the transmission error rate is high; however, it makes it less likely the error would be detected without further mechanisms. Moreover, because of this property, synchronous stream ciphers are very susceptible to active attacks – if an attacker can change a digit in the ciphertext, he might be able to make predictable changes to the corresponding plaintext bit; for example, flipping a bit in the ciphertext causes the same bit to be flipped in the plaintext.

2.4.3 Self-synchronizing stream ciphers

Another approach uses several of the previous N ciphertext digits to compute the keystream. Such schemes are known as self-synchronizing stream ciphers, asynchronous stream ciphers or ciphertext autokey (CTAK). The idea of self-synchronization was patented in 1946, and has the advantage that the receiver will automatically synchronise with the keystream generator after receiving N ciphertext digits, making it easier to recover if digits are dropped or added to the message stream. Single-digit errors are limited in their effect, affecting only up to N plaintext digits.

An example of a self-synchronising stream cipher is a block cipher in cipher feedback (CFB) mode.

2.5 Chaos-based random number generators

Since the seventies, the use of chaotic dynamics for the generation of random sequences and cryptographical applications has raised a lot of interests. It is clearly pointed out by some researchers that there exists a close relationship between chaos and cryptography, and many research works have been witnessed in the last two decades.

chaotic dynamics are usually studied in two different domains, the continuous time domain where the dynamics are generated from a chaotic system specified in differential equations, or a chaotic map quoted with recurrence relationship in the discrete time domain.

chaos possesses several distinct propertie, including sensitivity to initial conditions, ergodicity and wide band spectrum. contributing its unpredictable and random manner in practice. Although it is still controversy to equate these properties with randomness and claim a chaos-based random number generator to be good enough, a lot of designs and applications, in particularly, related with the secure communications have been proposed.

It is common to use a chaotic map for pseudo-random number generation. Due to the recent design of electonic circuits for the realization fo chaotic systems, it is also possible to generate the bit sequence by observing such dynamics, as a replacement of those physical random sources.

2.6 Continuous Chaos in Digital Computers

In the past two decades, the use of chaotic systems in the design of cryptosystems, PRNG, and hash functions, has become more and more frequent. Generally speaking, the chaos theory in the continuous field is used to analyze performances of related systems.

However, when chaotic systems are realized in digital computers with finite computing precisions, it is doubtful whether or not they can still preserve the desired dynamics of the continuous chaotic systems. Because most dynamical properties of chaos are meaningful only when dynamical systems evolve in the continuous phase space, these properties may become meaningless or ambiguous when the phase space is highly quantized (i.e., latticed) with a finite computing precision (in other words, dynamical degradation of continuous chaotic systems realized in finite computing precision).

The quantization errors, which are introduced into iterations of digital chaotic systems for every iteration, will make pseudo orbits depart from real ones with very complex and uncontrolled manners. Because of the sensitivity of chaotic systems on initial conditions, even ”trivial” changes of computer arithmetic can definitely change pseudo orbits’ structures.

Although all quantization errors are absolutely deterministic when the finite precision and the arithmetic are fixed, it is technically impossible to know and deal with all errors in digital iterations. Some random perturbation models have been proposed to depict quantization errors in digital chaotic systems, but they cannot exactly predict the actual dynamics of studied digital chaotic systems and has been criticized because of their essentially deficiencies

When chaotic systems are realized in finite precision, their dynamical properties will

be deeply different from the properties of continuous-value systems and some dynamical degradation will arise, such as short cycle length and decayed distribution. This phenomenon has been reported and analyzed in various situations [20, 65, 44, 21, 36].

Therefore, continuous chaos may collapse into the digital world and the ideal way to generate pseudo-random sequences is to use Chaotic iterations.

2.7 Chaos for Discrete Dynamical Systems

Consider a metric space $(\mathcal{X}, d)$ and a continuous function $f : \mathcal{X} \longrightarrow \mathcal{X}$, for one-dimensional dynamical systems of the form:

$$x^0 \in \mathcal{X} \text{ and } \forall n \in \mathbb{N}^*, x^n = f(x^{n-1}), \tag{1}$$

the following definition of chaotic behavior, formulated by Devaney [26], is widely accepted:

Definition 1 A dynamical system of form 1 is said to be chaotic if the following conditions hold.

- Topological transitivity:

$$\forall U, V \text{ open sets of } \mathcal{X},\ \exists k > 0, f^k(U) \cap V \neq \varnothing \tag{2}$$

  Intuitively, a topologically transitive map has points which eventually move under iteration from one arbitrarily small neighborhood to any other. Consequently, the dynamical system can not be decomposed into two disjoint open sets which are invariant under the map. Note that if a map possesses a dense orbit, then it is clearly topologically transitive.

- Density of periodic points in $\mathcal{X}$:

  Let $P = \{p \in \mathcal{X} | \exists n \in \mathbb{N}^* : f^n(p) = p\}$ the set of periodic points of f. Then P is dense in $\mathcal{X}$:

$$\overline{P} = \mathcal{X} \tag{3}$$

  Intuitively, Density of periodic orbits means that every point in the space is approached arbitrarily closely by periodic orbits. Topologically mixing systems failing this condition may not display sensitivity to initial conditions, and hence may not be chaotic.

- Sensitive dependence on initial conditions:

  $\exists \varepsilon > 0, \forall x \in \mathcal{X}, \forall \delta > 0, \exists y \in \mathcal{X}, \exists n \in \mathbb{N}, d(x, y) < \delta$ and $d\left(f^n(x), f^n(y)\right) \geqslant \varepsilon$.

  Intuitively, a map possesses sensitive dependence on initial conditions if there exist points arbitrarily close to x which eventually separate from x by at least ε under iteration of f. Not all points near x need eventually separate from x under iteration, but

there must be at least one such point in every neighborhood of x. If a map possesses sensitive dependence on initial conditions, then for all practical purposes, the dynamics of the map defy numerical computation. Small errors in computation which are introduced by round-off may become magnified upon iteration. The results of numerical computation of an orbit, no matter how accurate, may bear no resemblance whatsoever with the real orbit.

When f is chaotic, then the system $(\mathcal{X}, f)$ is chaotic and quoting Devaney: "it is unpredictable because of the sensitive dependence on initial conditions. It cannot be broken down or decomposed into two subsystems which do not interact because of topological transitivity. And, in the midst of this random behavior, we nevertheless have an element of regularity." Fundamentally different behaviors are consequently possible and occur in an unpredictable way.

2.8 Chaotic iterations

Definition 2 The set $\mathbb{B}$ denoting $\{0, 1\}$, let $f : \mathbb{B}^{\mathsf{N}} \longrightarrow \mathbb{B}^{\mathsf{N}}$ be an "iteration" function and $S \in \mathbb{S}$ be a chaotic strategy. Then, the so-called *chaotic iterations* are defined by [51]:

$$\left\{ \begin{array}{l} x^0 \in \mathbb{B}^{\mathsf{N}}, \\ \forall n \in \mathbb{N}^*, \forall i \in [\![1; \mathsf{N}]\!], x_i^n = \left\{ \begin{array}{ll} x_i^{n-1} & \text{if } S^n \neq i \\ f(x^{n-1})_{S^n} & \text{if } S^n = i. \end{array} \right. \end{array} \right. \tag{4}$$

In other words, at the n^{th} iteration, only the S^n-th cell is "iterated". Note that in a more general formulation, S^n can be a subset of components and $f(x^{n-1})_{S^n}$ can be replaced by $f(x^k)_{S^n}$, where $k < n$, describing for example delays transmission (see *e.g.* [7]). For the general definition of such chaotic iterations, see, e.g. [51].

Chaotic iterations generate a set of vectors (boolean vector in this paper), they are defined by an initial state x^0, an iteration function f, and a chaotic strategy S. The next subsection gives the outline proof that chaotic iterations satisfy Devaney's topological chaos property. Thus they can be used to define a new pseudo-random bit generator.

2.9 The generation of pseudorandom sequence

2.9.1 Blum Blum Shub

Blum Blum Shub generator [22] (usually denoted by BBS) takes the form:

$$x_{n+1} = x_n^2 \; mod \; m \tag{5}$$

where m is the product of two prime numbers (these prime numbers need to be congruent to 3 modulus 4).

2.9.2 The logistic map

The logistic map, given by:

$$x^{n+1} = \mu\, x^n(1 - x^n), \text{ with } x^0 \in (0, 1), \mu \in (3.99996, 4],$$

was originally introduced as a demographic model by Pierre François Verhulst in 1838. In 1947, Ulam and Von Neumann [61] studied it as a PRNG. This essentially requires mapping the states of the system $(x^n)_{n\in\mathbb{N}}$ to $\{0, 1\}^{\mathbb{N}}$. A simple way for turning x^n to a discrete bit symbol r is by using a threshold function as it is shown in Algorithm 1. A second usual way to obtain an integer sequence from a real system is to chop off the leading bits after moving the decimal point of each x to the right, as it is obtained in Algorithm 2.

Algorithm 1 An arbitrary round of logistic map 1

Input: the internal state x (a decimal number)
Output: r (a 1-bit word)

1: $x \leftarrow 4x(1 - x)$
2: **if** $x < 0$ **then**
3: $\quad r \leftarrow 0$;
4: **else**
5: $\quad r \leftarrow 1$;
6: return r

Algorithm 2 An arbitrary round of logistic map 2

Input: the internal state x (a decimal number)
Output: r (an integer)

1: $x \leftarrow 4x(1 - x)$
2: $r \leftarrow \lfloor 10000000x \rfloor$
3: return r

2.9.3 XORshift

XORshift is a category of very fast PRNGs designed by George Marsaglia [38]. It repeatedly uses the transform of *exclusive or* (XOR) on a number with a bit shifted version of it. The state of a XORshift generator is a vector of bits. At each step, the next state is obtained by applying a given number of XORshift operations to w-bit blocks in the current state, where $w = 32$ or 64. A XORshift operation is defined as follows. Replace the w-bit block by a bitwise XOR of the original block, with a shifted copy of itself by a positions either to the right or to the left, where $0 < a < w$. This Algorithm 3 has a period of $2^{32} - 1 = 4.29 \times 10^9$.

Algorithm 3 An arbitrary round of XORshift algorithm

Input: the internal state z (a 32-bits word)

Output: y (a 32-bits word)

1: $z \leftarrow z \oplus (z \ll 13)$;
2: $z \leftarrow z \oplus (z \gg 17)$;
3: $z \leftarrow z \oplus (z \ll 5)$;
4: $y \leftarrow z$;
5: return y

2.9.4 ISAAC

ISAAC is an array-based PRNG and a stream cipher designed by Robert Jenkins (1996) to be cryptographically secure [30]. The name is an acronym for Indirection, Shift, Accumulate, Add and Count. The ISAAC algorithm has similarities with RC4. It uses an array of 256 32-bit integers as the internal state, writing the results to another 256-integer array, from which they are read one at a time until empty, at which point they are recomputed. Since it only takes about 19 32-bit operations for each 32-bit output word, it is extremely fast on 32-bit computers.

We give the key-stream procedure of ISAAC in Algorithm 4. The internal state is x, the output array is r, and the inputs a, b, and c are those computed in the previous round. The value $f(a, i)$ in Table 4 is a 32-bit word, defined for all a and $i \in \{0, \ldots, 255\}$ as:

$$f(a,i) = \begin{cases} a \ll 13 & \text{if } i \equiv 0 \ mod\ 4, \\ a \gg 6 & \text{if } i \equiv 1 \ mod\ 4, \\ a \ll 2 & \text{if } i \equiv 2 \ mod\ 4, \\ a \gg 16 & \text{if } i \equiv 3 \ mod\ 4. \end{cases} \tag{6}$$

Algorithm 4 An arbitrary round of ISAAC algorithm

Input: a, b, c, and the internal state x

Output: an array r of 256 32-bit words

1: $c \leftarrow c + 1$;
2: $b \leftarrow b + c$;
3: **while** $i = 0, \ldots, 255$ **do**
4: $\quad s \leftarrow x_i$;
5: $\quad a \leftarrow f(a, i) + x_{(i+128)\ mod\ 256}$;
6: $\quad x_i \leftarrow a + b + x_{(x \gg 2)\ mod\ 256}$;
7: $\quad r_i \leftarrow s + x_{(x_i \gg 10)\ mod\ 256}$;
8: $\quad b \leftarrow r_i$;
9: return r

2.10 Version 1 CI algorithms

2.10.1 Chaotic iterations as PRNG

Our generator denoted by CI(PRNG1,PRNG2) is designed by the following process.

Let $\mathsf{N} \in \mathbb{N}^*, \mathsf{N} \geqslant 2$. Some chaotic iterations are fulfilled to generate a sequence $(x^n)_{n\in\mathbb{N}} \in \left(\mathbb{B}^{\mathsf{N}}\right)^{\mathbb{N}}$ of boolean vectors: the successive states of the iterated system. Some of these vectors are randomly extracted and their components constitute our pseudorandom bit flow.

Algorithm 5 An arbitrary round of the Version 1 CI generator

Input: the internal state x (an array of N 1-bit words)
Output: an array r of N 1-bit words

1: $a \leftarrow PRNG1()$;
2: $m \leftarrow a \; mod \; 2 + c$;
3: **while** $i = 0, \ldots, m$ **do**
4: $\quad b \leftarrow PRNG2()$;
5: $\quad S \leftarrow b \; mod \; \mathsf{N}$;
6: $\quad x_S \leftarrow \overline{x_S}$;
7: $r \leftarrow x$;
8: return r;

Chaotic iterations are realized as follows. Initial state $x^0 \in \mathbb{B}^{\mathsf{N}}$ is a boolean vector taken as a seed and chaotic strategy $(S^n)_{n\in\mathbb{N}} \in [\![1, \mathsf{N}]\!]^{\mathbb{N}}$ is constructed with PRNG2. Lastly, iterate function f is the vectorial boolean negation

$$f_0 : (x_1, ..., x_{\mathsf{N}}) \in \mathbb{B}^{\mathsf{N}} \longmapsto (\overline{x_1}, ..., \overline{x_{\mathsf{N}}}) \in \mathbb{B}^{\mathsf{N}}.$$

To sum up, at each iteration only S^i-th component of state X^n is updated, as follows

$$x_i^n = \begin{cases} x_i^{n-1} & \text{if } i \neq S^i, \\ \overline{x_i^{n-1}} & \text{if } i = S^i. \end{cases} \tag{7}$$

Finally, let $\mathcal{M}$ be a finite subset of $\mathbb{N}^*$. Some x^n are selected by a sequence m^n as the pseudorandom bit sequence of our generator, $(m^n)_{n\in\mathbb{N}} \in \mathcal{M}^{\mathbb{N}}$. So, the generator returns the following values: the components of x^{m^0}, followed by the components of $x^{m^0+m^1}$, followed by the components of $x^{m^0+m^1+m^2}$, *etc.* In other words, the generator returns the following bits:

$$x_1^{m_0} x_2^{m_0} x_3^{m_0} \ldots x_{\mathsf{N}}^{m_0} x_1^{m_0+m_1} x_2^{m_0+m_1} \ldots x_{\mathsf{N}}^{m_0+m_1} x_1^{m_0+m_1+m_2} x_2^{m_0+m_1+m_2} \ldots$$

or the following integers:

$$x^{m_0} x^{m_0+m_1} x^{m_0+m_1+m_2} \ldots$$

2.10.2 Version 1 CI PRNGs Algorithm

The basic design procedure of the novel generator is summed up in Algorithm 5. The internal state is x, the output array is r. a and b are those computed by PRNG1 and PRNG2. Lastly, k and N are constants and $\mathcal{M} = \{\mathrm{k}, \mathrm{k}+1\}$ ($k \geqslant 3\mathsf{N}$ is recommended).

CHAPTER 3

The Design And Development Of CIPRNG

In this chapter, some studies for CIPRNG algorithm are given. First of all, some researches of Version 1 CI are deepen. Then the designs of our three new versions of CI pseudo random number generators based on discrete chaotic iterations, satisfying Devaney's chaos, are proposed and discussed. Detail operations of this approach are described in this chapter, while their performance and a comparative study will be presented latter.

3.1 Development of Version 1 CI Algorithm

3.1.1 On the periodicity of chaotic orbit

Since chaotic iterations are constrained in a discrete space with 2^N elements, it is obvious that every chaotic orbit will eventually be periodic, i.e., finally goes to a cycle with limited length not greater than 2^N.

The schematic view of a typical orbit of a digital chaotic system is shown in Figure 3.1. Generally speaking, each digital chaotic orbit includes two connected parts: $x^0, x^1, \ldots, x^{l-1}$ and $x^l, x^{l+1}, \ldots, x^{l+n}$, which are respectively called transient (branch) and cycle. Accordingly, l and $n+1$ are respectively called transient length and cycle period, and $l+n$ is called orbit length. Thus,

Definition 3 A sequence $x = (x^1, ..., x^n)$ is said to be cyclic if a subset of successive terms is repeated from a given rank, until the end of x.

This generator based on discrete chaotic iterations generated by two pseudorandom sequences (m and w) has a long cycle length. If the cycle period of m and w are respectivelly

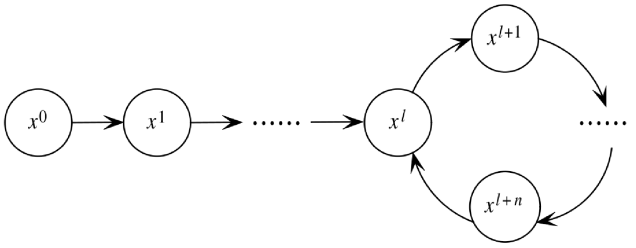


Figure 3.1: A pseudo orbit of a digital chaotic system

Table 3.1: Ideal cycle period

PRNG		Ideal cycle period
Logistic map		∞
XORshift		$2^{32}-1$
ISAAC		2^{8295}
Version 1 CI algorithms	**Logistic map 1+Logistic map 2**	∞
	XORshift+XORshift	2^{65}
	XORshift+ISAAC	2^{8328}
	ISAAC+ISAAC	2^{16591}

n_m and n_w, then in an ideal situation, the cycle period of the novel sequence is $n_m \times n_w \times 2$ (because $\bar{\bar{x}} = x$). Table 3.1 gives the ideal cycle period of various generators.

- m ($n_m = 2$): 12121212121212121212121212...
- w ($n_w = 4$): 1 23 4 12 3 41 2 34 1 23 4 12 3 41 2 34 1 23 4...
- x ($n_x = 2\times4\times2 = 16$): 0000(0) 1000(8) 1110(14) 1111(15) 0011(3) 0001(1) 1000(8) 1100(12) 1111(15) 0111(7) 0001(1) 0000(0) 1100(12) 1110(14) 0111(7) 0011(3) 0000(0) 1000(8) 1110(14) 1111(15) 0011(3) 0001(1) 1000(8) 1100(12) 1111(15) 0111(7) 0001(1) 0000(0) 1100(12) 1110(14) 0111(7) 0011(3)...

3.1.2 Security Analysis

In this section the concatenation of two strings u and v is classically denoted by uv. In a cryptographic context, a pseudo random generator is a deterministic algorithm G transforming strings into strings and such that, for any seed s of length m, $G(s)$ (the output of G on the input s) has size $l_G(m)$ with $l_G(m) > m$. The notion of secure PRNGs can now be defined as follows.

3.1.2.1 Algorithm expression conversion

For the convenience of security analysis, Version 1 CI Algorithm 5 is converted as Equation 1, internal state is x, S and T are those computed by PRNG1 and PRNG2, each round, x^{n-1} is updated to be x^n.

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}}-1]\!], S \in [\![0, 2^{\mathsf{N}}-1]\!]^{\mathbb{N}}, T \in [\![0, 2^{\mathsf{N}}-1]\!]^{\mathbb{N}} \\ C = S^n \& 1 + 3 * N \\ w^0 = T^m \bmod N, w^1 = T^{m+1}\&3, ...w^{C-1} = T^{m+C-1}\&3 \\ d^n = (1 \ll w^0) \oplus (1 \ll w^1) \oplus ...(1 \ll w^{C-1}) \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus d^n, \end{cases} \tag{1}$$

3.1.2.2 Proof

Definition 4 A cryptographic PRNG G is secure if for any probabilistic polynomial time algorithm D, for any positive polynomial p, and for all sufficiently large m's,

$$|Pr[D(G(U_m)) = 1] - Pr[D(U_{l_G(m)}) = 1] < \frac{1}{p(m)}, \tag{2}$$

where U_r is the uniform distribution over $0, 1^r$ and the probabilities are taken over U_m, $U_{l_G(m)}$ as well as over the internal coin tosses of D.

Intuitively, it means that there is no polynomial time algorithm that can distinguish a perfect uniform random generator from G with a non negligible probability. Note that it is quite easily possible to change the function l into any polynomial function l' satisfying $l'(m) > m$.

The generation schema developed in Algorithm 1 is based on 2 pseudo random generators. Let H be the "PRNG1" and I be the "PRNG2". We may assume, without loss of generality, that for any string S_0 of size L, the size of $H(S_0)$ is kL, then for any string T_0 of size M, it has $I(T_0)$ with kN, with $k > 2$. It means that $l_H(N) = kL$ and $l_I(N) = kM$. Let $S_1, ..., S_k$ be the string of length L such that $H(S_0) = S_1...S_k$ and $T_1, ..., T_k$ be the string of length M that $H(S_0) = T_1...T_k$ ($H(S_0)$ and $I(T_0)$ are the concatenations of S_i's and T_i's). The cryptographic PRNG X defined in Algorithm 1 is algorithm mapping any string of length $L + M + N$ $x_0S_0T_0$ into the string $x_0 \oplus d^1, x_0 \oplus d^1 \oplus d^2, ...(x_0 \bigoplus_{i=0}^{i=k} d^i)$ (Equation 1). One in particular has $l_X(L + M + N) = kN = l_H(N)$ and $k > M + L + N$. We announce that if one PRNG of H is secure, then the new one from Equation 1 is secure too.

Proposition 1 *If one of H is a secure cryptographic PRNG, then X is a secure cryptographic PRNG too.*

Proof The proposition is proven by contraposition. Assume that X is not secure. By Definition, there exists a polynomial time probabilistic algorithm D, a positive polynomial p, such that for all k_0 there exists $L + M + N \geq k_0$ satisfying

$$|\Pr[D(X(U_{L+M+N})) = 1] - \Pr[D(U_{kN} = 1]| \geq \frac{1}{p(L + M + N)}.$$

Define there is a w of size kL.

1. Decompose w into $w = w_1...w_k$.
2. Pick a string y of size N uniformly at random.
3. Pick a string of size $(3kN + \sum_{j=1}^{j=k}(w_j\&1))M$: u.
4. Decompose u into $u = u_1...u_{3kN+\sum_{j=1}^{j=k}(w_j\&1)}$.
5. Define $t_i = (\bigoplus_{l=3N(i-1)+(\sum_{l=1}^{l=i-1}(w_j\&1))+1}^{j=3N(i)+(\sum_{j=1}^{j=i}(w_j\&1))}(1 << u_l))$.

6. Compute $z = (y \oplus t_1)(y \oplus t_1 \oplus t_2)...(y \bigoplus_{i=1}^{i=k}(t_i))$.

7. Return $D(z)$.

On one hand, consider for each $y \in \mathbb{B}^{kN}$ the function φ_y from $\mathbb{B}^{kN}$ into $\mathbb{B}^{kN}$ mapping $t = t_1 \dots t_k$ (each t_i has length N) to $(y \oplus t_1)(y \oplus t_1 \oplus t_2) \dots (y \bigoplus_{i=1}^{i=k} t_i)$. On the other hand, treat each $u_l \in \mathbb{B}^{(3Nk+\sum_{j=0}^{j=k}(w_j\&1))M}$ the function ϕ_u from $\mathbb{B}^{(3kN+\sum_{j=0}^{j=k}(w_i\&1))M}$ into $mathbbB^{kN}$ mapping $w = w_1 \dots w_k$ (each w_i has length L) to $(\bigoplus_{l=1}^{l=3N+(w_1\&1)}(1 << u_l))((\bigoplus_{l=1+3N+(w_1\&1)}^{l=6N+(w_1\&1)+(w_1\&1)}(1 << u_l)) \dots (\bigoplus_{l=3N(k-1)+\sum_{j=1}^{j=k-1}(w_j\&1)}^{l=3Nk+\sum_{j=1}^{j=k}(w_j\&1)}(1 << u_l)$ By construction, one has for every w,

$$D'(w) = D(\varphi_y(\phi_u(w))), \tag{3}$$

Therefore, and using (3), one has $\Pr[D'(U_{kL}) = 1] = \Pr[D(\varphi_y(\phi_u(U_{kL}))) = 1]$ and, therefore,

$$\Pr[D'(U_{kL}) = 1] = \Pr[D(U_{kN}) = 1]. \tag{4}$$

Now, using (3) again, one has for every x,

$$\Pr[D'(U_{H(x)}) = 1] = \Pr[D(\varphi_y(\phi_u(U_{H(x)}))) = 1] \tag{5}$$

since where y and u_j are randomly generated.
By construction, $\varphi_y(\phi_u(x)) = X(yu_1w)$, hence

$$\Pr[D'(H(U_{kL})) = 1] = \Pr[D(X(U_{N+M+L})) = 1] \tag{6}$$

Compute the difference of Equation (6) and (5), one can deduce that there exists a polynomial time probabilistic algorithm D', a positive polynomial p, such that for all k_0 there exists $L + M + N \geq k_0$ satisfying

$$|\Pr[D'(H(U_{KL})) = 1] - \Pr[D(U_{kL}) = 1]| \geq \frac{1}{p(L + M + N)},$$

proving that H is not secure, which is a contradiction to the first place that one of them is cryptographic secure.

3.1.3 An Efficient, Cryptographically Secure, PRNG Based On Version 1 CI

In Table 3.2, an efficient, based on Version 1 CI, good random quality, and cryptographically secure PRNG algorithm is given.

The internal state x is defined as size of 12 bits, three 32-bit XORshift PRNGs ($xorshift1(), xorshift2(), xorshift3()$) are applied, each of them is split to 16 2-bit binary blocks (value is between 0 to 3), then the 3 LSBs (least significant bits) of the output from BBS $bbs()$ are decide 12 or 13 these blocks used to update the state. According to Section 3.1.2, this generator based on Version 1 CI can turn to be cryptographically secure if its I used is cryptographically secure, here I is chosen as BBS PRNG, which is believed to be the most secure PRNG method available [62], the t computed by BBS $bbs()$ is based on 32 bit m (Equation 5), the $log(log(m))$ LSBs of t can be treated as secure, hence, here 3LSBs chosen is qualified.

Input: x (a 12-bit word)
Output: r (a 12-bit word)
$x1 \leftarrow xorshift1()$;
$x2 \leftarrow xorshift2()$;
$x3 \leftarrow xorshift3()$;
$t \leftarrow bbs()$;
$t1 \leftarrow t\&1$;
$t2 \leftarrow t\&2$;
$t3 \leftarrow t\&4$;
$w1 \leftarrow 0$;
$w2 \leftarrow 0$;
$w3 \leftarrow 0$;
While $i = 1...12$
$w1 \leftarrow (w1 \oplus (1 \ll ((x1 \gg (i \times 2))\&3)))$;
$w2 \leftarrow (w2 \oplus (1 \ll ((x2 \gg (i \times 2))\&3)))$;
$w3 \leftarrow (w3 \oplus (1 \ll ((x3 \gg (i \times 2))\&3)))$;
EndWhile
if $(t1 \neq 0)$ **then** $w1 \leftarrow (w1 \oplus (1 \ll ((x1 \gg 26)\&3)))$;
if $(t2 \neq 0)$ **then** $w2 \leftarrow (w2 \oplus (1 \ll ((x2 \gg 26)\&3)))$;
if $(t3 \neq 0)$ **then** $w3 \leftarrow (w3 \oplus (1 \ll ((x3 \gg 26)\&3)))$;
$x \leftarrow x \oplus w1 \oplus (w2 \ll 4) \oplus (w3 \ll 8)$;
$r \leftarrow x$;
return r;
An arbitrary round of the algorithm

Table 3.2: An Efficient, Cryptographically Secure, PRNG Based On Version 1 CI

3.2 Version 2 CIPRNG Algorithm

3.2.1 Presentation

The CI generator (generator based on chaotic iterations) is designed by the following process. First of all, some chaotic iterations have to be done to generate a sequence $(x^n)_{n\in\mathbb{N}} \in \left(\mathbb{B}^{\mathsf{N}}\right)^{\mathbb{N}}$ ($\mathsf{N} \in \mathbb{N}^*, \mathsf{N} \geqslant 2$, N is not necessarily equal to 32) of boolean vectors, which are the successive states of the iterated system. Some of these vectors will be randomly extracted and our pseudo-random bit flow will be constituted by their components. Such chaotic iterations are realized as follows. Initial state $x^0 \in \mathbb{B}^{\mathsf{N}}$ is a boolean vector taken as a seed (see Section 3.2.2) and chaotic strategy $(S^n)_{n\in\mathbb{N}} \in [\![1, \mathsf{N}]\!]^{\mathbb{N}}$ is an irregular decimation of a random number sequence (Section 3.2.4). The iterate function f is the vectorial boolean negation:

$$f_0 : (x_1, ..., x_{\mathsf{N}}) \in \mathbb{B}^{\mathsf{N}} \longmapsto (\overline{x_1}, ..., \overline{x_{\mathsf{N}}}) \in \mathbb{B}^{\mathsf{N}}.$$

At each iteration, only the S^i-th component of state x^n is updated, as follows: $x_i^n = x_i^{n-1}$ if $i \neq S^i$, else $x_i^n = \overline{x_i^{n-1}}$. Finally, some x^n are selected by a sequence m^n as the pseudo-random bit sequence of our generator. $(m^n)_{n\in\mathbb{N}} \in \mathcal{M}^{\mathbb{N}}$ is computed from a PRNG, such as XORshift sequence $(y^n)_{n\in\mathbb{N}} \in [\![0, 2^{32} - 1]\!]$ (see Section 3.2.3). So, the generator returns the following values:
Bits:

$$x_1^{m_0} x_2^{m_0} x_3^{m_0} \dots x_{\mathsf{N}}^{m_0} x_1^{m_0+m_1} x_2^{m_0+m_1} \dots x_{\mathsf{N}}^{m_0+m_1} x_1^{m_0+m_1+m_2} \dots$$

or States:

$$x^{m_0} x^{m_0+m_1} x^{m_0+m_1+m_2} \dots$$

3.2.2 The seed

The unpredictability of random sequences is established using a random seed that is obtained by a physical source like timings of keystrokes. Without the seed, the attacker must not be able to make any predictions about the output bits, even when all details of the generator are known [59].

The initial state of the system x^0 and the first term y^0 of the input PRNG are seeded either by the current time in seconds since the Epoch, or by a number that the user inputs. Different ways are possible. For example, let us denote by t the decimal part of the current time. So x^0 can be $t \pmod{2^N}$ written in binary digits and $y^0 = t$.

3.2.3 Sequence m of returned states

The output of the sequence (y^n) is uniform in $[\![0, 2^{32} - 1]\!]$. However, we do not want the output of (m^n) to be uniform in $[\![0, N]\!]$, because in this case, the returns of our generator will not be uniform in $[\![0, 2^N - 1]\!]$, as it is illustrated in the following example. Let us suppose that $x^0 = (0, 0, 0)$. Then $m^0 \in [\![0, 3]\!]$.

- If $m^0 = 0$, then no bit will change between the first and the second output of our new CI PRNG. Thus $x^1 = (0, 0, 0)$.
- If $m^0 = 1$, then exactly one bit will change, which leads to three possible values for x^1, namely $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$.
- *etc.*

As each value in $[\![0, 2^3 - 1]\!]$ must be returned with the same probability, then the values $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ must occur for x^1 with the same probability. Finally we see that, in this example, $m^0 = 1$ must be three times probable as $m^0 = 0$. This leads to the following general definition for the probability of $m = i$:

$$P(m^n = i) = \frac{C_N^i}{2^N} \tag{7}$$

Then, here is an example for the (m^n) sequence with a selector function g_1

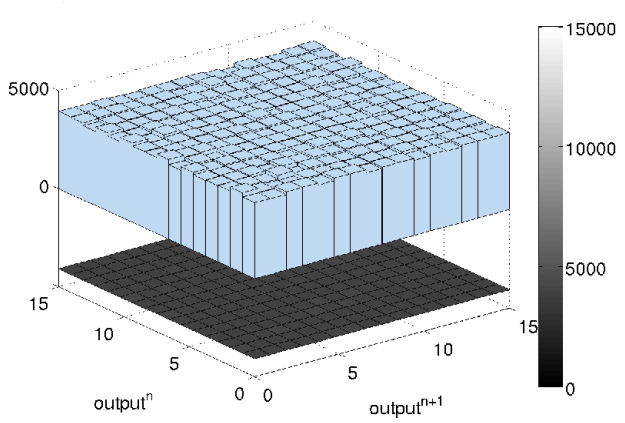
$$m^n = g_1(y^n) = \begin{cases} 0 \text{ if } 0 \leqslant \frac{y^n}{2^{32}} < \frac{C_N^0}{2^N}, \\ 1 \text{ if } \frac{C_N^0}{2^N} \leqslant \frac{y^n}{2^{32}} < \sum_{i=0}^{1} \frac{C_N^i}{2^N}, \\ 2 \text{ if } \sum_{i=0}^{1} \frac{C_N^i}{2^N} \leqslant \frac{y^n}{2^{32}} < \sum_{i=0}^{2} \frac{C_N^i}{2^N}, \\ \vdots \quad \vdots \\ N \text{ if } \sum_{i=0}^{N-1} \frac{C_N^i}{2^N} \leqslant \frac{y^n}{2^{32}} < 1. \end{cases} \tag{8}$$

Let us notice, to conclude this subsection, that our new CI PRNG can use any reasonable function as selector. In this paper, $g_1()$ and $g_2()$ are adopted for demonstration purposes, where:

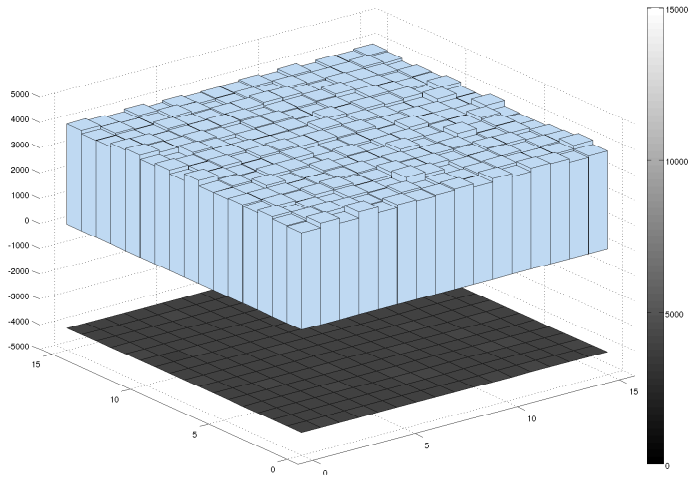
$$m^n = g_2(y^n) = \begin{cases} N \text{ if } 0 \leqslant \frac{y^n}{2^{32}} < \frac{C_N^0}{2^N}, \\ N-1 \text{ if } \frac{C_N^0}{2^N} \leqslant \frac{y^n}{2^{32}} < \sum_{i=0}^{1} \frac{C_N^i}{2^N}, \\ N-2 \text{ if } \sum_{i=0}^{1} \frac{C_N^i}{2^N} \leqslant \frac{y^n}{2^{32}} < \sum_{i=0}^{2} \frac{C_N^i}{2^N}, \\ \vdots \quad \vdots \\ 0 \text{ if } \sum_{i=0}^{N-1} \frac{C_N^i}{2^N} \leqslant \frac{y^n}{2^{32}} < 1. \end{cases} \tag{9}$$

In this thesis, $g_1()$ is the selector function unless noted otherwise. And we will show later that both of $g_1()$ and $g_2()$ can pass all of the performed tests.

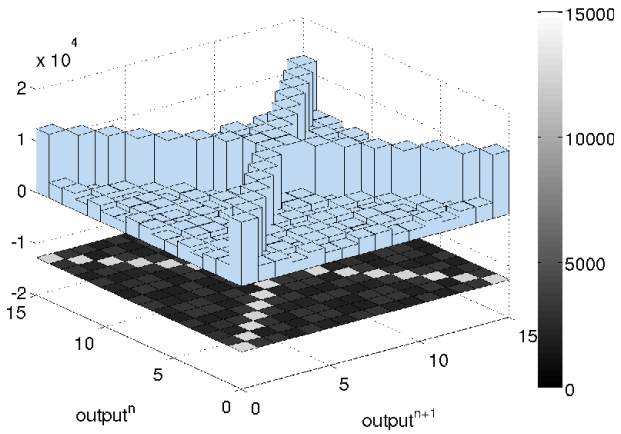
In order to evaluate our proposed method and compare its statistical properties with various other methods, the density histogram and intensity map of adjacent output have been computed. The length of x is $N = 4$ bits, and the initial conditions and control parameters are the same. A large number of sampled values are simulated (10^6 samples). Figure 3.2(a) and Figure 3.2(b) shows the intensity map for $m^n = g_1(y^n)$ and $m^n = g_2(y^n)$. In order to appear random, the histogram should be uniformly distributed in all areas. It can be observed that uniform histograms and flat color intensity maps are obtained when using our schemes. Another illustration of this fact is given by Figure 3.2(c), whereas its uniformity is further justified by the tests presented in Section 4.4.



(a) The histogram of adjacent output distribution $m^n = g_1(y^n)$



(b) The histogram of adjacent output distribution $m^n = g_2(y^n)$



(c) The histogram of adjacent output distribution $m^n = y^n \ mod \ 4$

Figure 3.2: Histogram and intensity maps

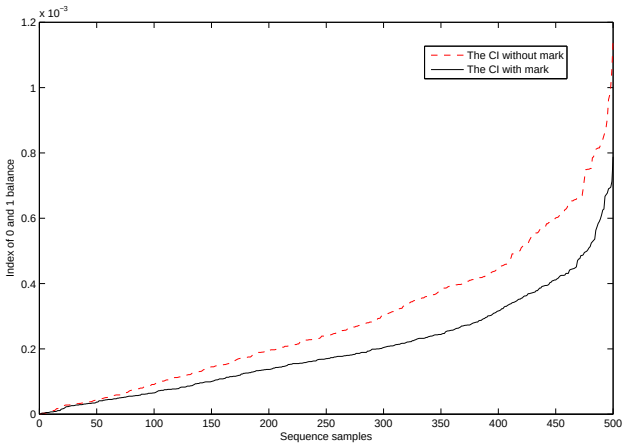


Figure 3.3: Balance property

3.2.4 Chaotic strategy

The chaotic strategy $(S^k) \in [\![1, N]\!]^{\mathbb{N}}$ is generated from a second XORshift sequence $(b^k) \in [\![1, N]\!]^{\mathbb{N}}$. The only difference between the sequences S and b is that some terms of b are discarded, in such a way that $\forall k \in \mathbb{N}, (S^{M^k}, S^{M^k+1}, \ldots, S^{M^{k+1}-1})$ does not contain any given integer twice, where $M^k = \sum_{i=0}^{k} m^i$. Therefore, no bit will change more than once between two successive outputs of our PRNG, increasing the speed of the former generator by doing so. S is said to be "an irregular decimation" of b. This decimation can be obtained by the following process.

Let $(d^1, d^2, \ldots, d^N) \in \{0, 1\}^N$ be a mark sequence, such that whenever $\sum_{i=1}^{N} d^i = m^k$, then $\forall i, d_i = 0$ ($\forall k$, the sequence is reset when d contains m^k times the number 1). This mark sequence will control the XORshift sequence b as follows:

- if $d^{b^j} \neq 1$, then $S^k = b^j$, $d^{b^j} = 1$, and $k = k + 1$,
- if $d^{b^j} = 1$, then b^j is discarded.

For example, if $b = 142\underline{2}33414211\underline{22}34...$ and $m = 4341...$, then $S = 1423\ 341\ 4123\ 4...$ However, if we do not use the mark sequence, then one position may change more than once and the balance property will not be checked, due to the fact that $\bar{\bar{x}} = x$. As an example, for b and m as in the previous example, $S = 1422\ 334\ 1421\ 1...$ and $S = 14\ 4\ 42\ 1...$ lead to the same output (because switching the same bit twice leads to the same state).

To check the balance property, a set of 500 sequences are generated with and without decimation, each sequence containing 10^6 bits. Figure 3.3 shows the percentages of differences between zeros and ones, and presents a better balance property for the sequences with decimation. This claim will be verified in the tests section (Section **??**).

Another example is given in Table 3.3, in which r means "reset" and the integers which are underlined in sequence b are discarded.

3.2.5 Version 2 CI Algorithm

The basic design procedure of the novel generator is summed up in Algorithm 6. The internal state is x, the output state is r. a and b are those computed by the two input PRNGs. The value $g_1(a)$ is an integer, defined as in Equation 8. Lastly, N is a constant defined by the user.

Algorithm 6 An arbitrary round of the Version 2 CI generator

Input: the internal state x (N bits)
Output: a state r of N bits

for $i = 0, \ldots, N$ **do**
  $d_i \leftarrow 0$
$a \leftarrow PRNG1()$
$m \leftarrow f(a)$
$k \leftarrow m$
while $i = 0, \ldots, k$ **do**
  $b \leftarrow PRNG2()\ mod\ \mathsf{N}$
  $S \leftarrow b$
  if $d_S = 0$ **then**
    $x_S \leftarrow \overline{x_S}$
    $d_S \leftarrow 1$
  else if $d_S = 1$ **then**
    $k \leftarrow k + 1$
$r \leftarrow x$ return r

As a comparison, the basic design procedure of the old generator is recalled in Algorithm 7 (a and b are computed by two input PRNGs, N and $c \geqslant 3\mathsf{N}$ are constants defined by the user). See Subsection 2.10 for further information.

Algorithm 7 An arbitrary round of the old CI generator

Input: the internal state x (an array of N 1-bit words)
Output: an array r of N 1-bit words

$a \leftarrow PRNG1()$;
$m \leftarrow a\ mod\ 2 + c$;
while $i = 0, \ldots, m$ **do**
  $b \leftarrow PRNG2()$;
  $S \leftarrow b\ mod\ \mathsf{N}$;
  $x_S \leftarrow \overline{x_S}$;
$r \leftarrow x$;
return r;

3.2.6 Illustrative Example of Version 2 CI (XORshift, XORshift)

In this example, $\mathsf{N} = 4$ is chosen for easy understanding and the input PRNG is XORshift PRNG. As stated before, the initial state of the system x^0 can be seeded by the decimal part t of the current time. For example, if the current time in seconds since the Epoch is 1237632934.484088, so $t = 484088$, then $x^0 = t\ (mod\ 16)$ in binary digits, *i.e.*, $x^0 = (0, 1, 0, 0)$.

To compute m sequence, Equation 8 can be adapted to this example as follows:

$$m^n = g_1(y^n) = \begin{cases} 0 & \text{if} \quad 0 \leqslant \frac{y^n}{2^{32}} < \frac{1}{16}, \\ 1 & \text{if} \quad \frac{1}{16} \leqslant \frac{y^n}{2^{32}} < \frac{5}{16}, \\ 2 & \text{if} \quad \frac{5}{16} \leqslant \frac{y^n}{2^{32}} < \frac{11}{16}, \\ 3 & \text{if} \quad \frac{11}{16} \leqslant \frac{y^n}{2^{32}} < \frac{15}{16}, \\ 4 & \text{if} \quad \frac{15}{16} \leqslant \frac{y^n}{2^{32}} < 1, \end{cases} \tag{10}$$

where y is generated by XORshift seeded with the current time. We can see that the probabilities of occurrences of $m = 0$, $m = 1$, $m = 2$, $m = 3$, $m = 4$, are $\frac{1}{16}$, $\frac{4}{16}$, $\frac{6}{16}$, $\frac{4}{16}$, $\frac{1}{16}$, respectively. This m determines what will be the next output x. For instance,

- If $m = 0$, the following x will be $(0, 1, 0, 0)$.
- If $m = 1$, the following x can be $(1, 1, 0, 0)$, $(0, 0, 0, 0)$, $(0, 1, 1, 0)$, or $(0, 1, 0, 1)$.
- If $m = 2$, the following x can be $(1, 0, 0, 0)$, $(1, 1, 1, 0)$, $(1, 1, 0, 1)$, $(0, 0, 1, 0)$, $(0, 0, 0, 1)$, or $(0, 1, 1, 1)$.
- If $m = 3$, the following x can be $(0, 0, 1, 1)$, $(1, 1, 1, 1)$, $(1, 0, 0, 1)$, or $(1, 0, 1, 0)$.
- If $m = 4$, the following x will be $(1, 0, 1, 1)$.

In this simulation, $m = 0, 4, 2, 2, 3, 4, 1, 1, 2, 3, 0, 1, 4, ...$ Additionally, b is computed with a XORshift generator too, but with another seed. We have found $b = 1, 4, 2, 2, 3, 3, 4, 1, 1, 4, 3, 2, 1, ...$

Chaotic iterations are made with initial state x^0, vectorial logical negation f_0, and strategy S. The result is presented in Table 3.3. Let us recall that sequence m gives the states x^n to return, which are here $x^0, x^{0+4}, x^{0+4+2}, \ldots$ So, in this example, the output of the generator is: 10100111101111110011... or 4,4,11,8,1...

3.2.7 Security Analysis

In this section the concatenation of two strings u and v is classically denoted by uv. In a cryptographic context, a pseudo random generator is a deterministic algorithm G transforming strings into strings and such that, for any seed s of length m, $G(s)$ (the output of G on the input s) has size $l_G(m)$ with $l_G(m) > m$. The notion of secure PRNGs can now be defined as follows.

m	0	4					2		2		
k	0	4				+1	2		2	+1	
b		1	4	2	$\underline{2}$	3	3	4	1	$\underline{1}$	4
d	r	r $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\\1\end{pmatrix}$		$\begin{pmatrix}1\\1\\1\\1\end{pmatrix}$	r $\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\\1\end{pmatrix}$	r $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$		$\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$
S		1	4	2		3	3	4	1		4
x^0	x^0					x^4		x^6			x^8
0	0	$\overset{1}{\rightarrow}$ 1				1		1	$\overset{1}{\rightarrow}$ 0		0
1	1			$\overset{2}{\rightarrow}$ 0		0		0			0
0	0				$\overset{3}{\rightarrow}$ 1	1	$\overset{3}{\rightarrow}$ 0	0			0
0	0		$\overset{4}{\rightarrow}$ 1			1		$\overset{4}{\rightarrow}$ 0	0		$\overset{4}{\rightarrow}$ 1

Binary Output: $x_1^0x_2^0x_3^0x_4^0x_1^4x_2^4x_3^4x_4^4x_1^6x_2^6... = 0100101110000001...$
Integer Output: $x^0, x^4, x^6, x^8... = 4, 11, 8, 1...$

Table 3.3: Example of New CI(XORshift,XORshift) generation

3.2.7.1 Algorithm expression conversion

For the convenience of security analysis, Version 2 CI Algorithm 6 is converted as Equation 12, internal state is x, S and T are those computed by PRNG1 and PRNG2, each round, x^{n-1} is updated to be x^n.

$$m^n = g(S^n) = \begin{cases} 0 \text{ if } 0 \leqslant S^n < C_{32}^0, \\ 1 \text{ if } C_{32}^0 \leqslant S^n < \sum_{i=0}^{1} C_{32}^i, \\ 2 \text{ if } \sum_{i=0}^{1} C_{32}^i \leqslant S^n < \sum_{i=0}^{2} C_{32}^i, \\ \vdots \quad \vdots \\ N \text{ if } \sum_{i=0}^{N-1} C_{32}^i \leqslant S^n < 1. \end{cases} \tag{11}$$

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}}, T \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ m = g(S^n)(\textit{Equation 11}); \\ d^n = 0; \\ d^n = h(d^n, m, T)(\textit{Equation 8}); \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus d^n, \end{cases} \tag{12}$$

3.2.7.2 Proof

Definition 5 A cryptographic PRNG G is secure if for any probabilistic polynomial time algorithm D, for any positive polynomial p, and for all sufficiently large m's,

$$|Pr[D(G(U_m)) = 1] - Pr[D(U_{l_G(m)}) = 1] < \frac{1}{p(m)}, \tag{13}$$

Algorithm 8 Algorithm for $h(d, m, T)$

Input: the internal state d, m, and PRNG sequence T
output: a state r

1: **while** $i = 0, \ldots, m$ **do**
2: $w^i \leftarrow T^{l+i} \ mod \ N$
3: **if** $d^n_{w^i} = 0$ **then**
4: $d^n_{w^i} \leftarrow 1$
5: **else if** $d^n_{w^i} = 1$ **then**
6: $k \leftarrow k + 1$
7: $r \leftarrow d^n$

where U_r is the uniform distribution over $0, 1^r$ and the probabilities are taken over U_m, $U_{l_G(m)}$ as well as over the internal coin tosses of D.

Intuitively, it means that there is no polynomial time algorithm that can distinguish a perfect uniform random generator from G with a non negligible probability. Note that it is quite easily possible to change the function l into any polynomial function l' satisfying $l'(m) > m$.

The generation schema developed in Algorithm 6 is based on 2 pseudo random generators. Let H be the "PRNG1" and I be the "PRNG2". We may assume, without loss of generality, that for any string S_0 of size L, the size of $H(S_0)$ is kL, then for any string T_0 of size M, it has $I(T_0)$ with kM, with $k > 2$. It means that $l_H(L) = kL$ and $l_I(M) = kM$. Let $S_1, ..., S_k$ be the string of length L such that $H(S_0) = S_1...S_k$ and $T_1, ..., T_k$ be the string of length M that $H(S_0) = T_1...T_k$ ($H(S_0)$ and $I(T_0)$ are the concatenations of S_i's and T_i's). The cryptographic PRNG X defined in Algorithm 12 is algorithm mapping any string of length $N + M + L$ $x_0 S_0 T_0$ into the string $x_0 \oplus d^1, x_0 \oplus d^1 \oplus d^2, ...(x_0 \bigoplus_{i=0}^{i=k} d^i)$ (Equation 12). One in particular has $l_X(L + M + N) = kN = l_H(N)$ and $k > M + L + N$. We announce that if one PRNG of H is secure, then the new one from Equation 1 is secure too.

Proposition 2 *If one of H is a secure cryptographic PRNG, then X is a secure cryptographic PRNG too.*

Proof The proposition is proven by contraposition. Assume that X is not secure. By Definition, there exists a polynomial time probabilistic algorithm D, a positive polynomial p, such that for all k_0 there exists $L + M + N \geq k_0$ satisfying

$$|\Pr[D(X(U_{L+M+N})) = 1] - \Pr[D(U_{kN} = 1]| \geq \frac{1}{p(L + M + N)}.$$

Define there is a w of size kL.

1. Decompose w into $w = w_1...w_k$.
2. Define m into $m_1 = w_1 \ mod \ N, m_2 = w_2 \ mod \ N, ...m_k = w_k \ mod \ N$.

3. Pick a string y of size N uniformly at random.

4. Pick a string $u = u_1, u_2...u_k$, which is satisfied with processing k times $h(0, m, u)$.

5. Define $t_i = h(0, m_i, u_i)$.

6. Compute $z = (y \oplus t_1)(y \oplus t_1 \oplus t_2)...(y \bigoplus_{i=1}^{i=k}(t_i))$.

7. Return $D(z)$.

On one hand, consider for each $y \in \mathbb{B}^{kN}$ the function φ_y from $\mathbb{B}^{kN}$ into $\mathbb{B}^{kN}$ mapping $t = t_1 \dots t_k$ (each t_i has length N) to $(y \oplus t_1)(y \oplus t_1 \oplus t_2) \dots (y \bigoplus_{i=1}^{i=k} t_i)$. On the other hand, treat each $u_l \in \mathbb{B}^{(3Nk+\sum_{j=0}^{j=k}(w_j\&1))M}$ the function ϕ_u from $\mathbb{B}^{(3kN+\sum_{j=0}^{j=k}(w_i\&1))M}$ into $mathbbB^{kN}$ mapping $w = w_1 \dots w_k$ (each w_i has length L) to $(\bigoplus_{l=1}^{l=3N+(w_1\&1)}(1 << u_l))((\bigoplus_{l=1+3N+(w_1\&1)}^{l=6N+(w_1\&1)+(w_1\&1)}(1 << u_l)) \dots (\bigoplus_{l=3N(k-1)+\sum_{j=1}^{j=k-1}(w_j\&1)}^{l=3Nk+\sum_{j=1}^{j=k}(w_j\&1)}(1 << u_l)$ By construction, one has for every w,

$$D'(w) = D(\varphi_y(\phi_u(w))), \tag{14}$$

Therefore, and using (14), one has $\Pr[D'(U_{kL}) = 1] = \Pr[D(\varphi_y(\phi_u(U_{kL}))) = 1]$ and, therefore,

$$\Pr[D'(U_{kL}) = 1] = \Pr[D(U_{kN}) = 1]. \tag{15}$$

Now, using (14) again, one has for every x,

$$\Pr[D'(U_{H(x)}) = 1] = \Pr[D(\varphi_y(\phi_u(U_{H(x)}))) = 1] \tag{16}$$

since where y and u_j are randomly generated.
By construction, $\varphi_y(\phi_u(x)) = X(yu_1w)$, hence

$$\Pr[D'(H(U_{kL})) = 1] = \Pr[D(X(U_{N+M+L})) = 1] \tag{17}$$

Compute the difference of Equation (17) and (16), one can deduce that there exists a polynomial time probabilistic algorithm D', a positive polynomial p, such that for all k_0 there exists $L + M + N \geq k_0$ satisfying

$$|\Pr[D'(H(U_{KL})) = 1] - \Pr[D(U_{kL}) = 1]| \geq \frac{1}{p(L + M + N)},$$

proving that H is not secure, which is a contradiction to the first place that one of them is cryptographic secure.

3.3 Version 3 LUT CI(XORshift,XORshift) algorithms

3.3.1 Introduction

The LUT (Lookup-Table) CI generator is an improved version of the new CI generator. The key-ideas are:

- To use a Lookup Table for a faster generation of strategies. These strategies satisfy the same property than the ones provided by the decimation process.

- And to use all the bits provided by the two inputted generators (to discard none of them).

These key-ideas are put together by the following way.

Let us firstly recall that in chaotic iterations, only the cells designed by S^n−th are "iterated" at the n^{th} iteration. S^n can be either a component (*i.e.*, only one cell is updated at each iteration, so $S^n \in [\![1; N]\!]$) or a subset of components (any number of cells can be updated at each iteration, that is, $S^n \subset [\![1; N]\!]$). The first kind of strategies are called "unary strategies" whereas the second one are denoted by "general strategies". In the last case, each term S^n of the strategy can be represented by an integer lower than 2^N, designed by $\mathcal{S}^n$, for a system having N bits: the k^{th} component of the system is updated at iteration number n if and only if the k^{th} digit of the binary decomposition of $\mathcal{S}^n$ is 1. For instance, let us consider that $\mathcal{S}^n = 5$, and that we iterate on a system having 6 bits ($N = 6$). As the integer 5 has a binary decomposition equal to 000101, we thus conclude that the cells number 1 and 3 will be updated when the system changes its state from x^n to x^{n+1}. In other words, in that situation, $\mathcal{S}^n = 5 \in [\![0, 2^6 - 1]\!] \Leftrightarrow S^n = \{1, 3\} \subset [\![1, 6]\!]$. To sum up, to provide a general strategy of $[\![1; N]\!]$ is equivalent to give an unary strategy in $[\![0; 2^N - 1]\!]$. Let us now take into account this remark.

Until now the proposed generators have been presented in this document by using unary strategies (obtained by the first inputted PRNG S) that are finally grouped by "packages" (the size of these packages is given by the second generator m): after having used each terms in the current package $S^{m^n}, ..., S^{m^{n+1}-1}$, the current state of the system is published as an output. Obviously, when considering the Version 2 CI version, these packages of unary strategies defined by the couple $(S, m) \in [\![1; N]\!] \times [\![0; N]\!]$ correspond to subsets of $[\![1; N]\!]$ having the form $\left\{S^{m^n}, ..., S^{m^{n+1}-1}\right\}$, which are general strategies. As stated before, these lasts can be rewritten as unary strategies that can be described as sequences in $[\![0; 2^N - 1]\!]$.

The advantage of such an equivalence is to reduce the complexity of the proposed PRNG. Indeed the new CI(S,m) generator can be written as:

$$x^n = x^{n-1} \wedge \mathcal{S}^n. \tag{18}$$

where $\mathcal{S}$ is the unary strategy (in $[\![0; 2^N - 1]\!]$) associated to the couple $(S, m) \in [\![1; N]\!] \times [\![0, N]\!]$.

The speed improvement is obvious, the sole issue is to understand how to change (S, m) by $\mathcal{S}$. The problem to consider is that all the sequences of $[\![0; 2^n - 1]\!]$ are not convenient. Indeed, the properties required for the couple (S, m) (S must not be uniformly distributed, and a cell cannot be changed twice between two outputs) must be translated in requirements for $\mathcal{S}$ if we want to satisfy both speed and randomness. Such constrains are solved by working on the sequence m and by using some well-defined Lookup Tables presented in the following sections.

3.3.2 Sequence m

In order to improve the speed of the proposed generator, the first plan is to take the best usage of the bits generated by the inputted PRNGs. The problem is that the PRNG generating the integers of m^n does not necessary takes its values into $[\![0, N]\!]$, where N is the size of the system.

For instance, in the new CI generator presented previously, this sequence is obtained by a XORshift, which produces integers belonging into $[\![0, 2^{32} - 1]\!]$. However, the iterated system has 4 cells ($N = 4$) in the example proposed previously thus, to define the sequence m^n, we compute the remainder modulo 4 of each integer provided by the XORshift generator. In other words, only the last 4 bits of each 32 bits vector generated by the second XORshift are used. Obviously this stage can be easily optimized, by splitting this 32-bits vector into 8 subsequences of 4 bits. Thus, a call of XORshift() will now generate 8 terms of the sequence m, instead of only one term in the former generator.

This common-sense action can be easily generalized to any size $N \leqslant 32$ of the system by the procedure described in Algorithm 9. The idea is simply to make a shift of the binary vector a produced by the XORshift generator, by 0, N, $2N$,... bits to the right, depending on the remainder c of n modulo $\lfloor N/32 \rfloor$ (that is, $a \gg (N \times c)$), and to take the bits between the positions $32 - N$ and 32 of this vector (corresponding to the right part "$\&(2^N - 1)$" of the formula). In that situation, all the bits provided by XORshift are used when N divide 32.

Algorithm 9 Generation of sequence b^n

1: $c = n \ mod \ \lfloor 32/N \rfloor$
2: **if** $c = 0$ **then**
3: $\quad a \leftarrow XORshift()$
4: $b^n \leftarrow (a \gg (N \times c))\&(2^N - 1)$
5: Return b^n

This Algorithm 9 produces a sequence $(b^n)_{n\in\mathbb{N}}$ of integers belonging into $[\![0, 2^N - 1]\!]$. It is now possible to define the sequence m by adapting the Equation 9 or Equation 8 as follows.

$$m^n = f(b^n) = \begin{cases} 0 \text{ if } 0 \leqslant b^n < C_N^0, \\ 1 \text{ if } C_N^0 \leqslant b^n < \sum_{i=0}^{1} C_N^i, \\ 2 \text{ if } \sum_{i=0}^{1} C_N^i \leqslant b^n < \sum_{i=0}^{2} C_N^i, \\ \vdots \quad \vdots \\ N \text{ if } \sum_{i=0}^{N-1} C_N^i \leqslant b^n < 2^N. \end{cases} \tag{19}$$

This common-sense measure can be improved another time if N is not very large by using the first Lookup Table of this document, which is called LUT-1. This improvement will be firstly explained through an example.

Let us consider that $N = 4$, so the sequence $(b^n)_{n\in\mathbb{N}}$ belongs into $[\![0, 15]\!]$. The function f of Equation 19 must translate each b^n into an integer $m^n \in [\![0, 4]\!]$, in such a way that the non-uniformity exposed previously is respected. Instead of defining the function f analytically,

Table 3.4: A LUT-1 table for $N = 4$

b^n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m^n	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4

a table can be given containing all the images of the integers into $[\![0, 15]\!]$ (see Table 3.4 for instance). As stated before, the frequencies of occurrence of the images 0,1,2, 3, and 4 must be respectively equal to $\frac{C_4^0}{2^4}$, $\frac{C_4^1}{2^4}$, $\frac{C_4^2}{2^4}$, $\frac{C_4^3}{2^4}$, and $\frac{C_4^4}{2^4}$. This requirement is equivalent to demand C_N^i times the number i, which can be translated in terms of permutations. For instance, when $N = 4$, any permutation of the list [0,1,1,1,1,2,2,2,2,2,2,3,3,3,3,4] is convenient to define the image of [0,1,2,...,14,15] by f.

This improvement is implemented in Algorithm 10, which return a table $lut1$ such that $m^n = lut1[b^n]$.

Algorithm 10 The LUT-1 table generation

```
for j = 0...N do
    i = 0
    while i < C_N^j do
        lut1[i] = j
        i = i + 1
Return lut1
```

3.3.3 Defining the chaotic strategy $\mathcal{S}$ with a LUT

The definition of the sequence m allows to determine the number of cells that have to change between two outputs of the LUT CI generator. There are C_N^m possibilities to change m bits in a vector of size N. As we have to choose between these C_N^m possibilities, we thus introduce the following sequence:

$$w^n = XORshift2() \ mod \ C_N^m \tag{20}$$

With this material it is now possible to define the LUT that provides convenient strategies to the LUT CI generator. If the size of the system is N, then this table has $N + 1$ columns, numbered from 0 to N. The column number m contains C_N^m values. All of these values have in common to present exactly m times the digit 1 and $N - m$ times the digit 0 in their binary decomposition. The order of appearance of these values in the column m has no importance, the sole requirement is that no column contains a same integer twice. Let us remark that this procedure leads to several possible LUTs.

An example of such a LUT is shown in Table 3.5, when Algorithm 12 gives a concrete procedure to obtain such tables. This procedure makes recursive calls to the function $LUT21$ defined in Algorithm 11. The $LUT21$ uses the following variables. b is used to avoid overlapping computations between two recursive calls, v is to save the sum value between these calls, and c counts the number of cells that have already been processed.

Algorithm 11 $LUT21$ procedure

```
Procedure LUT21(M, N, b, v, c)
count ← c
value ← v
if count == M then
    lut2[M][num] = value
    num = num + 1
else
    for i = b....N do
        value = value + 2^i
        count = count + 1
        Call recurse LUT21(M, N, i + 1, value, count)
        value = v
        count = c
End Procedure
```

Table 3.5: Example of a LUT for $N = 4$

w \ m	$m = 0$	$m = 1$	$m = 2$	$m = 3$	$m = 4$
$w = 0$	0	1	3	7	15
$w = 1$		2	5	11	
$w = 2$		4	6	13	
$w = 3$		8	9	14	
$w = 4$			10		
$w = 5$			12		

These parameters should be initialized as 0. For instance, the LUT presented in Table 3.5 is the $lut2$ obtained in Algorithm 11 with $N = 4$.

Algorithm 12 LUT-2 generation

```
for i = 0....N do
    Call LUT21(i, N, 0, 0, 0)
Return lut2
```

3.3.4 LUT CI(XORshift,XORshift) Algorithm

The LUT CI generator is defined by the following dynamical system:

$$x^n = x^{n-1} \wedge \mathcal{S}^n. \tag{21}$$

where $x^O \in [\![0, 2^N - 1$ is a seed and $\mathcal{S}^n = lut2[w^n][m^n] = lut2[w^n][lut1[b^n]]$, in which b^n is provided by Algorithm 9 and $w^n = XORshift2()\ mod\ C_N^m$. An iteration of this generator is

m	0	3	2	1
c	0	2	5	2
S	0	13	12	4
x^0	x^0	x^1	x^2	x^3
0	0	1	0	0
1	1	0	1	0
0	0	0	0	0
0	0	1	1	1

Binary Output: $x_1^0 x_2^0 x_3^0 x_4^0 x_1^1 x_2^1 x_3^1 x_4^1 x_1^2 x_2^2 ... = 0100100101010001...$
Integer Output: $x^0, x^1, x^2, x^3 ... = 4, 11, 8, 1...$

Table 3.6: Example of a LUT CI(XORshift,XORshift) generation

written in Algorithm 13.

Algorithm 13 LUT CI algorithm

1: $b^n \leftarrow PRNG1()$
2: $m^n = lut1[b^n]$
3: $w^n = PRNG2()$
4: $S^n = lut2[m][w]$
5: $x = x \wedge S^n$
6: Return x

3.3.5 LUT CI(XORshift,XORshift) example of use

In this example, $N = 4$ is chosen another time for easy understanding. As before, the initial state of the system x^0 can be seeded by the decimal part t of the current time. With the same current time than in the examples exposed previously, we have $x^0 = (0, 1, 0, 0)$ (or $x^0 = 4$).

Algorithm 10 provides the LUT-1 depicted in Table 3.4. The first XORshift generator has returned $y = 0, 11, 7, 2, 10, 4, 1, 0, 3, 9, ...$ By using this LUT, we obtain $m = 0, 3, 2, 1, 2, 1, 1, 0, 1, 2, ...$ Then the Algorithm 12 is computed, leading to the LUT-2 given by Table 3.5.

So chaotic iterations of Algorithm 13 can be realized, to obtain in this example: 0100100101010001... or 4,9,5,1...

3.3.6 Security Analysis

Consider about the proof for Version 1 and Version 2 CIPRNG, Version 3 CIPRNG is very similar to their conditions, they are all mix two PRNGs to produce output, hence the same contraposition method is able to apply to prove that Version 3 CIPRNG is cryptographically secure when PRNG1 is secure.

3.4 New version (Version 4) of CI

3.4.1 XOR CIPRNG

Instead of updating only one cell at each iteration as Version 1, Version 2 and Version 3 CI we can try to choose a subset of components and to update them together. Such an attempt leads to a kind of merger of the two random sequences. When the updating function is the vectorial negation, this algorithm can be rewritten as follows [16]:

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ d^n = S^n \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus d^n, \end{cases} \tag{22}$$

This rewriting can be understood as follows. The $n-$th term S^n of the sequence S, which is an integer of N binary digits, presents the list of cells to update in the state x^n of the system (represented as an integer having N bits too). More precisely, the $k-$th component of this state (a binary digit) changes if and only if the $k-$th digit in the binary decomposition of S^n is 1.

The single basic component presented in Eq. 12 is of ordinary use as a good elementary brick in various PRNGs. It corresponds to the discrete dynamical system in chaotic iterations.

3.4.2 Introduction

According to the XOR CI in Equation 12, we can try to add more complexity in updating the subset at each iteration. Such an attempt leads to a kind of merger of several sequences, when the updating function is the vectorial negation, this algorithm can be written as follows:

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!] \\ S(1), S(2)...S(M) \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ T \in [\![0, 2^{\mathsf{M}} - 1]\!]^{\mathbb{M}} \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus g_3(S^n(1), S^n(2), ...S^n(M), T^n) \end{cases} \tag{23}$$

The iteration function is next introduced.

In the iteration function, $S(1), S(2)...S(M)$ are random number sequences composed by XORshifts, T^n is generated from CSPRNG: BBS. $(t_1, t_2, \ldots, t_M) \in \{0, 1\}^M$ is the binary representation of T (2^M-bit numbers) . A control sequence T^n decimates the sequences produced by the other generators $S^n(1), S^n(2), ...S^n(M)$ to do bitwise exclusive or. According to the following decimation rule:

- if $t_i^n \neq 1$, then $S^n(i)$ is discarded,
- if $t_i^n = 1$, then $S^n(i)$ is kept for bitwise exclusive or computing.

In brief, the output sequence x^n produced based on chaotic iterations is updating by bitwise exclusive or of an irregular decimation of $S(1), S(2)...S(M)$ in terms of the bits of T^n.

There are M $n-th$ terms $S^n(1), ..., S^n(M)$ of sequences $S(1), S(2)...S(M)$, which are the integers of N bits binary digits; and the term T^n of sequence T are with integers of M bits binary digits, which is equally the number of S sequences. They present the list of cells to update in the state x^n of the system (integer of N bits too) in $g_3(S^n(1), S^n(2), ...S^n(M), T^n)$, and its algorithm is shown in Algorithm 14: where the value of each bit cell in T^n decide its corresponding $S^n(i)$ would be used to bitwise exclusive or computing or not. More accurately, the $k-th$ component of this stat (a binary digit) changes if only if the $k-th$ digit in the binary decomposition of the output of $g_3(S^n(1), S^n(2), ...S^n(M), T^n)$ is 1.

Algorithm 14 Algorithm for $g_3(S^n(1), S^n(2), ...S^n(M), T^n)$

Input: sequences $S^n(1), S^n(2), ...S^n(M)$ and T^n
output: a state r (length N bits)

1: $b \leftarrow T^n$
2: $r \leftarrow 0$
3: **while** $i = 1 \ldots M$(Size of the b) **do**
4: $\quad c \leftarrow S^n(i)$
5: $\quad$ **if** $b\&(2^{i-1}) \neq 0$ **then**
6: $\quad\quad r \leftarrow r \oplus c$
7: return r

3.4.3 Security Analysis

In this subsection the concatenation of two strings u and v is classically denoted by uv. In a cryptographic context, a pseudorandom generator is a deterministic algorithm G transforming strings into strings and such that, for any seed s of length m, $G(s)$ (the output of G on the input s) has size $l_G(m)$ with $l_G(m) > m$. The notion of secure PRNGs can now be defined as follows.

Definition 6 A cryptographic PRNG G is secure if for any probabilistic polynomial time algorithm D, for any positive polynomial p, and for all sufficiently large m's,

$$|Pr[D(G(U_m)) = 1] - Pr[D(U_{l_G(m)}) = 1] < \frac{1}{p(m)}, \tag{24}$$

where U_r is the uniform distribution over $0, 1^r$ and the probabilities are taken over U_m, $U_{l_G(m)}$ as well as over the internal coin tosses of D.

Intuitively, it means that there is no polynomial time algorithm that can distinguish a perfect uniform random generator from G with a non negligible probability. Note that it is quite easily possible to change the function l into any polynomial function l' satisfying $l'(m) > m$. In [16], version 3 CI has been proven that if applied PRNG is cryptographic secure, then the CIPRNG is also cryptographic secure. Here, the proof for the updated version of CI (Equation 23) is given.

The generation schema developed in Equation 23 is based on $M + 1$ pseudorandom generators. Let $H_1, H_2 ... H_M$ be the PRNGs which are used to update the bit cell of internal state, and I be the PRNG which decide which H_j PRNG is available in this round updating. We may assume, without loss of generality, that for any string $S_i(j)$ of size N, the size of $H_j(S_i(j))$ is kN, then for any string T_0 of size M, it has $I(T_0)$ with kM, here $k > 2$,. It means that $l_H(NM) = kNM$ and $l_I(M) = kM$. Let $S_1(1), ..., S_k(2)$, $S_1(2), ..., S_k(2)$, ... $S_1(M), ..., S_k(M)$ and $T_1, ..., T_k$ be the $M + 1$ sequences of strings (S strings are in N bits, and the T string is in M bits). Such that $H_j(S_0(j)) = S_1(j)...S_k(j)$ and $I(T_0) = T_1...T_k$ ($H_i(S(i)_0)$ are the concatenation of the $S_i(j)$ and T_i's). The cryptographic PRNG X defined in Equation 23 is algorithm mapping any string of length $M + NM + N$ $x^0 g_3(S^1(1), S^1(2), ...S^1(M), T^1)$ into the string $x^0 \oplus g_3(S^1(1), S^1(2), ...S^1(M), T^1), x^0 \oplus g_3(S^1(1), S^1(2), ...S^1(M), T^1) \oplus g_3(S^2(1), S^2(2), ...S^2(M), T^2), ...(x^0 \bigoplus_{i=0}^{i=k} g_3(S^i(1), S^i(2), ...S^i(M), T^i))$ (Equation 23). One in particular has $l_X(M + NM + N) = kN = l_H(M)$, here $kN \geq M + NM + N$. We announce that if PRNG I is secure, then the new one from Equation 23 is secure too.

Proposition 3 *If I is a secure cryptographic PRNG, then X is a secure cryptographic PRNG too.*

Proof The proposition is proven by contraposition. Assume that X is not secure. By Definition, there exists a polynomial time probabilistic algorithm D, a positive polynomial p, such that for all k_0 there exists $M + NM + N \geq k_0$ satisfying

$$|\Pr[D(X(U_{M+NM+N})) = 1] - \Pr[D(U_{kN} = 1)]| \geq \frac{1}{p((M + NM + N))}.$$

We describe a new probabilistic algorithm D' on inputs W (each is of size kM):

1. Decompose w into $w = w_1 \dots w_k$.
2. Pick a string y of size N uniformly at random.
3. Pick M strings of size kN: $u(1), ...u(M)$.
4. Decompose each $u(j)$ into $u(j) = u_1(j) \dots u_k(j)$.
5. Define $t_i = \bigoplus_{j=0}^{j=M-1}((w_i >> j)\&1) \times u_i(j + 1))$ from Algorithm 14;
6. Compute $z = (y \oplus t_1)(y \oplus t_1 \oplus t_2)...(y \bigoplus_{i=1}^{i=k}(t_i))$.
7. Return $D(z)$.

Consider for each $y \in \mathbb{B}^{kN}$ the function φ_y from $\mathbb{B}^{kN}$ into $\mathbb{B}^{kN}$ mapping $t = t_1 \dots t_k$ (each t_i has length N) to $(y \oplus t_1)(y \oplus t_1 \oplus t_2) \dots (y \bigoplus_{i=1}^{i=k} t_i)$. By construction, one has for every t,

$$D'(w) = D(\varphi_y(t)), \tag{25}$$

where y is randomly generated. Moreover, for each y, φ_y is injective: if $(y \oplus t_1)(y \oplus t_1 \oplus t_2) \dots (y \bigoplus_{i=1}^{i=k_1} t_i) = (y \oplus t'_1)(y \oplus t'_1 \oplus t'_2) \dots (y \bigoplus_{i=1}^{i=k} t'_i)$, then for every $1 \leq j \leq k$,

$y \bigoplus_{i=1}^{i=j} t'_i = y \bigoplus_{i=1}^{i=j} t_i$. It follows, by a direct induction, that $t_i = t'_i$. Then also consider for each $u_i(j) \in \mathbb{B}^{kN}$ the function ϕ_u from $\mathbb{B}^{kN}$ into $\mathbb{B}^{kN}$ mapping $w = w_1 \dots w_k$ (each w_i has length M) to $(\bigoplus_{j=0}^{j=M-1}((w_1 >> j)\&1) \times u_1(j+1)))(\bigoplus_{j=0}^{j=M-1}((w_2 >> j)\&1) \times u_2(j+1))) \dots (\bigoplus_{j=0}^{j=M-1}((w_k >> j)\&1) \times u_k(j+1)))$. The $u_i(j)$ is generated by $H(j)$ PRNG, ϕ_u is injective: if $(\bigoplus_{j=0}^{j=M-1}((w_1 >> j)\&1) \times u_1(j+1)))(\bigoplus_{j=0}^{j=M-1}((w_2 >> j)\&1) \times u_2(j+1))) \dots (\bigoplus_{j=0}^{j=M-1}((w_k >> j)\&1) \times u_k(j+1))) = (\bigoplus_{j=0}^{j=M-1}((w'_1 >> j)\&1) \times u_1(j+1)))(\bigoplus_{j=0}^{j=M-1}((w'_2 >> j)\&1) \times u_2(j+1))) \dots (\bigoplus_{j=0}^{j=M-1}((w'_k >> j)\&1) \times u_k(j+1)))$, $w_i = w'_i$ can be found. Then according to Equation 25:

$$D'(w) = D(\varphi_y(\phi_u(w))), \tag{26}$$

Furthermore, using (26), one has $\Pr[D'(U_{kM}) = 1] = \Pr[D(\varphi_y(\phi_u(U_{kM}))) = 1]$ and, therefore,

$$\Pr[D'(U_{kM}) = 1] = \Pr[D(U_{kM}) = 1]. \tag{27}$$

Now, using (26) again, one has for every x,

$$D'(I(x)) = D(\varphi_y(\phi_u(I(x)))), \tag{28}$$

since where y and all $u(j)$ are randomly generated.
By construction, $\varphi_y(\phi_u(I(x))) = X(yxu(1)...u(M))$, hence

$$\Pr[D'(I(U_M)) = 1] = \Pr[D(X(U_{M+NM+N})) = 1]. \tag{29}$$

Using Equation (29) minus (27), one can deduce that there exists a polynomial time probabilistic algorithm D', a positive polynomial p, such that for all k_0 there exists $M + NM + N \geq k_0$ satisfying

$$|\Pr[D'(I(U_M)) = 1] - \Pr[D'(U_{kM} = 1)]| \geq \frac{1}{p(M + NM + N)},$$

proving that I is not secure, which is a contradiction to the first place that it is cryptographic secure.

3.4.4 Efficient cryptographic secure PRNG based on CI

Here, in Table 3.7, an efficient, based on CI, good random quality, and cryptographically secure PRNG algorithm is described, it can split into two parts:

First part is based on Equation 23, For FPGA application, it would be very suitable due to it can be easily arranged to be processed on parallel, more than that, according to the description of Section 3.4.3, the new versions of CIPRNG can turn to be cryptographically secure. For constructing the generator which is cryptographic secure, M + 1 kinds of classic PRNGs would be applied, and due to Proposition 1, it simply consists in replacing one of them by a cryptographically secure one. We have chosen BBS PRNG in this design due to its high security. Some believe that the BBS algorithm is the most secure PRNG method available [62]. The security of BBS is based on its long period and the difficulty in

Input: x (a 32-bit word)
Output: r (a 32-bit word)
$t1 \leftarrow xorshift1()$; $t2 \leftarrow xorshift2()$; $t4 \leftarrow bbs()$;   **if** $t4\&1 \neq 0$; **then** $x \leftarrow x \oplus (t1\&0x0fffffffff)$;   **if** $t4\&2 \neq 0$; **then** $x \leftarrow x \oplus (t1 >> 32)$;   **if** $t4\&4 \neq 0$; **then**$x \leftarrow x \oplus (t2\&0x0fffffffff)$; $x \leftarrow x \oplus (t2 >> 32)$; $r \leftarrow x$;;   return r;
An arbitrary round of the algorithm

Table 3.7: Algorithm efficient for FPGA

predicting the sequence even if all previously generated bits are known. Despite the strong security of the algorithm, the BBS sequence generator is simple and easy to understood. Thus it is used to perform T in Equation 23 since its slow efficient, and due to size of m is in 32 bits, according to the rule of secure bits extracted $log(log(m))$, each output x's four least significant bits (LSBs) are considered to be secure to use, here we set $M = 3$. Then the $M = 3$ PRNGs to play the role of S chosen to be two XORshift based on 64 bits. In Table 3.7, $xorshift1$ and $xorshift2$ represents them. Each XORshift output are separated into two 32 bits, and it leads to four 32 bits binaries. Three one of them (first and second 32 bits of $xorshift1$, and first 32 bits of $xorshift2$) are controlled by the bits output of BBS PRNG (bbs) respectively as Equation 23 told, the rest 32 bits (second 32 bits of $xorshift2$) is used in the part 2 of the algorithm. If one bit cell of bbs output is 0, then the corresponding 32 bits do not take part in exclusive-or processing. On the contrary, if the bit cell is 1, such bits would be exclusive-or with the state.

According to our experiments, if there is only the first part of the algorithm, it can not give very good statistically randomness output, as told from [9], increasing using CI is able to improve the statistical property. Hence, the second part of the algorithm is to use the second 32 bits of $xorshift2$ to process Equation 12 with the output of first part. Then the output of algorithm in Table 3.7 can keep to be of property of CI and cryptographically secure due to [16].

This algorithm is very similar to the efficient GPU CI version (successfully pass TestU01 [56]) from [16], except the in GPU version there is no BBS to decide the higher 32 bits of XORshifts to join the processing, and three 64 bits XORshfit PRNGs have been applied. However, GPU version is not proven to be cryptographic secure.

CHAPTER 4

Randomness Of CIPRNGs

4.1 Some famous statistical tests of random number generators

A theoretical proof for the randomness of a generator is impossible to give, therefore statistical inference based on observed sample sequences produced by the generator seems to be the best option. Considering the properties of binary random sequences, various statistical tests can be designed to evaluate the assertion that the sequence is generated by a perfectly random source. We have performed certain statistical tests for various CI PRNGs we proposed. These tests include TestU01 [56], NIST suite [53], Diehard battery of tests [37], and Comparative test parameters. For completeness and for reference, we give in the following subsection a brief description of each of the aforementioned tests.

4.1.1 NIST statistical test suite

Among the numerous standard tests for pseudo-randomness, a convincing way to show the randomness of the produced sequences is to confront them to the NIST (National Institute ofStandards and Technology) Statistical Test, because it is an up-to-date test suite proposed by the Information Technology Laboratory (ITL). A new version of the Statistical Test Suite (Version 2.0) has been released in August 11, 2010.

The NIST test suite SP 800-22 is a statistical package consisting of 15 tests. They were developed to test the randomness of binary sequences produced by hardware or software based cryptographic PRNGs. These tests focus on a variety of different types of non-randomness that could exist in a sequence.

For each statistical test, a set of $P - values$ (corresponding to the set of sequences) is produced. The interpretation of empirical results can be conducted in any number of ways. In this paper, the examination of the distribution of P-values to check for uniformity ($P - value_T$) is used. The distribution of P-values is examined to ensure uniformity. If $P - value_T \geqslant 0.0001$, then the sequences can be considered to be uniformly distributed.

In our experiments, 100 sequences (s = 100), each with 1,000,000-bit long, are generated and tested. If the $P - value_T$ of any test is smaller than 0.0001, the sequences are considered to be not good enough and the generating algorithm is not suitable for usage.

In what follows, the fifteen tests of the NIST Statistical tests suite, are recalled. A more detailed description for those tests could be found in [53].

- **Frequency (Monobit) Test (FT)** is to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence.

- **Frequency Test within a Block (FBT)** is to determine whether the frequency of ones in an M-bit block is approximately $M/2$, as would be expected under an assumption of randomness.(M is the length of each block.)

- **Runs Test (RT)** is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence. In particular, this test determines whether the oscillation between such zeros and ones is too fast or too slow.

- **Test for the Longest Run of Ones in a Block (LROBT)** is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence.

- **Binary Matrix Rank Test (BMRT)** is to check for linear dependence among fixed length substrings of the original sequence.

- **Discrete Fourier Transform (Spectral) Test (DFTT)** is to detect periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness.

- **Non-overlapping Template Matching Test (NOTMT)** is to detect generators that produce too many occurrences of a given non-periodic (aperiodic) pattern.(m is the length in bits of each template which is the target string.)

- **Overlapping Template Matching Test (OTMT)** is the number of occurrences of pre-specified target strings.(m is the length in bits of the template–in this case, the length of the run of ones.)

- **Maurer's "Universal Statistical" Test (MUST)** is to detect whether or not the sequence can be significantly compressed without loss of information.(L is the length of each block, and Q is the number of blocks in the initialization sequence)

- **Linear Complexity Test (LCT)** is to determine whether or not the sequence is complex enough to be considered random.(M is the length in bits of a block.)

- **Serial Test (ST)** is to determine whether the number of occurrences of the 2^m m-bit.(m is the length in bits of each block.) overlapping patterns is approximately the same as would be expected for a random sequence.

- **Approximate Entropy Test (AET)** is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and m+1) against the expected result for a random sequence.(m is the length of each block.)

- **Cumulative Sums (Cusum) Test (CST)** is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences.

- **Random Excursions Test (RET)** is to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence.

- **Random Excursions Variant Test (REVT)** is to detect deviations from the expected number of visits to various states in the random walk.

4.1.2 Diehard battery of tests

The Diehard battery of tests was developed in 1996 by Prof. Georges Marsaglia from the Florida State University for testing randomness of sequences of numbers [77]. It has been the most sophisticated standard for over a decade. Because of the stringent requirements in the Diehard test suite, a generator passing Diehard battery of tests can be considered good as a rule of thumb. It was supposed to give a better way of analysis in comparison to original FIPS statistical tests.

The Diehard battery of tests consists of 18 different independent statistical tests. Each test requires binary file of about 10-12 million bytes in order to run the full set of tests. As the NIST test suite, most of the tests in Diehard return a $p - value$, which should be uniform on $[0, 1)$ if the input file contains truly independent random bits.Those $p - value$s are obtained by $p = F(X)$, where F is the assumed distribution of the sample random variable X (often normal). But that assumed F is just an asymptotic approximation, for which the fit will be worst in the tails. Thus occasional $p - value$s near 0 or 1, such as 0.0012 or 0.9983 can occur. Unlike the NIST test suite, the test is considered to be successful when the $p - value$ is in range where $[0 + \alpha, 1 - \alpha]$ is the level of significance of the test.

For example, with a level of significance of 5%, p-value are expected to be in $[0.025, 0.975]$. Note that if the $p - value$ is not in this range, it means that the null hypothesis for randomness is rejected even if the sequence is truly random. These tests are:

- **Birthday Spacings** Choose random points on a large interval. The spacings between the points should be asymptotically Poisson distributed. The name is based on the birthday paradox.

- **Overlapping Permutations** Analyze sequences of five consecutive random numbers. The 120 possible orderings should occur with statistically equal probability

- **Ranks of matrices** Select some number of bits from some number of random numbers to form a matrix over 0,1, then determine the rank of the matrix. Count the ranks.

- **Monkey Tests** Treat sequences of some number of bits as ”words”. Count the overlapping words in a stream. The number of ”words” that don’t appear should follow a known distribution. The name is based on the infinite monkey theorem.

- **Count the 1’s** Count the 1 bits in each of either successive or chosen bytes. Convert the counts to ”letters”, and count the occurrences of five-letter ”words”

- **Parking Lot Test** Randomly place unit circles in a $100x100$square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully ”parked” circles should follow a certain normal distribution.

- **Minimum Distance Test** Randomly place 8,000 points in a $10,000x10,000$ square, then find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.
- **Random Spheres Test** Randomly choose 4,000 points in a cube of edge 1,000. Center a sphere on each point, whose radius is the minimum distance to another point. The smallest sphere's volume should be exponentially distributed with a certain mean.
- **The Sqeeze Test** Multiply 231 by random floats on [0,1) until you reach 1. Repeat this 100,000 times. The number of floats needed to reach 1 should follow a certain distribution.
- **Overlapping Sums Test** Generate a long sequence of random floats on [0,1). Add sequences of 100 consecutive floats. The sums should be normally distributed with characteristic mean and sigma.
- **Runs Test** Generate a long sequence of random floats on [0,1). Count ascending and descending runs. The counts should follow a certain distribution.
- **The Craps Test** Play 200,000 games of craps, counting the wins and the number of throws per game. Each count should follow a certain distribution.

4.1.3 Comparative test parameters

In this section, five well-known statistical tests [40] are used ascomparison tools. They encompass frequency and autocorrelation tests. In what follows, $s = s^0, s^1, s^2, \ldots, s^{n-1}$ denotes a binary sequence of length n. The question is to determine whether this sequence possesses some specific characteristics that a truly random sequence would be likely to exhibit. The tests are introduced in this subsection and results are given in the next one.

Frequency test (monobit test) The purpose of this test is to check if the numbers of 0's and 1's are approximately equal in s, as it would be expected for a random sequence. Let n_0, n_1 denote these numbers. The statistic used here is

$$X_1 = \frac{(n_0 - n_1)^2}{n},$$

which approximately follows a χ^2 distribution with one degree of freedom when $n \geqslant 10^7$.

Serial test (2-bit test) The purpose of this test is to determine if the number of occurrences of 00, 01, 10 and 11 as subsequences of s are approximately the same. Let n_{00}, n_{01}, n_{10}, and n_{11} denote the number of occurrences of $00, 01, 10$, and 11 respectively. Note that $n_{00} + n_{01} + n_{10} + n_{11} = n - 1$ since the subsequences are allowed to overlap. The statistic used here is:

$$X_2 = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1,$$

which approximately follows a χ^2 distribution with 2 degrees of freedom if $n \geqslant 21$.

Poker test The poker test studies if each pattern of length m (without overlapping) appears the same number of times in s. Let $\lfloor \frac{n}{m} \rfloor \geqslant 5 \times 2^m$ and $k = \lfloor \frac{n}{m} \rfloor$. Divide the sequence s into k non-overlapping parts, each of length m. Let n_i be the number of occurrences of the i^{th} type of sequence of length m, where $1 \leqslant i \leqslant 2^m$. The statistic used is

$$X_3 = \frac{2^m}{k} \left(\sum_{i=1}^{2^m} n_i^2 \right) - k,$$

which approximately follows a χ^2 distribution with $2^m - 1$ degrees of freedom. Note that the poker test is a generalization of the frequency test (setting $m = 1$ in the poker test yields the frequency test).

Runs test The purpose of the runs test is to figure out whether the number of runs of various lengths in the sequence s is as expected, for a random sequence. A run is defined as a pattern of all zeros or all ones, a block is a run of ones, and a gap is a run of zeros. The expected number of gaps (or blocks) of length i in a random sequence of length n is $e_i = \frac{n-i+3}{2^{i+2}}$. Let k be equal to the largest integer i such that $e_i \geqslant 5$. Let B_i, G_i be the number of blocks and gaps of length i in s, for each $i \in [\![1, k]\!]$. The statistic used here will then be:

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i},$$

which approximately follows a χ^2 distribution with $2k - 2$ degrees of freedom.

Autocorrelation test The purpose of this test is to check for coincidences between the sequence s and (non-cyclic) shifted versions of it. Let d be a fixed integer, $1 \leqslant d \leqslant \lfloor n/2 \rfloor$. The$A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$ is the amount of bits not equal between the sequence and itself displaced by d bits. The statistic used is:

$$X_5 = \frac{2\left(A(d) - \frac{n-d}{2}\right)}{\sqrt{n-d}},$$

which approximately follows a normal distribution $\mathcal{N}(0, 1)$ if $n - d \geqslant 10$. Since small values of $A(d)$ are as unexpected as large values, a two-sided test should be used.

4.1.4 TestU01 Statistical Test

TestU01 is extremely diverse in implementing classical tests, cryptographic tests, new tests proposed in the literature, and original tests. In fact, it encompasses most of the other testsuites. Seven batteries of tests in the TestU01 package are listed as follows:

- **Small Crush.** The first battery to check, with 15 $p-$values reported. This is a fast collection of tests used to be sure that the basic requirements of randomness are satisfied. In case of success, this battery should be followed by Crush and BigCrush.

- **Crush.** This battery includes many difficult tests, like those described in [31]. It uses approximately 2^35 random numbers and applies 96 statistical tests (it computes a total of 144 test statistics and p-values)
- **Big Crush.** it uses approximately 2^38 random numbers and applies 106 tests (it computes 160 test statistics and p-values) A suite of very stringent statistical tests, and the most difficult battery to pass.
- **Rabbit.** This battery of tests reports 38 $p-$values.
- **Alphabit.** Alphabit and AlphabitFile have been designed primarily to test hardware random bits generators. 17 $p-$values are reported.
- **Pseudo-DieHARD.** This battery implements most of the tests contained in the popular battery DieHARD or, in some cases, close approximations to them. It is not a very stringent battery. Indeed, there is no generator that can pass Crush and BigCrush batteries and fail Pseudo-DieHARD, while the converse occurs for several defective generators. 126 $p-$values are reported here.
- **FIPS_140_2.** The NIST (National Institute of Standards and Technology) of the U.S. federal government has proposed a statistical test suite. It is used to evaluate the randomness of bitstreams produced by cryptographic random number generators. This battery reports 16 $p-$values.

Six predefined batteries of tests are available in TestU01; three of them are for sequences of U (0, 1) random numbers and the three others are for bit sequences. In the first category, we have SmallCrush, Crush, and BigCrush To test a RNG for general use. one could first apply the small and fast battery SmallCrush. If it passes, one could then apply the more stringent battery Crush, and finally the yet more time-consuming battery BigCrush. These batteries of tests include the classical tests described in Knuth [31], for example, the run, poker, coupon collector, gap, max-of-t, and permutation tests. There are collision and birthday spacings tests in 2, 3, 4, 7, 8 dimensions, several close pairs tests in 2, 3, 5, 7, 9 dimensions, and correlation tests. Some tests use the generated numbers as a sequence of "random" bits: random walk tests, linear complexity tests, a Lempel-Ziv compression test, several Hamming weights tests, matrix rank tests, run and correlation tests, among others.

The batteries Rabbit, Alphabit, and BlockAlphabit are for binary sequences (e.g., a cryptographic pseudorandom generator or a source of random bits produced by a physical device). They were originally designed to test a finite sequence contained in a binary file. When invoking the battery, one must specify the number n_B of bits available for each test. When the bits are in a file, n_B must not exceed the number of bits in the file, and each test will reuse the same sequence of bits starting from the beginning of the file (so the tests are not independent). When the bits are produced by a generator, each test uses a different stream. In both cases, the parameters of each test are chosen automatically as a function of n_B . The batteries Alphabit and Rabbit can be applied on a binary file considered as a source of random bits. They can also be applied on a programmed generator. Alphabit has been

defined primarily to test hardware random bits generators. The battery PseudoDIEHARD applies most of the tests in the well-known DIEHARD suite of Marsaglia [106]. The battery FIPS_140_2 implements the small suite of tests of the FIPS_140_2 standard from NIST. The batteries described in this module will write the results of each test (on standard output) with a standard level of details (assuming that the boolean switches of module swrite have their default values), followed by a summary report of the suspect p-values obtained from the specific tests included in the batteries. It is also possible to get only the summary report in the output, with no detailed output from the tests, by setting the boolean switch swrite_Basic to FALSE. Rabbit and Alphabit apply 38 and 17 different statistical tests, respectively.

Some of the tests compute more than one statistic (and p-value) using the same stream of random numbers and these statistics are thus not independent. That is why the number of statistics in the summary reports is larger than the number of tests in the description of the batteries.

- **Small Crush.**smarsa_BirthdaySpacings
  sknuth_Collision
  sknuth_Gap
  sknuth_SimpPoker
  sknuth_CouponCollector
  sknuth_MaxOft
  svaria_WeightDistrib
  smarsa_MatrixRank
  sstring_HammingIndep
  swalk_RandomWalk1

- **Crush.**smarsa_SerialOver
  smarsa_CollisionOver
  smarsa_BirthdaySpacings
  snpair_ClosePairs
  snpair_ClosePairsBitMatch
  sknuth_SimpPoker
  sknuth_CouponCollector
  sknuth_Gap
  sknuth_Run
  sknuth_Permutation
  sknuth_CollisionPermut
  sknuth_MaxOft
  svaria_SampleProd
  svaria_SampleMean
  svaria_SampleCorr
  svaria_AppearanceSpacings
  svaria_WeightDistrib

svaria_SumCollector
smarsa_MatrixRank
smarsa_Savir2
smarsa_GCD
swalk_RandomWalk1
scomp_LinearComp
scomp_LempelZiv
sspectral_Fourier3
sstring_LongestHeadRun
sstring_PeriodsInStrings
sstring_HammingWeight2
sstring_HammingCorr
sstring_HammingIndep
sstring_Run
sstring_AutoCor

- **Big Crush.** smarsa_SerialOver
  smarsa_CollisionOver
  smarsa_BirthdaySpacings
  snpair_ClosePairs
  sknuth_SimpPoker
  sknuth_CouponCollector
  sknuth_Gap
  sknuth_Run
  sknuth_Permutation
  sknuth_CollisionPermut
  sknuth_MaxOft
  svaria_SampleProd
  svaria_SampleMean
  svaria_SampleCorr
  svaria_AppearanceSpacings
  svaria_WeightDistrib
  svaria_SumCollector
  smarsa_MatrixRank
  smarsa_Savir2
  smarsa_GCD
  swalk_RandomWalk1
  scomp_LinearComp
  scomp_LempelZiv
  sspectral_Fourier3
  sstring_LongestHeadRun
  sstring_PeriodsInStrings
  sstring_HammingWeight2

sstring_HammingCorr
sstring_HammingIndep
sstring_Run
sstring_AutoCor

- **Rabbit.** smultin_MultinomialBitsOver
snpair_ClosePairsBitMatch
svaria_AppearanceSpacings
scomp_LinearComp
scomp_LempelZiv
sspectral_Fourier1
sspectral_Fourier3
sstring_LongestHeadRun
sstring_PeriodsInStrings
sstring_HammingWeight
sstring_HammingCorr
sstring_HammingIndep
sstring_AutoCor
sstring_Run
smarsa_MatrixRank
swalk_RandomWalk1

- **Alphabit.** smultin_MultinomialBitsOver
sstring_HammingIndep
sstring_HammingCorr
swalk_RandomWalk1

- **Pseudo-DieHARD.** Birthday Spacings test
Overlapping 5-Permutation test
Binary Rank Tests for Matrices test
Bitstream test
OPSO test
OQSO test
DNA test
Count-the-1's test
Parking Lot test
Minimum Distance test
3-D Spheres test
Squeeze test
Overlapping Sums test
Runs test
Craps test

- **FIPS_140_2.** Monobit test
  "poker" test
  Runs test
  Longest Run of Ones in a Block test

TestU01 suite implements hundreds of tests and reports $p-$values. If a $p-$value is within $[0.001, 0.999]$, the associated test is a success. A $p-$value lying outside this boundary means that its test has failed.

4.2 Test results for some PRNGs

4.2.1 Results of NIST

In our experiments, 100 sequences (s = 100) of 1,000,000 bits are generated and tested. If the value $\mathbb{P}_T$ of any test is smaller than 0.0001, the sequences are considered to not be good enough and the generator is unsuitable. Table 4.1 shows $\mathbb{P}_T$ of the sequences for BBS, Logistic map, XORshift and ISAAC in Section 2.9. If there are at least two statistical values in a test, this test is marked with an asterisk and the average value is computed to characterize the statistical values.

4.2.2 Results of Diehard

Table 4.2 gives the results derived from applying the DieHARD battery of tests to Logistic map, XORshift and ISAAC in Section 2.9.

4.2.3 Results of comparative test parameters

We show in Table 4.3 a comparison between BBS, Logistic map, XORshift and ISAAC in Section 2.9.

4.2.4 Results of TestU01

Table 4.4 gives the results derived from applying the TestU01 battery of tests to the PRNGs considered in BBS, Logistic map, XORshift and ISAAC in Section 2.9.

4.2.5 Conclusion

From the results of the TestU01, NIST, Comparative test parameters and DieHARD batteries of tests applied to Logistic map, XORshift and ISAAC in Section 2.9, the worst situation obviously appears when using the BBS, logistic map or XORshift, andISAAC can pass the three batteries of tests.

We can conclude that BBS performs very bad in the tests due to its low periodic property.

Table 4.1: NIST SP 800-22 test results ($\mathbb{P}_T$)

Test name	BBS	Logistic	XORshift	ISAAC
Frequency (Monobit) Test	0.32435	0.53414	0.14532	0.67868
Frequency Test within a Block	0.000000	0.00275	0.45593	0.10252
Runs Test	0.000000	0.00001	0.21330	0.69931
Longest Run of Ones in a Block Test	0.000000	0.08051	0.28966	0.43727
Binary Matrix Rank Test	0.000000	0.67868	0.00000	0.89776
Discrete Fourier Transform (Spectral) Test	0.000000	0.57490	0.00535	0.51412
Non-overlapping Template Matching Test*	0.000000	0.28468	0.50365	0.55515
Overlapping Template Matching Test	0.000000	0.10879	0.86769	0.63711
Universal Statistical Test	0.000000	0.02054	0.27570	0.69931
Linear Complexity Test	0.043355	0.79813	0.92407	0.03756
Serial Test* (m=10)	0.000000	0.41542	0.75792	0.32681
Approximate Entropy Test (m=10)	0.000000	0.02054	0.41902	0.30412
Cumulative Sums (Cusum) Test*	0.000000	0.60617	0.81154	0.36786
Random Excursions Test*	0.000000	0.53342	0.41923	0.50711
Random Excursions Variant Test*	0.000000	0.28507	0.52833	0.40930
Success	2/15	15/15	14/15	15/15

Table 4.2: Results of DieHARD battery of tests

No.	Test name	BBS	Logistic	XORshift	ISAAC
1	Overlapping Sum	Pass	Pass	Pass	Pass
2	Runs Up 1	Pass	Pass	Pass	Pass
	Runs Down 1	Pass	Pass	Pass	Pass
	Runs Up 2	Pass	Pass	Pass	Pass
	Runs Down 2	Pass	Pass	Pass	Pass
3	3D Spheres	Fail	Pass	Pass	Pass
4	Parking Lot	Fail	Pass	Pass	Pass
5	Birthday Spacing	Fail	Pass	Pass	Pass
6	Count the ones 1	Fail	Pass	Fail	Pass
7	Binary Rank 6×8	Fail	Pass	Pass	Pass
8	Binary Rank 31×31	Fail	Pass	Fail	Pass
9	Binary Rank 32×32	Fail	Pass	Fail	Pass
10	Count the ones 2	Fail	Pass	Pass	Pass
11	Bit Stream	Fail	Pass	Pass	Pass
12	Craps Wins	Fail	Pass	Pass	Pass
	Throws	Fail	Pass	Pass	Pass
13	Minimum Distance	Fail	Pass	Pass	Pass
14	Overlapping Perm.	Fail	Pass	Pass	Pass
15	Squeeze	Fail	Pass	Pass	Pass
16	OPSO	Fail	Pass	Pass	Pass
17	OQSO	Fail	Pass	Pass	Pass
18	DNA	Fail	Pass	Pass	Pass
	Number of tests passed	2	18	15	18

Then binary Matrix Rank Test is failed for XORshift in NIST statistical test suite. The focus of the test is the rank of disjoint sub-matrices of the entire sequence. Note that this test also appears in the DIEHARD battery of tests.

XORshift fails three individual tests contained into the DieHARD battery, namely: "Count the ones", "Binary Rank 31×31", and "Binary Rank 32×32". We can thus conclude that, in the random numbers obtained with XORshift, only the least significant bits seem to be independent.

Seven tests were failed with a $p - value$ practically equal to 0 or 1 for logistic map and XORshift in TestU01 Statistical Test. It is clear that the null hypothesis H_0 must be

Table 4.3: Comparative test parameters with a 10^7 bits sequence

Method	Threshold values	BBS	Logistic	XORshift	ISAAC
Monobit	3.8415	0.3485	0.1280	1.7053	0.1401
Serial	5.9915	2.0079	0.1302	2.1466	0.1430
Poker	316.9194	9.7216	240.2893	248.9318	236.8670
Runs	55.0027	33.2067	26.5667	18.0087	34.1273
Autocorrelation	1.6449	-0.7603	0.0373	0.5099	-2.1712

Table 4.4: TestU01 Statistical Test

Test name	Battery	Parameters	BBS	Logistic	XORshift	ISAAC
Rabbit	32×10^9 bits	38	26	21	14	0
Alphabit	32×10^9 bits	17	9	16	9	0
Pseudo DieHARD	Standard	126	8	0	2	0
FIPS_140_2	Standard	16	0	0	0	0
Small Crush	Standard	15	10	4	5	0
Crush	Standard	144	117	95	57	0
Big Crush	Standard	160	134	125	55	0
Number of failures		516	304	261	146	0

rejected for these two bit streams. H_0 is equivalent to saying that for each integer $t > 0$, the vector $(u_0, ..., u_{t-1})$is uniformly distributed over the t-dimensional unit cube $[0, 1]^t$

4.3 Test results and comparative analysis for Version 1 CI

4.3.1 How to choose good parameters

4.3.1.1 Small Crush test and correlation with k

To exhibit the correlation between the parameter k such that $\mathcal{M}$ = {k, k+1} (see Section 2.10.1) and the success rate, we have used CI(ISAAC, XORshift) PRNG as a concrete example.

In Figure 4.1 is plotted the Small Crush test on sequences generated by CI(ISAAC, XORshift). Small Crush is succeeded when the total number of passes is 15. In these figures, the ordinates are the number of successful passes a sequence goes through. Thus, a qualified sequence must have most of the time a passing value equal to 15, with possible occasional failures (even a true random sequence can occasionally fail these tests). It can be seen in Figure 4.1, and it as been obtained too in other simulations we have realized,

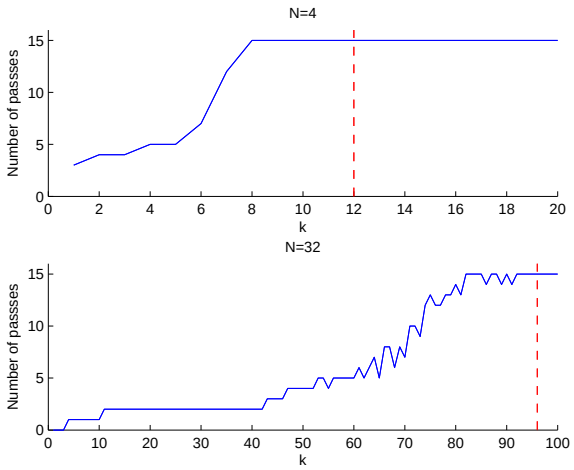


Figure 4.1: Small Crush for CI(ISAAC,XORshift)

that when $k > 3\mathsf{N}$, the sequences tend to pass the Small Crush test.

4.3.1.2 Correlation between TestU01 results and N

To compare the sequences generated with different parameters N in a more quantitative manner, we have set $k = 3N + 1$, and the same analysis for the four CI(X,Y) generators through TestU01 has been repeated. Let us recall that the TestU01 suite implements 518 tests and reports $p-$values, which must be within $[0.001, 0.999]$ for a passing test.

Figure 4.2 shows the number of passing sequences generated by the CI(X,Y) PRNGs proposed in this paper, with the same parameters and initial values. It can be seen that $\mathsf{N} = 4$ gives the best results.

For this reason, it is the default $\mathsf{N} = 4$ with $k = 12$ mode in following sections.

4.3.2 Results of NIST

In our experiments, 100 sequences (s = 100) of 1,000,000 bits are generated and tested. If the value $\mathbb{P}_T$ of any test is smaller than 0.0001, the sequences are considered to not be good enough and the generator is unsuitable. Table 4.5 shows $\mathbb{P}_T$ of the sequences based on discrete chaotic iterations using different schemes. If there are at least two statistical values in a test, this test is marked with an asterisk and the average value is computed to characterize the statistical values.

4.3.3 Results of Diehard

Table 4.6 gives the results derived from applying the DieHARD battery of tests to the PRNGs considered in this work.

Table 4.5: NIST SP 800-22 test results ($\mathbb{P}_T$) for Version 1 CI algorithms (N = 4)

Test name	Version 1 CI			
	Logistic + Logistic	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Frequency (Monobit) Test	0.85138	0.59554	0.40119	0.33536
Frequency Test within a Block	0.38382	0.55442	0.89776	0.47890
Runs Test	0.31908	0.45593	0.31908	0.89942
Longest Run of Ones in a Block Test	0.13728	0.01671	0.08558	0.02357
Binary Matrix Rank Test	0.69931	0.61630	0.47498	0.25021
Discrete Fourier Transform (Spectral) Test	0.12962	0.00019	0.77918	0.08945
Non-overlapping Template Matching Test*	0.48473	0.53225	0.53568	0.24587
Overlapping Template Matching Test	0.47498	0.33453	0.36691	0.94873
Universal Statistical Test	0.09657	0.03292	0.26224	0.32486
Linear Complexity Test	0.41902	0.40119	0.61715	0.88345
Serial Test* (m=10)	0.53427	0.01339	0.33453	0.43803
Approximate Entropy Test (m=10)	0.99146	0.13728	0.53414	0.34759
Cumulative Sums (Cusum) Test*	0.75530	0.04646	0.31915	0.23536
Random Excursions Test*	0.65406	0.50362	0.50804	0.87633
Random Excursions Variant Test*	0.55388	0.34777	0.48400	0.64242
Success	15/15	15/15	15/15	15/15

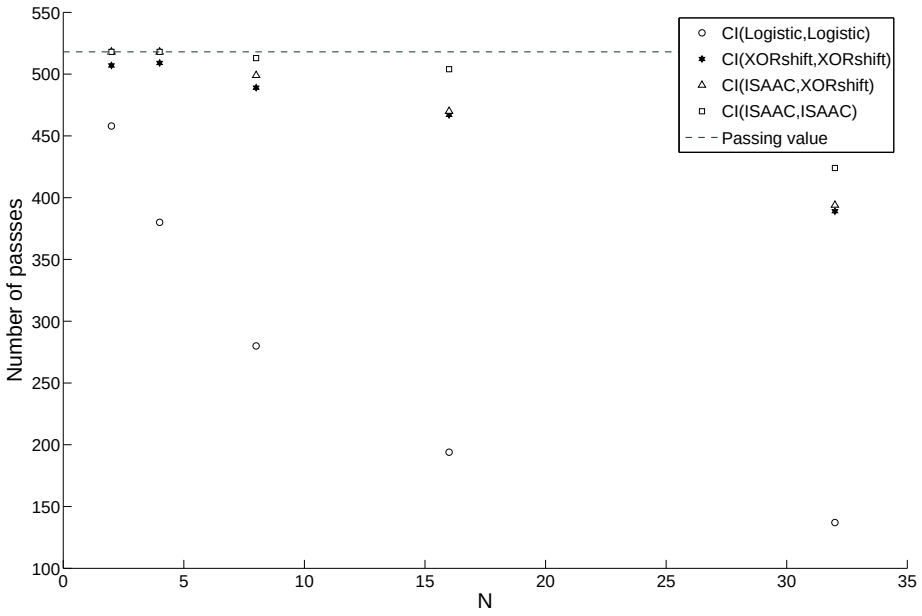


Figure 4.2: TestU01 results

4.3.4 Results of comparative test parameters

We show in Table 4.7 a comparison between Version 1 CI(Logistic, Logistic), Version 1 CI(XORshift, XORshift), Version 1 CI(ISAAC, XORshift), and Version 1 CI(BBS, XORshift).

4.3.5 Results of TestU01

In a sound theoretical basis, a PRNG based on discrete chaotic iterations (CI) is a composite generator which combines the features of two PRNGs. The first generator constitutes the initial condition of the chaotic dynamical system. The second generator randomly chooses which outputs of the chaotic system must be returned. The intention of this combination is to cumulate the effects of chaotic and random behaviors, to improve the statistical and security properties relative to each generator taken alone.

This PRNG based on discrete chaotic iterations may utilize any reasonable RNG as inputs. For demonstration purposes, Logistic map, XORshift and ISAAC are adopted here.

Table 4.8 gives the results derived from applying the TestU01 battery of tests to the PRNGs considered in this work.

4.3.6 Conclusion

Our Version 1 CI PRNG based on discrete chaotic iterations combines the features of two PRNGs, in order to improve their statistical properties. BBS, Logistic map, XORshift, and

Table 4.6: Results of DieHARD battery of tests for Version 1 CI algorithms (N = 4)

No.	Test name	Version 1 CI			
		Logistic + Logistic	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
1	Overlapping Sum	Pass	Pass	Pass	Pass
2	Runs Up 1	Pass	Pass	Pass	Pass
	Runs Down 1	Pass	Pass	Pass	Pass
	Runs Up 2	Pass	Pass	Pass	Pass
	Runs Down 2	Pass	Pass	Pass	Pass
3	3D Spheres	Pass	Pass	Pass	Pass
4	Parking Lot	Pass	Pass	Pass	Pass
5	Birthday Spacing	Pass	Pass	Pass	Pass
6	Count the ones 1	Pass	Pass	Pass	Pass
7	Binary Rank 6×8	Pass	Pass	Pass	Pass
8	Binary Rank 31×31	Pass	Pass	Pass	Pass
9	Binary Rank 32×32	Pass	Pass	Pass	Pass
10	Count the ones 2	Pass	Pass	Pass	Pass
11	Bit Stream	Pass	Pass	Pass	Pass
12	Craps Wins	Pass	Pass	Pass	Pass
	Throws	Pass	Pass	Pass	Pass
13	Minimum Distance	Pass	Pass	Pass	Pass
14	Overlapping Perm.	Pass	Pass	Pass	Pass
15	Squeeze	Pass	Pass	Pass	Pass
16	OPSO	Pass	Pass	Pass	Pass
17	OQSO	Pass	Pass	Pass	Pass
18	DNA	Pass	Pass	Pass	Pass
	Number of tests passed	18	18	18	18

ISAAC are adopted here for demonstration purposes. The results of comparative test parameters confirm that the proposed CI PRNGs are all able to pass these tests. Statistical results of comparative test parameters for both CI(XORshift, XORshift) and CI(ISAAC, XORshift) (proven to be cryptographically secure)are better for most of the parameters, leading to the conclusion that these generators are more secure than the others. This im-

Table 4.7: Comparative test parameters for Version 1 CI(X,Y) with a 10^7 bits sequence (N = 4)

Method	Threshold values	Version 1 CI			
		Logistic + Logistic	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Monobit	3.8415	1.0368	3.5689	0.0569	0.6641
Serial	5.9915	1.1758	3.5765	0.9828	0.6506
Poker	316.9194	269.0607	222.3683	243.8415	262.6440
Runs	55.0027	36.5479	28.4237	29.3195	30.3116
Autocorrelation	1.6449	0.4054	0.3403	0.6141	0.9455

Table 4.8: TestU01 Statistical Test for Version 1 CI algorithms (N = 4)

Test name			Version 1 CI			
			Logistic + Logistic	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Rabbit	32×10^9 bits	38	7	2	0	2
Alphabit	32×10^9 bits	17	3	0	0	2
Pseudo DieHARD	Standard	126	0	0	0	0
FIPS_140_2	Standard	16	0	0	0	0
Small Crush	Standard	15	2	0	0	0
Crush	Standard	144	47	4	0	10
Big Crush	Standard	160	79	3	0	41
Number of failures		518	138	9	0	55

provement clearly appears in the TestU01 results: i.e. XORshift alone fails 142 of these tests, whereasCI(XORshift, XORshift) only fails 9 out of 518. In other words, in addition of having chaotic properties, our PRNG based on discrete chaotic iterations can pass more performed tests than its individual components taken alone.

Table 4.9: SP 800-22 test results ($\mathbb{P}_T$) for Version 2 CI(XORshift, XORshift)N = 32

Method	Version 2 CI		
	$m^n = y^n \ mod \ N$	no mark	$g_2()$
Frequency (Monobit) Test	0.00040	0.08556	0.41943
Frequency Test within a Block	0	0	0.67862
Runs Test	0.28966	0.55448	0.33452
Longest Run of Ones in a Block Test	0.01096	0.43723	0.88313
Binary Matrix Rank Test	0	0.65794	0.75972
Discrete Fourier Transform (Spectral) Test	0	0	0.00085
Non-overlapping Template Matching Test*	0.02007	0.37333	0.51879
Overlapping Template Matching Test	0	0	0.24924
Maurer's "Universal Statistical" Test	0.69936	0.96424	0.12963
Linear Complexity Test	0.36699	0.92423	0.35045
Serial Test* (m=10)	0	0.28185	0.25496
Approximate Entropy Test (m=10)	0	0.38381	0.75971
Cumulative Sums (Cusum) Test*	0	0	0.34245
Random Excursions Test*	0.46769	0.34788	0.18977
Random Excursions Variant Test*	0.28779	0.46505	0.26563
Success	8/15	11/15	15/15

4.4 Test results and comparative analysis for Version 2 CI

4.4.1 Results of NIST

In our experiments, 100 sequences (s = 100) of 1,000,000 bits are generated and tested. If the value $\mathbb{P}_T$ of any test is smaller than 0.0001, the sequences are considered to not be good enough and the generator is unsuitable. Table 4.9 and Table 4.10 n show $\mathbb{P}_T$ of the sequences based on discrete chaotic iterations using different schemes. If there are at least two statistical values in a test, this test is marked with an asterisk and the average value is computed to characterize the statistical values.

We can conclude from Table 4.9 that the worst situations are obtained with the Version 2 CI ($m^n = y^n \ mod \ N$) and Version 2 CI (no mark) generators. Version 2 CI ($m^n = g_2(y^n)$) as Version 2 CI ($m^n = g_1(y^n)$) in Table 4.10 has almost all successfully passed the NIST statistical test suite, the structure of mixing BBS and XORshift can not pass the tests of "Approximate Entropy" and "Serial", the reason might relates to poor statistical properties of BBS in long term output, though it can be treated as cryptographically secure.

Table 4.10: NIST SP 800-22 test results ($\mathbb{P}_T$) for Version 2 CI algorithms

Test name	Version 2 CI		
	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Frequency (Monobit) Test	0.47498	0.88317	0.83430
Frequency Test within a Block	0.89776	0.40119	0.33453
Runs Test	0.81653	0.31908	0.00576
Longest Run of Ones in a Block Test	0.79813	0.06688	0.47498
Binary Matrix Rank Test	0.26224	0.88317	0.69931
Discrete Fourier Transform (Spectral) Test	0.00716	0.33453	0.59559
Non-overlapping Template Matching Test*	0.44991	0.46467	0.51446
Overlapping Template Matching Test	0.51412	0.69931	0.88317
Universal Statistical Test	0.67868	0.24928	0.06282
Linear Complexity Test	0.65793	0.65793	0.94630
Serial Test* (m=10)	0.42534	0.90619	0.00000
Approximate Entropy Test (m=10)	0.63719	0.22482	0.00000
Cumulative Sums (Cusum) Test*	0.27968	0.84065	0.14139
Random Excursions Test*	0.28740	0.30075	0.34625
Random Excursions Variant Test*	0.48668	0.34294	0.55048
Success	15/15	15/15	13/15

4.4.2 Results of Diehard

Table 4.11 gives the results derived from applying the DieHARD battery of tests to the PRNGs considered in this work. Very the same reason as NIST test suite's results, mixing of BBS and XORshift is not able to pass them all.

4.4.3 Results of comparative test parameters

We show in Table 4.12 a comparison between Version 2 CI(XORshift, XORshift), Version 2 CI(ISAAC, XORshift), and Version 2 CI(BBS, XORshift).

4.4.4 Results of TestU01

Table 4.13 gives the results derived from applying the TestU01 battery of tests to the PRNGs considered in this work.

Table 4.11: Results of DieHARD battery of tests for Version 2 CI algorithms (N = 32)

No.	Test name	Version 2 CI			
		Logistic + Logistic	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
1	Overlapping Sum	Pass	Pass	Pass	Pass
2	Runs Up 1	Pass	Pass	Pass	Pass
	Runs Down 1	Pass	Pass	Pass	Pass
	Runs Up 2	Pass	Pass	Pass	Pass
	Runs Down 2	Pass	Pass	Pass	Pass
3	3D Spheres	Pass	Pass	Pass	Pass
4	Parking Lot	Pass	Pass	Pass	Pass
5	Birthday Spacing	Pass	Pass	Pass	Pass
6	Count the ones 1	Pass	Pass	Pass	Fail
7	Binary Rank 6×8	Pass	Pass	Pass	Pass
8	Binary Rank 31×31	Pass	Pass	Pass	Pass
9	Binary Rank 32×32	Pass	Pass	Pass	Pass
10	Count the ones 2	Pass	Pass	Pass	Pass
11	Bit Stream	Pass	Pass	Pass	Pass
12	Craps Wins	Pass	Pass	Pass	Pass
	Throws	Pass	Pass	Pass	Pass
13	Minimum Distance	Pass	Pass	Pass	Pass
14	Overlapping Perm.	Pass	Pass	Pass	Pass
15	Squeeze	Pass	Pass	Pass	Pass
16	OPSO	Pass	Pass	Pass	Pass
17	OQSO	Pass	Pass	Pass	Pass
18	DNA	Pass	Pass	Pass	Fail
	Number of tests passed	18	18	18	16

4.4.5 Conclusion

In a sound theoretical basis, a New PRNG based on discrete chaotic iterations (CI) is a composite generator which combines the features of two PRNGs. The first generator constitutes the initial condition of the chaotic dynamical system. The second generator ran-

Table 4.12: Comparative test parameters for Version 2 CI(X,Y) with a 10^7 bits sequence (N = 32)

Method	Threshold values	Version 2 CI		
		XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Monobit	3.8415	3.5689	0.9036	0.5788
Serial	5.9915	3.5765	1.1229	0.7378
Poker	316.9194	123.6831	173.8604	319.8609
Runs	55.0027	28.4237	40.4606	49.4057
Autocorrelation	1.6449	0.3403	0.1245	-2.0276

Table 4.13: TestU01 Statistical Test for Version 2 CI algorithms (N = 32)

Test name			Version 2 CI		
			Logistic + Logistic	ISAAC + XORshift	BBS + XORshift
Rabbit	32×10^9 bits	38	0	0	7
Alphabit	32×10^9 bits	17	0	0	6
Pseudo DieHARD	Standard	126	0	0	1
FIPS_140_2	Standard	16	0	0	0
Small Crush	Standard	15	0	0	3
Crush	Standard	144	0	0	11
Big Crush	Standard	160	0	0	29
Number of failures			0	0	57

domly chooses which outputs of the chaotic system must be returned. The intention of this combination is to cumulate the effects of chaotic and random behaviors, to improve the statistical and security properties relative to each generator taken alone.

This Version 2 CI PRNG based on discrete chaotic iterations may utilize any reasonable RNG as inputs. For demonstration purposes, XORshift and ISAAC are adopted here.

The results of the TestU01, NIST, Comparative test parameters and DieHARD batteries of tests confirm that the proposed Version 2 CI PRNGs are all able to pass these tests. And it is better than Version 1 CI algorithms

The improvement clearly appears in the TestU01 results: i.e. XORshift alone fails 142 of these tests, whereasVersion 1 CI(XORshift, XORshift) fails 9 out of 518, Version 2 CI(XORshift, XORshift) can pass all the tests. However for CI(BBS, XORshift), to choose m fluents more than to choose k in the output quality, Version 2 CI shows worse performance than Version 1.

Detail comparative analysis will be presented in the latter section.

4.5 Test results and comparative analysis for Version 3 LUT CI

4.5.1 Results of NIST

In our experiments, 100 sequences (s = 100) of 1,000,000 bits are generated and tested. If the value $\mathbb{P}_T$ of any test is smaller than 0.0001, the sequences are considered to not be good enough and the generator is unsuitable. If there are at least two statistical values in a test, this test is marked with an asterisk and the average value is computed to characterize the statistical values.

The Version 3 LUT CI we used in test are all in $N = 4$ bits format. We can conclude from Table 4.14 that except mixture of BBS and XORshift, all others have successfully passed the NIST statistical test suite. The same reason as previously described, BBS's poor statistical performance leads outputs of LUT-1 distribute not very uniform.

4.5.2 Results of Diehard

Table 4.15 gives the results derived from applying the DieHARD battery of tests to the PRNGs considered in this work. Very the same reason as NIST test suite's results, mixture of BBS and XORshift is not able to pass them all, though it is proven to be cryptographically secure, however its poor performance in statistical deny its qualify to be a good generator.

4.5.3 Results of comparative test parameters

We show in Table 4.16 a comparison between Version 3 CI(XORshift, XORshift), Version 3 CI(ISAAC, XORshift), and Version 3 CI(BBS, XORshift). According to these 10^7 bits long sequence tests, again (BBS, XORshift) shows the worst performance.

4.5.4 Results of TestU01

Table 4.17 gives the results derived from applying the TestU01 battery of tests to the PRNGs considered in this work.

4.5.5 Conclusion

By using some well-defined Lookupand due to the rewrite of the way to generate strategies, the generator based on chaotic iterations works faster and is more secure. The speed of LUT

Table 4.14: NIST SP 800-22 test results ($\mathbb{P}_T$) for Version 3 LUT CI algorithms

Test name	Version 3 LUT CI		
	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Frequency (Monobit) Test	0.32435	0.33171	0.00000
Frequency Test within a Block	0.85643	0.42327	0.13233
Runs Test	0.11623	0.31908	0.00000
Longest Run of Ones in a Block Test	0.74254	0.86688	0.00000
Binary Matrix Rank Test	0.23224	0.88317	0.90311
Discrete Fourier Transform (Spectral) Test	0.12316	0.34578	0.59559
Non-overlapping Template Matching Test*	0.43295	0.32637	0.00000
Overlapping Template Matching Test	0.31472	0.55915	0.00000
Universal Statistical Test	0.37864	0.24925	0.06282
Linear Complexity Test	0.65723	0.31793	0.94630
Serial Test* (m=10)	0.43532	0.55190	0.00000
Approximate Entropy Test (m=10)	0.34254	0.12482	0.00000
Cumulative Sums (Cusum) Test*	0.11272	0.04065	0.14139
Random Excursions Test*	0.02003	0.32275	0.34625
Random Excursions Variant Test*	0.43554	0.234294	0.55048
Success	15/15	15/15	8/15

Table 4.15: Results of DieHARD battery of tests for Version 3 LUT CI algorithms (N = 4)

No.	Test name	Version 3 CI			
		Logistic + Logistic	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
1	Overlapping Sum	Pass	Pass	Pass	Fail
2	Runs Up 1	Pass	Pass	Pass	Pass
	Runs Down 1	Pass	Pass	Pass	Pass
	Runs Up 2	Pass	Pass	Pass	Pass
	Runs Down 2	Pass	Pass	Pass	Pass
3	3D Spheres	Pass	Pass	Pass	Fail
4	Parking Lot	Pass	Pass	Pass	Pass
5	Birthday Spacing	Pass	Pass	Pass	Fail
6	Count the ones 1	Pass	Pass	Pass	Fail
7	Binary Rank 6×8	Pass	Pass	Pass	Pass
8	Binary Rank 31×31	Pass	Pass	Pass	Fail
9	Binary Rank 32×32	Pass	Pass	Pass	Fail
10	Count the ones 2	Pass	Pass	Pass	Fail
11	Bit Stream	Pass	Pass	Pass	Pass
12	Craps Wins	Pass	Pass	Pass	Fail
	Throws	Pass	Pass	Pass	Pass
13	Minimum Distance	Pass	Pass	Pass	Pass
14	Overlapping Perm.	Pass	Pass	Pass	Pass
15	Squeeze	Pass	Pass	Pass	Pass
16	OPSO	Pass	Pass	Pass	Fail
17	OQSO	Pass	Pass	Pass	Fail
18	DNA	Pass	Pass	Pass	Fail
	Number of tests passed	18	18	18	8

Table 4.16: Comparative test parameters for Version 3 CI(X,Y) with a 10^7 bits sequence (N = 4)

Method	Threshold values	Version 3 CI		
		XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Monobit	3.8415	3.5689	0.9036	1.5788
Serial	5.9915	3.5765	1.1229	3.378
Poker	316.9194	123.6831	173.8604	409.3320
Runs	55.0027	28.4237	40.4606	88.4153
Autocorrelation	1.6449	0.3403	0.1245	-2.0276

Table 4.17: TestU01 Statistical Test for Version 3 CI algorithms (N = 4)

Test name			Version 3 CI		
			Logistic + Logistic	ISAAC + XORshift	BBS + XORshift
Rabbit	32×10^9 bits	38	0	0	18
Alphabit	32×10^9 bits	17	0	0	8
Pseudo DieHARD	Standard	126	0	0	11
FIPS_140_2	Standard	16	0	0	0
Small Crush	Standard	15	0	0	7
Crush	Standard	144	0	0	51
Big Crush	Standard	160	0	0	77
Number of failures			0	0	165

CI can be hugely improved compare to Version 1 and Version 2 CI. As the tests shown, it also can offer a sufficient speed and level of security.

This Version 3 CI PRNG based on discrete chaotic iterations may utilize any reasonable RNG as inputs. For demonstration purposes, XORshift, BBS, ISAAC are adopted here.

The results of the TestU01, NIST, Comparative test parameters and DieHARD batteries of tests confirm that the proposed Version 3 CI PRNGs are all able to pass these tests. And it is better than Version 1 CI algorithms, also use less computation source than Version 2 CI algorithm.

4.6 Tests results and comparative analysis for Version 4 CI

4.6.1 Results of NIST

In our experiments, 100 sequences (s = 100) of 1,000,000 bits are generated and tested. If the value $\mathbb{P}_T$ of any test is smaller than 0.0001, the sequences are considered to not be good enough and the generator is unsuitable. If there are at least two statistical values in a test, this test is marked with an asterisk and the average value is computed to characterize the statistical values.

The Version 4 CI we used in test are all in $N = 32$ bits format. We can conclude from Table 4.18 that all PRNGs have successfully passed the NIST statistical test suite. Even with BBS's not good property, the Version 4 CIPRNG (BBS, Xorshift) still can perform well.

4.6.2 Results of Diehard

Table 4.19 gives the results derived from applying the DieHARD battery of tests to the PRNGs considered in this work. The same as NIST test suite's results, the mixing of BBS and XORshift shows better performance than in the other CI Versions, all generators are successfully pass the tests.

4.6.3 Results of comparative test parameters

We show in Table 4.20 a comparison between Version 4 CI(XORshift, XORshift), Version 4 CI(ISAAC, XORshift), and Version 4 CI(BBS, XORshift). According to these 10^7 bits long sequence tests, they all show very good statistical performance.

4.6.4 Results of TestU01

Table 4.21 gives the results derived from applying the TestU01 battery of tests to the PRNGs considered in this work.

4.6.5 Conclusion

The new Version CI cause a great compromise among cryptographically secure property, statistical performance and efficiency, it is very suitable used in hardware implementation,

Table 4.18: NIST SP 800-22 test results ($\mathbb{P}_T$) for Version 3 LUT CI algorithms

Test name	Version 4 CI		
	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Frequency (Monobit) Test	0.21414	0.43622	0.24563
Frequency Test within a Block	0.23423	0.43536	0.13233
Runs Test	0.56471	0.23425	0.23562
Longest Run of Ones in a Block Test	0.33252	0.86688	0.12346
Binary Matrix Rank Test	0.01450	0.25689	0.90311
Discrete Fourier Transform (Spectral) Test	0.25462	0.32324	0.59559
Non-overlapping Template Matching Test*	0.79521	0.32637	0.03984
Overlapping Template Matching Test	0.69342	0.55915	0.13839
Universal Statistical Test	0.44654	0.24925	0.06282
Linear Complexity Test	0.97319	0.31793	0.54630
Serial Test* (m=10)	0.58993	0.55190	0.98234
Approximate Entropy Test (m=10)	0.39284	0.12482	0.12345
Cumulative Sums (Cusum) Test*	0.43582	0.04065	0.14139
Random Excursions Test*	0.92001	0.32275	0.34625
Random Excursions Variant Test*	0.24567	0.234294	0.55048
Success	15/15	15/15	15/15

Table 4.19: Results of DieHARD battery of tests for Version 4 CI algorithms (N = 4)

No.	Test name	Version 4 CI			
		Logistic + XORshift	XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
1	Overlapping Sum	Pass	Pass	Pass	Pass
2	Runs Up 1	Pass	Pass	Pass	Pass
	Runs Down 1	Pass	Pass	Pass	Pass
	Runs Up 2	Pass	Pass	Pass	Pass
	Runs Down 2	Pass	Pass	Pass	Pass
3	3D Spheres	Pass	Pass	Pass	Pass
4	Parking Lot	Pass	Pass	Pass	Pass
5	Birthday Spacing	Pass	Pass	Pass	Pass
6	Count the ones 1	Pass	Pass	Pass	Pass
7	Binary Rank 6×8	Pass	Pass	Pass	Pass
8	Binary Rank 31×31	Pass	Pass	Pass	Pass
9	Binary Rank 32×32	Pass	Pass	Pass	Pass
10	Count the ones 2	Pass	Pass	Pass	Pass
11	Bit Stream	Pass	Pass	Pass	Pass
12	Craps Wins	Pass	Pass	Pass	Pass
	Throws	Pass	Pass	Pass	Pass
13	Minimum Distance	Pass	Pass	Pass	Pass
14	Overlapping Perm.	Pass	Pass	Pass	Pass
15	Squeeze	Pass	Pass	Pass	Pass
16	OPSO	Pass	Pass	Pass	Pass
17	OQSO	Pass	Pass	Pass	Pass
18	DNA	Pass	Pass	Pass	Pass
	Number of tests passed	18	18	18	18

Table 4.20: Comparative test parameters for Version 4 CI(X,Y) with a 10^7 bits sequence (N = 32)

Method	Threshold values	Version 3 CI		
		XORshift + XORshift	ISAAC + XORshift	BBS + XORshift
Monobit	0.8115	1.0689	0.9036	0.5788
Serial	2.3312	0.5765	1.0229	1.378
Poker	112.2190	123.6831	133.8604	79.3320
Runs	33.0027	28.4237	13.4606	12.4153
Autocorrelation	0.6449	1.3403	0.1845	-0.7276

Table 4.21: TestU01 Statistical Test for Version 4 CI algorithms (N = 4)

Test name			Version 4 CI		
			Logistic + XORshift	ISAAC + XORshift	BBS + XORshift
Rabbit	32×10^9 bits	0	0	0	0
Alphabit	32×10^9 bits	0	0	0	0
Pseudo DieHARD	Standard	0	0	0	0
FIPS_140_2	Standard	0	0	0	0
Small Crush	Standard	0	0	0	0
Crush	Standard	0	0	0	0
Big Crush	Standard	0	0	0	0
Number of failures		0	0	0	0

also the software implementation.

This Version 4 CI PRNG based on discrete chaotic iterations is able to utilize any reasonable RNG as inputs. For demonstration purposes, XORshift, BBS, ISAAC are adopted here.

The results of the TestU01, NIST, Comparative test parameters and DieHARD batteries of tests confirm that the proposed Version 4 CI PRNGs are all able to pass these tests. And it is better than the other CI algorithms, it is very adaptive to parallel application.

4.7 Assessment of two chaotic iterations schemes based on XORshift Generator

In this section, the comparison between Version 1-4 CIPRNG in statistical aspect. For fairy, all the tested CIPRNGs are using XORshift as is subset PRNGs, and the assignments of the parameters are the most optimized according to the experiments: thus the N for Version 1 and 3 CI is 4, for Version 2 and 4 is 32.

4.7.1 NIST

In our experiments, 100 sequences (s = 100) of 1,000,000 bits are generated and tested. If the value $\mathbb{P}_T$ of any test is smaller than 0.0001, the sequences are considered to not be good enough and the generator is unsuitable. Table 4.22 shows $\mathbb{P}_T$ of the sequences based on discrete chaotic iterations using different schemes. If there are at least two statistical values in a test, this test is marked with an asterisk and the average value is computed to characterize the statistical values. We can conclude from Table 4.22 that XORshift has failed 1 test, whereas both the old generator and CI(XORshift, XORshift) have successfully passed the NIST statistical test suite. This result shows the good behavior of both PRNGs in the aforementioned basic tests that evaluate the independence of real numbers.

4.7.2 Diehard

Table 4.23 gives the results derived from applying the DieHARD battery of tests to the PRNGs considered in this work. As it can be observed, the results of the individual tests Count the ones 1, Binary Rank 31×31 and Binary Rank 32×32 show that in the random numbers obtained with the XORshift generator only the least significant bits seem to be independent. This explains the poor behavior of this PRNG in the aforementioned basic tests that evaluate the independence of real numbers. But the generator based on discrete chaotic iterations (Version 1 - 4 CI PRNGs) can pass all the DieHARD battery of tests. This proves that the security of the given generator has been improved by chaotic iterations.

4.7.3 Comparative test parameters

We show in Table 4.24 a comparison between our new generator Version 1 - 4 CI(XORshift, XORshift) PRNGs and a PRNG based on a simple XORshift. Time (in seconds) is related to the duration needed by each algorithm to generate a 2×10^5 bits long sequence. The

Table 4.22: NIST SP 800-22 test results (N = 4 for Version 1 and 3 CIPRNG,N = 32 for XORshift generator, Version 2 and 4 CIPRNG)

PRNG	Classic	CI PRNG versions			
Method	XORshift	1	2	3	4
Frequency (Monobit) Test	0.1453	0.5955	0.4744	0.3257	0.8841
Frequency Test within a Block	0.4553	0.5524	0.8970		
Runs Test	0.2134	0.4551	0.8161		
Longest Run of Ones in a Block Test	0.2890	0.0126	0.7398		
Binary Matrix Rank Test	0.0000	0.6126	0.2621		
Discrete Fourier Transform (Spectral) Test	0.0051	0.0002	0.0071		
Non-overlapping Template Matching Test*	0.5036	0.5322	0.4499		
Overlapping Template Matching Test	0.8676	0.3345	0.5141		
Universal Statistical Test	0.2757	0.0329	0.6786		
Linear Complexity Test	0.9240	0.4011	0.6579		
Serial Test* (m=10)	0.7579	0.0133	0.4253		
Approximate Entropy Test (m=10)	0.4190	0.1373	0.6371		
Cumulative Sums (Cusum) Test*	0.8115	0.0464	0.2796		
Random Excursions Test*	0.4192	0.5036	0.2874		
Random Excursions Variant Test*	0.5283	0.3477	0.4866		
Success	14/15	15/15	15/15		

Table 4.23: Results of DieHARD battery of tests

No.	Test name	Classic	CI PRNG versions			
		XORshift	1	2	3	4
1	Overlapping Sum	Pass	Pass	Pass	Pass	Pass
2	Runs Up 1	Pass	Pass	Pass	Pass	Pass
	Runs Down 1	Pass	Pass	Pass	Pass	Pass
	Runs Up 2	Pass	Pass	Pass	Pass	Pass
	Runs Down 2	Pass	Pass	Pass	Pass	Pass
3	3D Spheres	Pass	Pass	Pass	Pass	Pass
4	Parking Lot	Pass	Pass	Pass	Pass	Pass
5	Birthday Spacing	Pass	Pass	Pass	Pass	Pass
6	Count the ones 1	Fail	Pass	Pass	Pass	Pass
7	Binary Rank 6×8	Pass	Pass	Pass	Pass	Pass
8	Binary Rank 31×31	Fail	Pass	Pass	Pass	Pass
9	Binary Rank 32×32	Fail	Pass	Pass	Pass	Pass
10	Count the ones 2	Pass	Pass	Pass	Pass	Pass
11	Bit Stream	Pass	Pass	Pass	Pass	Pass
12	Craps Wins	Pass	Pass	Pass	Pass	Pass
	Throws	Pass	Pass	Pass	Pass	Pass
13	Minimum Distance	Pass	Pass	Pass	Pass	Pass
14	Overlapping Perm.	Pass	Pass	Pass	Pass	Pass
15	Squeeze	Pass	Pass	Pass	Pass	Pass
16	OPSO	Pass	Pass	Pass	Pass	Pass
17	OQSO	Pass	Pass	Pass	Pass	Pass
18	DNA	Pass	Pass	Pass	Pass	Pass
	Number of tests passed	15	18	18	18	18

test has been conducted using the same computer and compiler with the same optimization settings for both algorithms, in order to make it as fair as possible.

As a comparison of the overall stability of these PRNGs, similar tests have been computed for different sequence lengths (see Figures 4.3 - 4.7). For the monobit test comparison (Figure 4.3), XORshift and Version 2-4 CI(XORshift, XORshift) PRNGs present the same values which are stable in a low level which never exceeds 1.2. Indeed, the new generators distributes very randomly the zeros and ones, whatever the length of the de-

Table 4.24: Comparison with Version 1 CI(XORshift,XORshift) for a 2×10^5 bits sequence(N = 32)

PRNGS		Classic	CI PRNG versions			
Method	The threshold values	XORshift	1	2	3	4
Monobit	3.84	1.71	2.77	0.33		
Serial	5.99	2.15	2.88	0.74		
Poker	316.91	248.93	222.36	262.82		
Runs	55.00	18.01	21.92	16.78		
hline Autocorrelation	1.64	0.50	0.02	0.08		
hline Time	Second	0.10s	0.41s	0.20s		

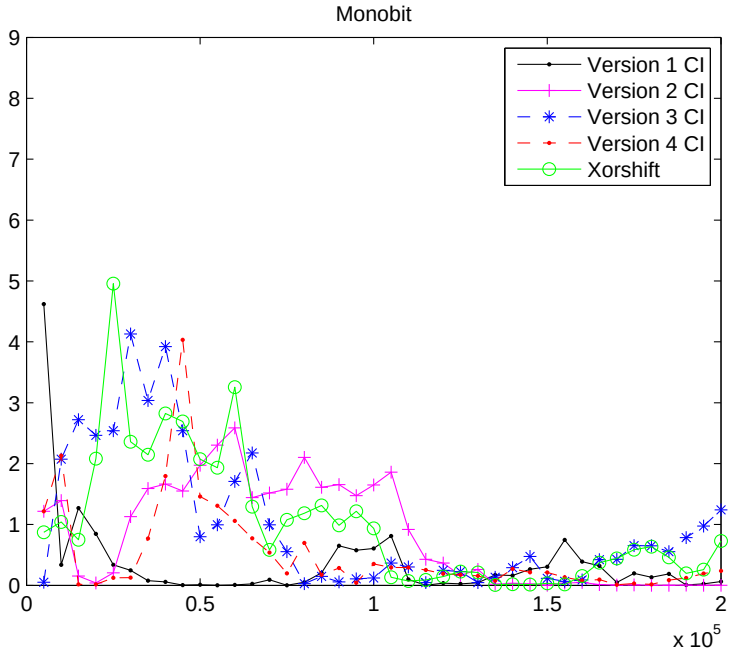


Figure 4.3: Comparison of monobits tests

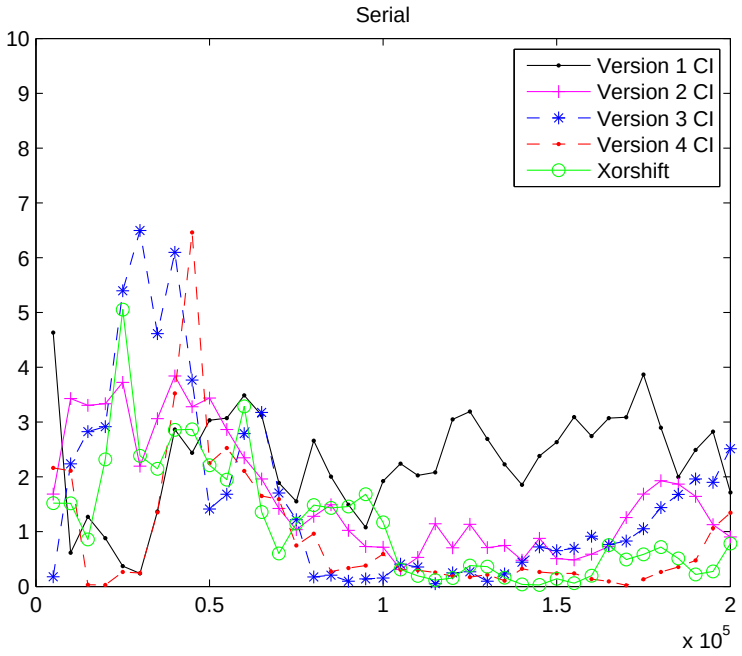


Figure 4.4: Comparison of serial tests

sired sequence. It can also be remarked that the XORshift generator presents the worst performance, but the values are within the standard boundary.

Figure 4.4 shows the serial test comparison. The CI generators outputs perform this test, during the length of $2 \times 10^4 to 5 \times 10^4$, Version 2 and 3 CI generators express a little overflow, except that all generators occurrences of 00, 01, 10, and 11 are very close to each other.

The poker test comparison with $m = 8$ is shown in Figure 4.5. In some length, the XORshift is the not very stable generator in this tests. And our CI version generators are good that their value are slower than the threshold. Indeed, the value of m and the length of the sequences should be enlarged to be certain that the chaotic iterations express totally their complex behavior. In that situation, the performances of our generators in the poker test can be improved.

The graph of the CI generators is the most stable one during the runs test comparison (Figure 4.6). Moreover, this trend is reinforced when the lengths of the tested sequences are increased.

The comparison of autocorrelation tests is presented in Figure 4.7. The CI generators clearly dominate these tests, whereas the score of the CI generators are all good.

To sum up we can claim that the CI generators, which newer version perform faster than its former version, outperforms all of the other generators in these statistical tests, especially when producing long output sequences.

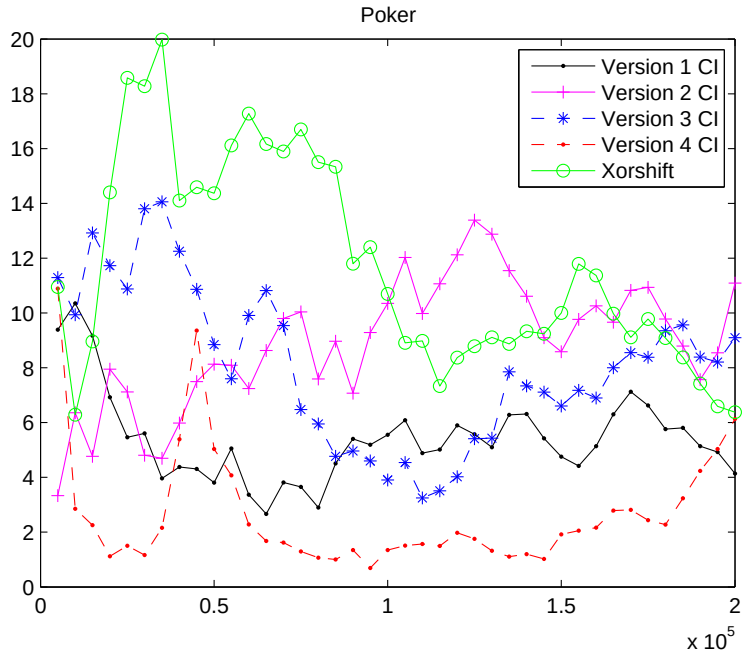


Figure 4.5: Comparison of poker tests

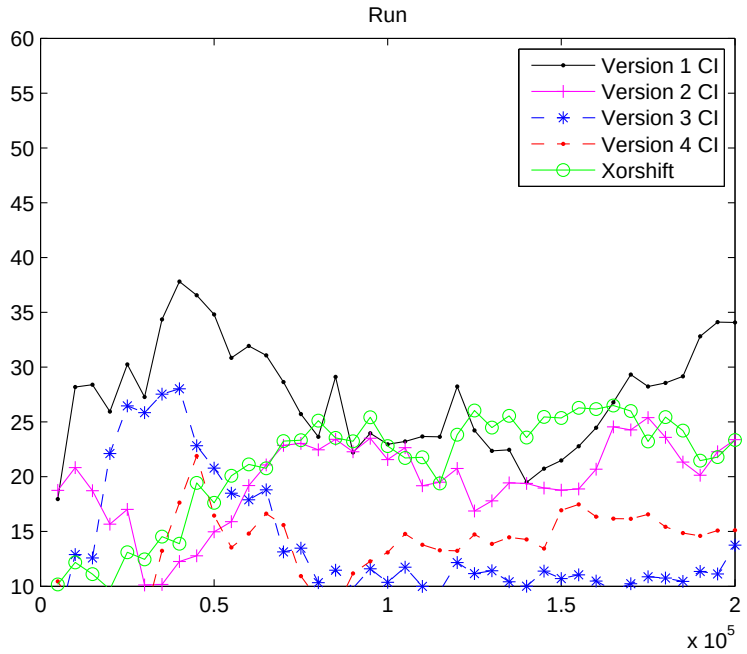


Figure 4.6: Comparison of runs tests

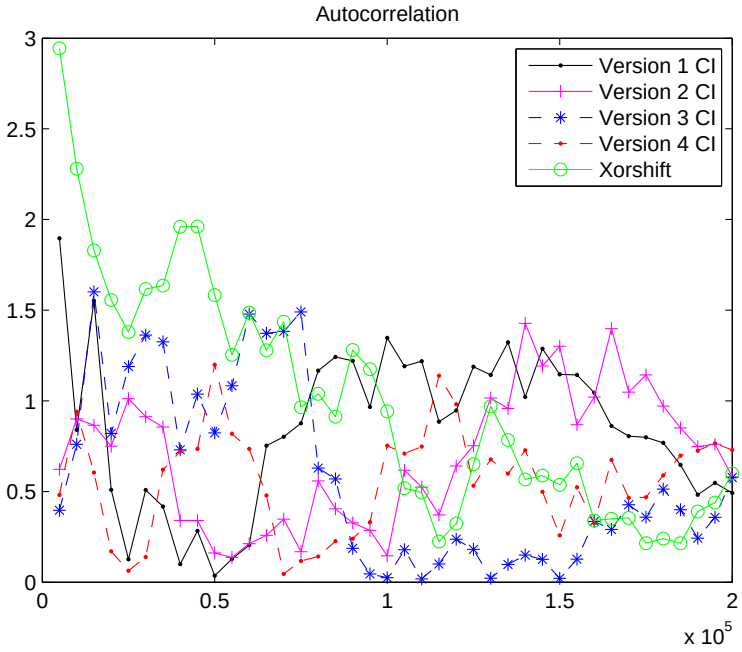


Figure 4.7: Comparison of autocorrelation tests

4.7.4 A flexible output

We assume that the initial state X is given as arrays of N-bit integers. Thus, the output size can be flexibly chosenell as N. Our PRNGs can generate discrete numbers where the number of states could not match to a power of 2, which is suitable for stochastic differential equations as an example.(An attractive property of discrete random numbers is that they require a small number of random bits–3 bits) [32]. Moreover, due to the fact that CI process is a simple bitwise change, the speed of output integers and binary numbers is almost the same.

In the following section, we will discuss the security level for various N by Statistical test. For both CI generator, various N can pass all the NIST and DIEHARD test. Table 4.25 gives the results derived from applying the TestU01 battery of tests to the PRNGs considered in this work. As observed,an conclude that the effective range of N for Version 2 CI is bigger than for Version 1 CI by TestU01. And also, this new scheme for obtaining a PRNG by combining two XORshift generators in CI give better properties than the old one (and the individual XORshift alone). It can be observed that the XORshift generator fails 146 tests.

Table 4.25: TestU01 Statistical Test

CI PRNG	Battery	N=2	N=4	N=8	N=16	N=32
Version 1 CI (XORshift,XORshift)	Rabbit	2	2	2	2	3
	Alphabit	0	0	0	2	2
	Pseudo DieHARD	0	0	0	0	0
	FIPS_140_2	0	0	0	0	0
	Small Crush	0	0	0	1	0
	Crush	4	4	9	16	46
	Big Crush	5	3	18	30	78
	Number of failures	11	9	29	51	129
Version 2 CI (XORshift,XORshift)	Rabbit 0	0	0	0	0	
	Alphabit	4	0	0	0	0
	Pseudo DieHARD	8	2	0	0	0
	FIPS_140_2	2	0	0	0	0
	Small Crush	0	0	0	0	0
	Crush	0	0	0	0	0
	Big Crush	0	0	0	0	0
	Number of failures	14	2	0	0	0

Chapter 5

An optimization technique on pseudorandom generators based on chaotic iterations

In this chapter, the statistical analysis of the three methods mentioned above are carried out systematically, and the results are discussed. Indeed PRNGs are often based on modular arithmetic, logical operations like bitwise exclusive or (XOR), and on circular shifts of bit vectors. However the security level of some PRNGs of this kind has been revealed inadequate by today's standards. Since different biased generators can possibly have their own side effects when inputted into our mixed generators, it is normal to enlarge the set of tested inputted PRNGs, to determine if the observed improvement still remains. We will thus show in this research work that the intended statistical improvement is really effective for all of these most famous generators.

5.1 About some Well-known PRNGs

5.1.1 Introduction

Knowing that there is no universal generator, it is strongly recommended to test a stochastic application with a large set of different PRNGs [49]. They can be classified in four major classes: linear generators, lagged generators, inversive generators, and mix generators:

- **Linear generators**, defined by a linear recurrence, are the most commonly analyzed and utilized generators. The main linear generators are LCGs and MLCG.
- **Lagged generators** have a general recursive formula that use various previously computed terms in the determination of the new sequence value.
- **Inversive congruential generators** form a recent class of generators that are based on the principle of congruential inversion.
- **Mixed generators** result from the need for sequences of better and better quality, or at least longer periods. This has led to mix different types of PRNGs, as follows: $x^i = y^i \oplus z^i$

For instance, inversive generators are very interesting for verifying simulation results obtained with a linear congruential generator (LCG), because their internal structure and

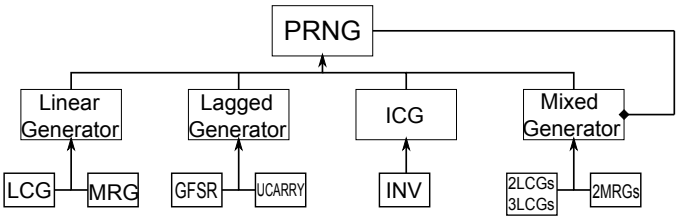


Figure 5.1: Ontological class hierarchy of PRNGs

correlation behavior strongly differs from what LCGs produce. Since these generators have revealed several issues, some scientists refrain from using them. In what follows, chaotic properties will be added to these PRNGs, leading to noticeable improvements observed by statistical test. Let us firstly explain with more details the generators studied in this research work (for a synthetic view, see Fig. 5.1).

5.1.2 Details of some Existing Generators

Here are the modules of PRNGs we have chosen to experiment.

5.1.2.1 LCG

This PRNG implements either the simple or the combined linear congruency generator (LCGs). The simple LCG is defined by the recurrence:

$$x^n = (ax^{n-1} + c) \; mod \; m \tag{1}$$

where a, c, and x^0 must be, among other things, non-negative and less than m [56]. In what follows, 2LCGs and 3LCGs refer as two (resp. three) combinations of such LCGs. For further details, see [34].

5.1.2.2 MRG

This module implements multiple recursive generators (MRGs), based on a linear recurrence of order k, modulo m [56]:

$$x^n = (a^1 x^{n-1} + \; ... \; + a^k x^{n-k}) \; mod \; m \tag{2}$$

Combination of two MRGs (referred as 2MRGs) is also be used in this paper.

5.1.2.3 UCARRY

Generators based on linear recurrences with carry are implemented in this module. This includes the add-with-carry (AWC) generator, based on the recurrence:

$$\begin{aligned} x^n &= (x^{n-r} + x^{n-s} + c^{n-1}) \; mod \; m, \\ c^n &= (x^{n-r} + x^{n-s} + c^{n-1})/m, \end{aligned} \tag{3}$$

the SWB generator, having the recurrence:

$$\begin{aligned} x^n &= (x^{n-r} - x^{n-s} - c^{n-1}) \; mod \; m, \\ c^n &= \begin{cases} 1 & \text{if } (x^{i-r} - x^{i-s} - c^{i-1}) < 0 \\ 0 & \text{else,} \end{cases} \end{aligned} \tag{4}$$

and the SWC generator designed by R. Couture, which is based on the following recurrence:

$$\begin{aligned} x^n &= (a^1 x^{n-1} \oplus \; ... \; \oplus a^r x^{n-r} \oplus c^{n-1}) \; mod \; 2^w, \\ c^n &= (a^1 x^{n-1} \oplus \; ... \; \oplus a^r x^{n-r} \oplus c^{n-1}) \; / \; 2^w. \end{aligned} \tag{5}$$

5.1.2.4 GFSR

This module implements the generalized feedback shift register (GFSR) generator, that is:

$$x^n = x^{n-r} \oplus x^{n-k} \tag{6}$$

5.1.2.5 INV

Finally, this module implements the nonlinear inversive generator, as defined in [56], which is:

$$x^n = \begin{cases} (a^1 + a^2 / z^{n-1}) \; mod \; m & \text{if } z^{n-1} \neq 0 \\ a^1 & \text{if } z^{n-1} = 0. \end{cases} \tag{7}$$

5.2 statistical tests

Considering the properties of binary random sequences, various statistical tests can be designed to evaluate the assertion that the sequence is generated by a perfectly random source. We have performed some statistical tests for the CIPRNGs proposed here. These tests include NIST suite [53] and DieHARD battery of tests [37]. For completeness and for reference, we give in the following subsection a brief description of each of the aforementioned tests.

5.2.1 NIST statistical tests suite

Among the numerous standard tests for pseudo-randomness, a convincing way to show the randomness of the produced sequences is to confront them to the NIST (National Institute of Standards and Technology) statistical tests, being an up-to-date tests suite proposed by the Information Technology Laboratory (ITL). A new version of the Statistical tests suite has been released in August 11, 2010.

The NIST tests suite SP 800-22 is a statistical package consisting of 15 tests. They were developed to test the randomness of binary sequences produced by hardware or software based cryptographic pseudorandom number generators. These tests focus on a variety of different types of non-randomness that could exist in a sequence.

For each statistical test, a set of $P - values$ (corresponding to the set of sequences) is produced. The interpretation of empirical results can be conducted in various ways. In this

paper, the examination of the distribution of P-values to check for uniformity ($P - value_T$) is used. The distribution of $P - values$ is examined to ensure uniformity. If $P - value_T \geqslant 0.0001$, then the sequences can be considered to be uniformly distributed.

In our experiments, 100 sequences (s = 100), each with 1,000,000-bit long, are generated and tested. If the $P - value_T$ of any test is smaller than 0.0001, the sequences are considered to be not good enough and the generating algorithm is not suitable for usage.

5.2.2 DieHARD battery of tests

The DieHARD battery of tests has been the most sophisticated standard for over a decade. Because of the stringent requirements in the DieHARD tests suite, a generator passing this battery of tests can be considered good as a rule of thumb.

The DieHARD battery of tests consists of 18 different independent statistical tests. This collection of tests is based on assessing the randomness of bits comprising 32-bit integers obtained from a random number generator. Each test requires 2^{23} 32-bit integers in order to run the full set of tests. Most of the tests in DieHARD return a $P - value$, which should be uniform on $[0, 1)$ if the input file contains truly independent random bits. These $P - values$ are obtained by $P = F(X)$, where F is the assumed distribution of the sample random variable X (often normal). But that assumed F is just an asymptotic approximation, for which the fit will be worst in the tails. Thus occasional $P - values$ near 0 or 1, such as 0.0012 or 0.9983, can occur. An individual test is considered to be failed if the $P - value$ approaches 1 closely, for example $P > 0.9999$.

5.3 Results and discussion

Table 5.1 shows the results on the batteries recalled above, indicating that almost all the PRNGs cannot pass all their tests. In other words, the statistical quality of these PRNGs cannot fulfill the up-to-date standards presented previously. We will show that the CIPRNG can solve this issue.

To illustrate the effects of this CIPRNG in detail, experiments will be divided in three parts:

1. **Single CIPRNG**: The PRNGs involved in CI computing are of the same category.

2. **Mixed CIPRNG**: Two different types of PRNGs are mixed during the chaotic iterations process.

3. **Multiple CIPRNG**: The generator is obtained by repeating the composition of the iteration function as follows: $x^0 \in \mathbb{B}^{\mathsf{N}}$, and $\forall n \in \mathbb{N}^*, \forall i \in [\![1; \mathsf{N}]\!]$,

$$x_i^n = \begin{cases} x_i^{n-1} & \text{if } S^n \neq i \\ \forall j \in [\![1; \mathsf{m}]\!], f^m(x^{n-1})_{S^{nm+j}} & \text{if } S^{nm+j} = i. \end{cases} \tag{8}$$

   m is called the *functional power*.

Table 5.1: NIST and DieHARD tests suite passing rates for PRNGs without CI

Types of PRNGs	Linear PRNGs		Lagged PRNGs				ICG PRNGs	Mixed PRNGs		
PRNG / *Tests*	LCG	MRG	AWC	SWB	SWC	GFSR	INV	LCG2	LCG3	MRG2
NIST	11/15	14/15	**15/15**	**15/15**	14/15	14/15	14/15	14/15	14/15	14/15
DieHARD	16/18	16/18	15/18	16/18	**18/18**	16/18	16/18	16/18	16/18	16/18

We have performed statistical analysis of each of the aforementioned CIPRNGs. The results are reproduced in Tables 5.1 and 5.2. The scores written in boldface indicate that all the tests have been passed successfully, whereas an asterisk "*" means that the considered passing rate has been improved.

5.3.1 Tests based on the Single CIPRNG

The statistical tests results of the PRNGs using the single CIPRNG method are given in Table 5.2. We can observe that, except for the Xor CIPRNG, all of the CIPRNGs have passed the 15 tests of the NIST battery and the 18 tests of the DieHARD one. Moreover, considering these scores, we can deduce that both the single Old CIPRNG and the single New CIPRNG are relatively steadier than the single Xor CIPRNG approach, when applying them to different PRNGs. However, the Xor CIPRNG is obviously the fastest approach to generate a CI random sequence, and it still improves the statistical properties relative to each generator taken alone, although the test values are not as good as desired.

Therefore, all of these three ways are interesting, for different reasons, in the production of pseudorandom numbers and, on the whole, the single CIPRNG method can be considered to adapt to or improve all kinds of PRNGs.

To have a realization of the Xor CIPRNG that can pass all the tests embedded into the NIST battery, the Xor CIPRNG with multiple functional powers are investigated in Section 5.3.3.

5.3.2 Tests based on the Mixed CIPRNG

To compare the previous approach with the CIPRNG design that uses a Mixed CIPRNG, we have taken into account the same inputted generators than in the previous section. These inputted couples $(PRNG_1, PRNG_2)$ of PRNGs are used in the Mixed approach as follows:

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus PRNG_1 \oplus PRNG_2, \end{cases} \tag{9}$$

With this Mixed CIPRNG approach, both the Old CIPRNG and New CIPRNG continue to pass all the NIST and DieHARD suites. In addition, we can see that the PRNGs using a Xor CIPRNG approach can pass more tests than previously. The main reason of this success is that the Mixed Xor CIPRNG has a longer period. Indeed, let n_P be the period of a PRNG P, then the period deduced from the single Xor CIPRNG approach is obviously equal to:

$$n_{SXORCI} = \begin{cases} n_P & \text{if } x^0 = x^{n_P} \\ 2n_P & \text{if } x^0 \neq x^{n_P}. \end{cases} \tag{10}$$

Let us now denote by n_{P1} and n_{P2} the periods of respectively the $PRNG_1$ and $PRNG_2$ generators, then the period of the Mixed Xor CIPRNG will be:

$$n_{XXORCI} = \begin{cases} LCM(n_{P1}, n_{P2}) & \text{if } x^0 = x^{LCM(n_{P1}, n_{P2})} \\ 2LCM(n_{P1}, n_{P2}) & \text{if } x^0 \neq x^{LCM(n_{P1}, n_{P2})}. \end{cases} \tag{11}$$

Table 5.2: NIST and DieHARD tests suite passing rates for PRNGs with CI

Types of PRNGs	Linear PRNGs		Lagged PRNGs				ICG PRNGs	Mixed PRNGs		
Single CIPRNG / *Tests*	LCG	MRG	AWC	SWB	SWC	GFSR	INV	LCG2	LCG3	MRG2
Old CIPRNG										
NIST	**15/15** *	**15/15** *	**15/15**	**15/15**	**15/15** *	**15/15** *	**15/15** *	**15/15** *	**15/15** *	**15/15**
DieHARD	**18/18** *	**18/18** *	**18/18** *	**18/18** *	**18/18**	**18/18** *	**18/18** *	**18/18** *	**18/18** *	**18/18** *
New CIPRNG										
NIST	**15/15** *	**15/15** *	**15/15**	**15/15**	**15/15** *	**15/15** *	**15/15** *	**15/15** *	**15/15** *	**15/15**
DieHARD	**18/18** *	**18/18** *	**18/18** *	**18/18** *	**18/18**	**18/18** *	**18/18** *	**18/18** *	**18/18** *	**18/18** *
Xor CIPRNG										
NIST	14/15*	**15/15** *	**15/15**	**15/15**	14/15	**15/15** *	14/15	**15/15** *	**15/15** *	**15/15**
DieHARD	16/18	16/18	17/18*	**18/18** *	**18/18**	**18/18** *	16/18	16/18	16/18	16/18

In Table 5.3, we only show the results for the Mixed CIPRNGs that cannot pass all DieHARD suites (the NIST tests are all passed). It demonstrates that Mixed Xor CIPRNG involving LCG, MRG, LCG2, LCG3, MRG2, or INV cannot pass the two following tests, namely the "Matrix Rank 32x32" and the "COUNT-THE-1's" tests contained into the DieHARD battery. Let us recall their definitions:

- **Matrix Rank 32x32.** A random 32x32 binary matrix is formed, each row having a 32-bit random vector. Its rank is an integer that ranges from 0 to 32. Ranks less than 29 must be rare, and their occurences must be pooled with those of rank 29. To achieve the test, ranks of 40,000 such random matrices are obtained, and a chisquare test is performed on counts for ranks 32,31,30 and for ranks ≤ 29.
- **COUNT-THE-1's TEST** Consider the file under test as a stream of bytes (four per 2 bit integer). Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte: 0,1, or 2 yield A, 3 yields B, 4 yields C, 5 yields D and 6,7, or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (over-lapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test: Q5-Q4, the difference of the naive Pearson sums of $(OBS - EXP)^2/EXP$ on counts for 5- and 4-letter cell counts.

The reason of these fails is that the output of LCG, LCG2, LCG3, MRG, and MRG2 under the experiments are in 31-bit. Compare with the Single CIPRNG, using different PRNGs to build CIPRNG seems more efficient in improving random number quality (mixed Xor CI can 100% pass NIST, but single cannot).

5.3.3 Tests based on the Multiple CIPRNG

Until now, the combination of at most two input PRNGs has been investigated. We now regard the possibility to use a larger number of generators to improve the statistics of the generated pseudorandom numbers, leading to the multiple functional power approach. For the CIPRNGs which have already pass both the NIST and DieHARD suites with 2 inputted PRNGs (all the Old and New CIPRNGs, and some of the Xor CIPRNGs), it is not meaningful to consider their adaption of this multiple CIPRNG method, hence only the Multiple Xor CIPRNGs, having the following form, will be investigated.

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus S^{nm} \oplus S^{nm+1} \ldots \oplus S^{nm+m-1}, \end{cases} \tag{12}$$

The question is now to determine the value of the threshold m (the functional power) making the multiple CIPRNG being able to pass the whole NIST battery. Such a question is answered in Table 5.4.

Table 5.3: Scores of mixed Xor CIPRNGs when considering the DieHARD battery

$PRNG_1$ \ $PRNG_0$	LCG	MRG	INV	LCG2	LCG3	MRG2
LCG		16/18	16/18	16/18	16/18	16/18
MRG	16/18		16/18	16/18	16/18	16/18
INV	16/18	16/18		16/18	16/18	16/18
LCG2	16/18	16/18	16/18		16/18	16/18
LCG3	16/18	16/18	16/18	16/18		16/18
MRG2	16/18	16/18	16/18	16/18	16/18	

Table 5.4: Functional power m making it possible to pass the whole NIST battery

Inputted *PRNG*	LCG	MRG	SWC	GFSR	INV	LCG2	LCG3	MRG2
Threshold value m	19	7	2	1	11	9	3	4

5.3.4 Results Summary

We can summarize the obtained results as follows.

1. The CIPRNG method is able to improve the statistical properties of a large variety of PRNGs.

2. Using different PRNGs in the CIPRNG approach is better than considering several instances of one unique PRNG.

3. The statistical quality of the outputs increases with the functional power m.

In this chapter, we first have formalized the CI methods that has been already presented in previous Internet conferences. These CI methods are based on iterations that have been topologically proven as chaotic. Then 10 usual PRNGs covering all kinds of generators have been applied, and the NIST and DieHARD batteries have been tested. Analyses show that PRNGs using the CIPRNG methods do not only inherit the chaotic properties of the CI iterations, they also have improvements of their statistics. This is why CIPRNG techniques should be considered as post-treatments on pseudorandom number generators to improve both their randomness and security.

Chapter 6

FPGA Acceleration of CIPRNGs

In this chapter, the proposition is to improve widely the efficient of the formerly proposed generators, without any lack of chaos properties. To do that, Field programmable gate array (FPGA) method is proposed.

6.1 introduction

Nowadays, FPGAs are very successfully applied to implement random sequence generation due to its ability in highly parallelizable task [23, 25, 58]. Such device allows us to generate pseudorandom numbers with remarkable speed. In these and other implementations, FPGA has advantages in performance, design time, power consumption, flexibility, cost and so on.

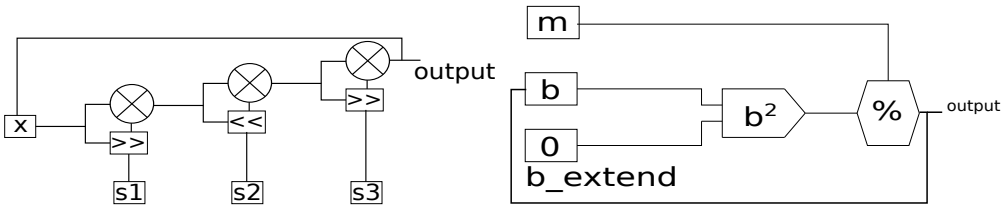
According to the previous chapters' description, it has proven that the chaotic iterations (CIs), to be a very decent tool for computing iterative algorithms, satisfies the chaos property, as it is defined by Devaney. The chaotic behaviour of CIs is developed attempt to obtain an unpredictable PRNG, in [16], its efficient implementations on GPU have been designed in, expressing that a very large quantity of pseudorandom numbers can be generated per second. Here, generators based on chaotic iteration is designed specifically for FPGA hardware, with that, the rate of generation can be hugely increased.

6.2 FPGA design

In order to take benefits from the computing power of FPGA, a whole processing needs to spread into several independent blocks of threads that can be computed simultaneously. In general, the larger the number of threads is, the more logistic elements of FPGA are used, and the less branching instructions are used (if, while, ...), the better the performances on FPGA is. Obviously, having these requirements in mind, suitable CIPRNG algorithms which produce random numbers with chaotic properties should be chosen to apply to FPGA. The Verilog-HDL [4] is used to help program.

6.2.1 CI Version selection

According to the comparison in Chapter 4, it can be seen that Version 4 CI algorithms is the most adaptable of all, its loop processing of them are both able to be replaced by parallel computing to increase the efficiency. To be noticed, the CIPRNGs used here are both proven to be cryptographically secure (check Section 3.4.3) by using BBS and three



(a) XORshift (b) BBS

Figure 6.1: The structure of processing for XORshift and BBS in FPGA each clock step

XORshift PRNGs, and its statistical performance is withstand the TestU01 test suite (Section 4.6).

6.2.2 Design of XORshift

The structure of PRNG XORshift designed in Verilog-HDL is shown in Figure 6.1(a), there are four inputs, first one is the initial state, which cost 64 bits of register units, then the other three ones are used to define how to shift, in FPGA, shifting cost no elements (just using different bit cell of the input). Then there are $64-s1+64-s2+64-s3 = 192-s1-s2-s3$ logic gates elements applied for XORshift processing. In program, we define each FPGA clock positive edge, the XORshift will work, since these are simple processing for FPGA, every clock step can lead to one output.

6.2.3 Design of BBS

Figure 6.1(b) gives the design of BBS in FPGA, there are two 32 bits length input: b and m, m is also a register stores the value of M which will not be changed, register b stores the every time state, when it is processing square computation, another register b_extend is used to combine with b to a 64 bits length data to avoid overflow. At last output is taking the three LSBs from the output of % as output. BBS performs the function while every positive edge of clock.

6.2.4 Design of the Version 4 CI

For composition of the CI PRNG, two XORshifts and BBS are connected to work together (Figure 6.2.4), as shown, the three bits of the output of BBS are the switches for the corresponding 32 bits from XORshift outputs. Every round of the CI RPNG processing costs two times clock positive edge to finish: In first clock, the four PRNGs are processed in parallel; Then in second clock, the results of the classic PRNGs are combined with the state to produce the output (32 bits binary).

In our experiment, type $EP2C8Q208C8$ from Altera company's CYCLONE II FPGA series is applied, it can give 50 MHz working clock frequency in default, with the help of phase-lock loop (PLL) device, the frequency could be increased into 400 MHz,then our generator based on CI can achieve $400(MHz) \div 2(times) \times 32(bits)$ which can achieve about

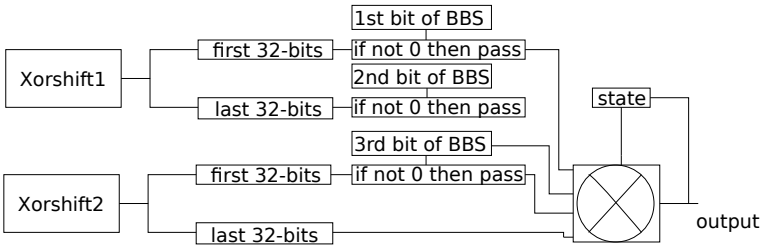


Figure 6.2: The structure of CI PRNG in FPGA

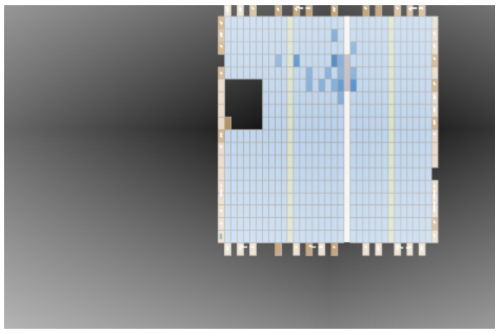


Figure 6.3: The sources cost in $EP2C8Q208C8$ FPGA board

6400 Mbits per second, and there are 100 logic elements used totally, that only takes 1% in $EP2C8Q208C8$ (100 from 8256 logic elements shown in Figure 6.3). In next section, an information hiding application is expressed by the FPGA device.

CHAPTER 7

Applications in cryptology

7.1 Application example of the use of the proposed PRNG

Cryptographically secure PRNGs are fundamental tools to communicate through the Internet. Original and encrypted image are shown in Figures 7.1(a) and 7.2(a), whereas Figure 7.1(b) and 7.2(b) depict their histograms. Obviously the distribution of the encrypted image is very close to the uniform distribution, which improves the protection against statistical attacks.

Figure 7.3 shows the correlation distribution of two horizontally adjacent pixels, both in the original and in the encrypted images. Correlation coefficients in the horizontal, vertical, and diagonal directions concerning these two images are presented in Table 7.1. Obviously, the correlation is important in the original image, whereas it is low and can be ignored in the encrypted image. These simple illustrations tend to prove that the use of CI PRNGs for cryptographic applications can be studied, to determine whether these chaotic generators is cryptographically secure working at the application or not. These study has been partially initiated in [14, 15, 11, 10], in which our generators have been used as a component of watermarking scheme. The robustness of this scheme has been evaluated, which has led to results as good as possible, thus reinforcing our opinion that these generators would probably be useful in cryptographic applications. The question of whether CI PRNGs are working application well or not, will thus be raised in our next work.

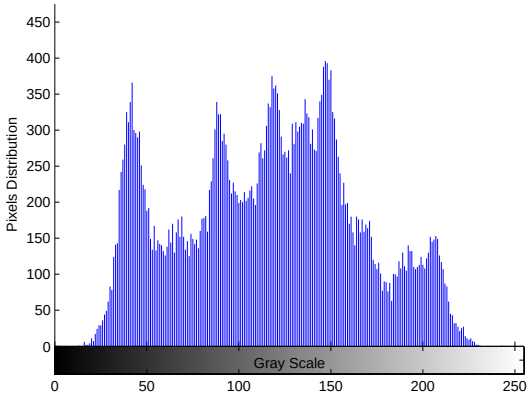
7.2 Introduction

Information hiding is now an integral part of Internet technologies. In the field of social search engines, for example, contents like pictures or movies are tagged with descriptive

Table 7.1: Correlation coefficients of two adjacent pixels in the original image and the encrypted image

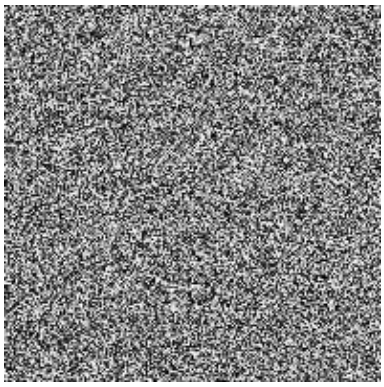
Direction	**Original image**	**Encrypted image**
Horizontal	0.9245	-0.0059
Vertical	0.9617	-0.0048
Diagonal	0.8967	-0.0052

(a) Original image.

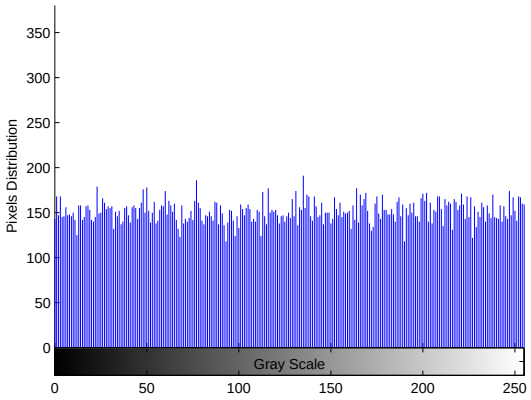


(b) Histogram.

Figure 7.1: Distribution of original image

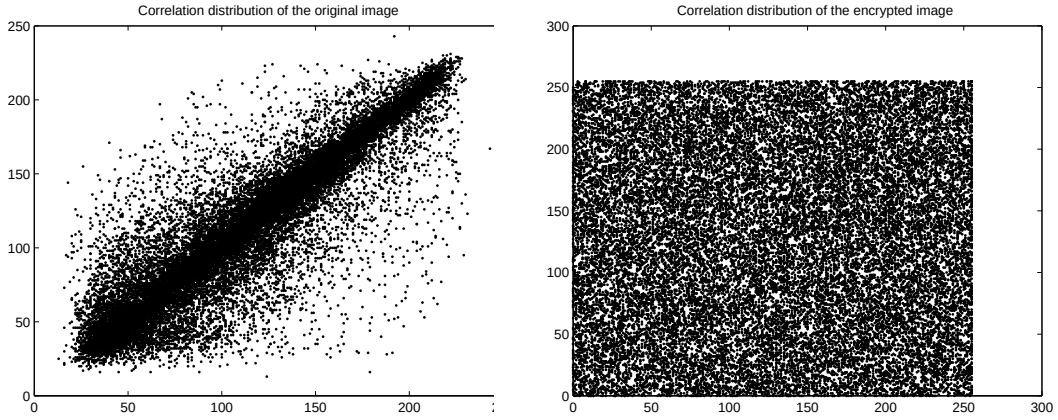


(a) Encrypted image.



(b) Histogram.

Figure 7.2: Distribution of encrypted image



(a) Original image. (b) Encrypted image.

Figure 7.3: Correlation distributions of two horizontally adjacent pixels

labels by contributors, and search results are determined by these descriptions. These collaborative taggings, used for example in Flickr [2] and Delicious [1] websites, contribute to the development of a Semantic Web, in which every Web page contains machine-readable metadata that describe its content. Information hiding technologies can be used for embedding these metadata. The advantage of its use is the possibility to realize social search without websites and databases: descriptions are directly embedded into media, whatever their formats. Robustness is required in this situation, as descriptions should resist to modifications like resizing, compression, and format conversion.

The Internet security field is also concerned by watermarking technologies. Steganography and cryptography are supposed to be used by terrorists to communicate through the Internet. Furthermore, in the areas of defense or in industrial espionage, many information leaks using steganographic techniques have been discovered. Lastly, watermarking is often cited as a possible solution to digital rights managements issues, to counteract piracy of digital work in an Internet based entertainment world [42].

7.3 Definition of a Chaos-Based Information Hiding Scheme

Let us now introduce our information hiding scheme based on CI generators.

7.3.1 Most and least significant coefficients

Let us define the notions of most and least significant coefficients of an image.

Definition 1 For a given image, most significant coefficients (in short MSCs), are coefficients that allow the description of the relevant part of the image, *i.e.*, its richest part (in terms of embedding information), through a sequence of bits.

For example, in a spatial description of a grayscale image, a definition of MSCs can be the sequence constituted by the first four bits of each pixel (see Figure 7.4). In a discrete

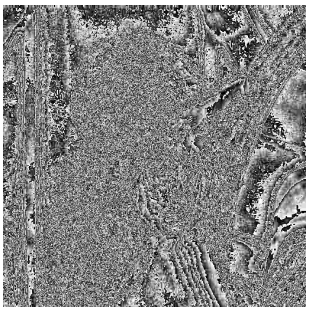
cosine frequency domain description, each 8×8 block of the carrier image is mapped onto a list of 64 coefficients. The energy of the image is mostly contained in a determined part of themselves, which can constitute a possible sequence of MSCs.

Definition 2 By least significant coefficients (LSCs), we mean a translation of some insignificant parts of a medium in a sequence of bits (insignificant can be understand as: "which can be altered without sensitive damages").

These LSCs can be, for example, the last three bits of the gray level of each pixel (see Figure 7.4). Discrete cosine, Fourier, and wavelet transforms can be used also to generate LSCs and MSCs. Moreover, these definitions can be extended to other types of media.

(a) Lena.

(b) MSCs of Lena.



(c) LSCs of Lena ($\times$17).

Figure 7.4: Example of most and least significant coefficients of Lena.

LSCs are used during the embedding stage. Indeed, some of the least significant coefficients of the carrier image will be chaotically chosen by using our PRNG. These bits will be either switched or replaced by the bits of the watermark. The MSCs are only useful in case of authentication; mixture and embedding stages depend on them. Hence, a coefficient should not be defined at the same time as a MSC and a LSC: the last can be altered while the first is needed to extract the watermark.

7.3.2 Stages of the scheme

Our CI generator-based information hiding scheme consists of two stages: (1) mixture of the watermark and (2) its embedding.

7.3.2.1 Watermark mixture

Firstly, for security reasons, the watermark can be mixed before its embedding into the image. A first way to achieve this stage is to apply the bitwise exclusive or (XOR) between the watermark and the CI generator. In this paper, we introduce a new mixture scheme based on chaotic iterations. Its chaotic strategy, which depends on our PRNG, will be highly sensitive to the MSCs, in the case of an authenticated watermarking.

7.3.2.2 Watermark embedding

Some LSCs will be switched, or substituted by the bits of the possibly mixed watermark. To choose the sequence of LSCs to be altered, a number of integers, less than or equal to the number M of LSCs corresponding to a chaotic sequence U, is generated from the chaotic strategy used in the mixture stage. Thus, the U^k-th least significant coefficient of the carrier image is either switched, or substituted by the k^{th} bit of the possibly mixed watermark. In case of authentication, such a procedure leads to a choice of the LSCs which are highly dependent on the MSCs [13].

On the one hand, when the switch is chosen, the watermarked image is obtained from the original image whose LSBs $L = \mathbb{B}^{\mathsf{M}}$ are replaced by the result of some chaotic iterations. Here, the iterate function is the vectorial boolean negation,

$$f_0 : (x_1, ..., x_{\mathsf{M}}) \in \mathbb{B}^{\mathsf{M}} \longmapsto (\overline{x_1}, ..., \overline{x_{\mathsf{M}}}) \in \mathbb{B}^{\mathsf{M}}, \tag{1}$$

the initial state is L, and the strategy is equal to U. In this case, the whole embedding stage satisfies the topological chaos properties [13], but the original medium is required to extract the watermark. On the other hand, when the selected LSCs are substituted by the watermark, its extraction can be done without the original cover (blind watermarking). In this case, the selection of LSBs still remains chaotic because of the use of the CI generator, but the whole process does not satisfy topological chaos [13]. The use of chaotic iterations is reduced to the mixture of the watermark. See the following sections for more detail.

7.3.2.3 Extraction

The chaotic strategy can be regenerated even in the case of an authenticated watermarking, because the MSCs have not changed during the embedding stage. Thus, the few altered LSCs can be found, the mixed watermark can be rebuilt, and the original watermark can be obtained. In case of a switch, the result of the previous chaotic iterations on the watermarked image should be the original cover. The probability of being watermarked decreases when the number of differences increase.

If the watermarked image is attacked, then the MSCs will change. Consequently, in case of authentication and due to the high sensitivity of our PRNG, the LSCs designed to receive the watermark will be completely different. Hence, the result of the recovery will have no similarity with the original watermark.

The chaos-based data hiding scheme is summed up in Figure 7.5.

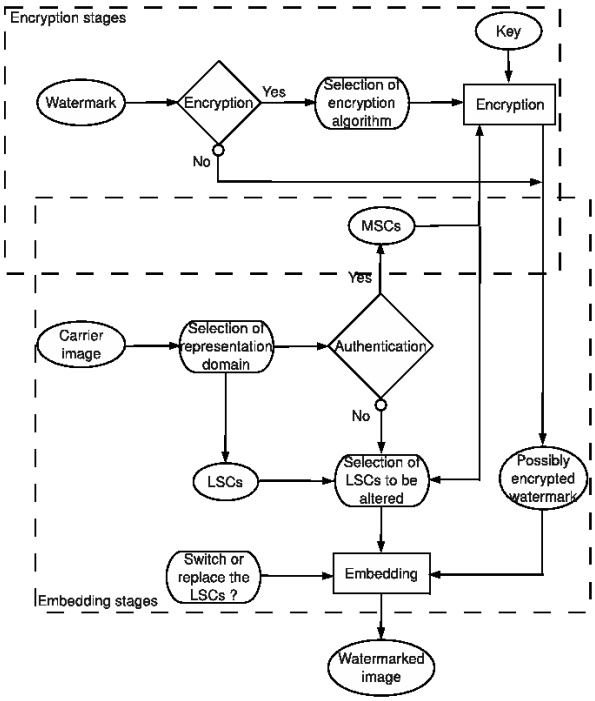


Figure 7.5: The chaos-based data hiding decision tree.

7.4 Software Implementation Experiment Protocol

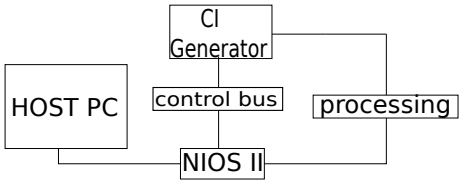
In this section, an information hiding application based on chaotic iterations is represented, a watermark is encrypted and embedded into a cover image using the scheme presented in the previous section and CIPRNG. The Version 4 CI PRNG (BBS, XORshift) is chosen, since its cryptographically secure property, good statistical performance and high efficiency. The carrier image is the well-known Lena, which is a 256 grayscale image, and the watermark is the 64×64 pixels binary image depicted in Figure 7.7.

We program our application via JAVA [3], where The watermark is encrypted by using chaotic iterations: the initial state x^0 is the watermark, considered as a boolean vector, the iteration function is the vectorial logical negation, and the chaotic strategy $(S^k)_{k\in\mathbb{N}}$ is defined with Version 4 CIs(BBS, XORshift), where initial parameters constitute the secret key and $N = 64$. The JAVA application is working as a windows interface(shown if Figure 7.4), firstly the embed file is chosen as Figure 7.4, then we choose the image which used to carry the contents (Figure 7.4), lastly after processing, the output image is produced, and in Figure 7.8, the difference between original image and encrypted image is also shown.

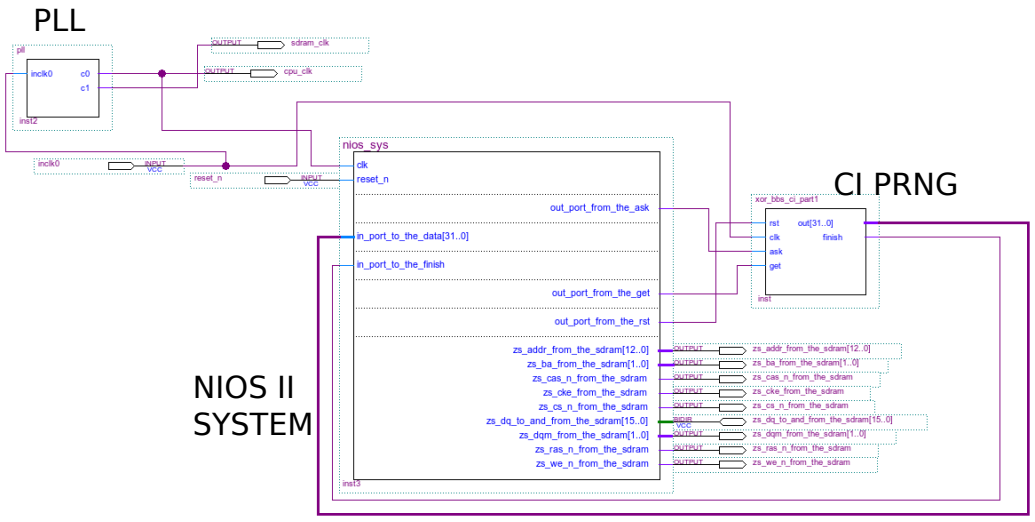
7.5 Hardware Implementation Experiment Protocol

In this section, the watermarking algorithm based on the chaotic PRNG presented above by FPGA is given, as an illustration of use of this PRNG based on CI.

Here the 32-bit embedded-processor architecture designed specifically for the Altera



(a) Structure



(b) Schematic

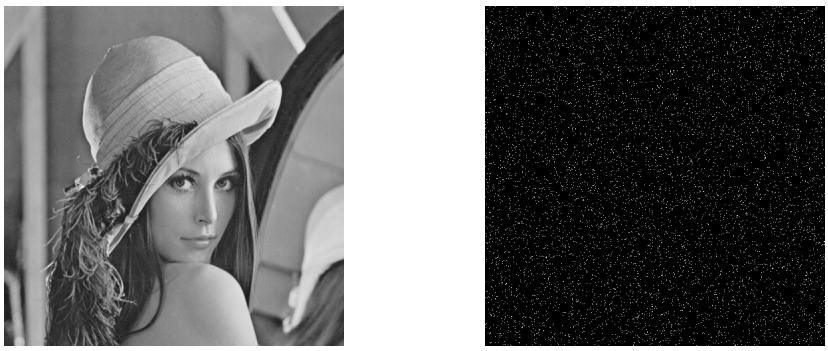
Figure 7.6: NIOS II setting in FPGA

(a) The original image

(b) The watermark

Figure 7.7: Original images

family of FPGAs is applied to execute this application. Nios II incorporates many enhancements over the original Nios architecture, making it more suitable for a wider range of embedded computing applications, from DSP to system-control [5]. Figure 7.6(a) shows the structure of this application, the NIOS II system can read the image from the HOST computer side, then according the control bus to operate the CI generator to produce random bits, and at last the processing results are transmitted back into the host. In Figure 7.6(b), the NIOS II is using the most powerful version the CYCLONE II can support(NIOS II/f),



(a) Output image (b) Difference

Figure 7.8: The output of watermarking application

then 4 KB on chip memory and 16 MB SDRAM are set, the *PLL* device is used to enhance the clock frequency from 50 to 200 MHz, the data bus connects NIOS II system and generator is in 32 bits.

7.6 Robustness evaluation

In what follows, the embedding domain is the spatial domain, Version 4 CI(BBS,XORshift) with parameters .... has been used to encrypt the watermark, MSCs are the four first bits of each pixel (useful only in case of authentication), and LSCs are the three next bits.

To prove the efficiency and the robustness of the proposed algorithm, some attacks are applied to our chaotic watermarked image. For each attack, a similarity percentage with the watermark is computed, this percentage is the number of equal bits between the original and the extracted watermark, shown as a percentage. Let us notice that a result less than or equal to 50% implies that the image has probably not been watermarked.

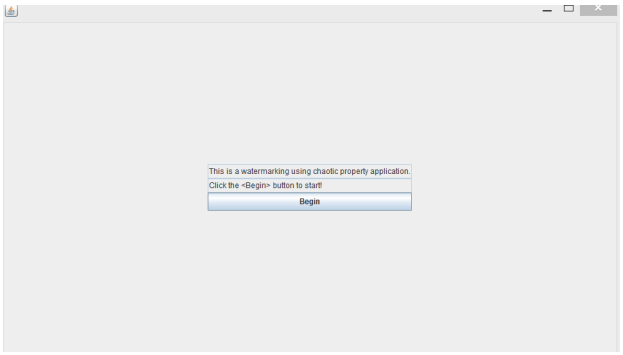
7.6.0.4 Zeroing attack

In this kind of attack, a watermarked image is zeroed, such as in Figure 7.10(a). In this case, the results in Table 1 have been obtained.

UNAUTHENTICATION		AUTHENTICATION	
Size (pixels)	Similarity	Size (pixels)	Similarity
10	99.31%	10	92.34%
50	98.55%	50	57.11%
100	92.40%	100	54.42%
200	71.01%	200	50.93%

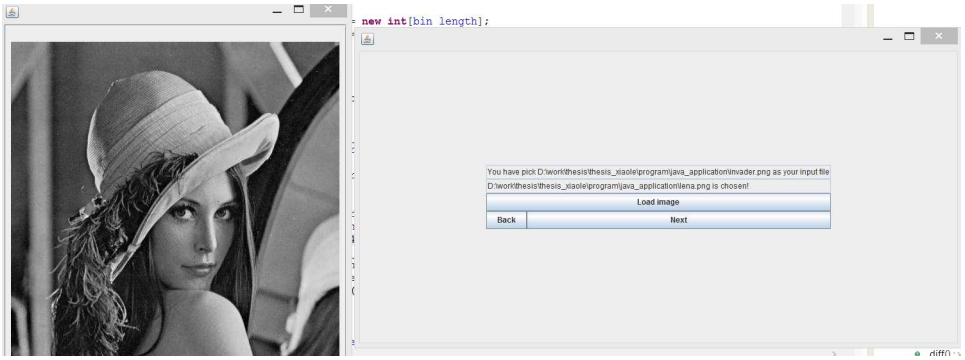
Table. 1. Cropping attacks

In Figure 7.11, the decrypted watermarks are shown after a crop of 50 pixels and after a crop of 10 pixels, in the authentication case.



(a) The start windows of JAVA watermarking Application

(b) Choosing the contends to be embed



(c) Choosing the contends to be embed

Figure 7.9: Java application for watermarking based on chaotic iteration generator

(a) Cropping attack

(b) Rotation attack

Figure 7.10: Watermarked Lena after attacks.

(a) Unauthentication (50×50).

(b) Authentication (50×50).

(c) Authentication (10×10).

Figure 7.11: Extracted watermark after a cropping attack.

By analyzing the similarity percentage between the original and the extracted watermark, we can conclude that in case of unauthentication, the watermark still remains after a zeroing attack: the desired robustness is reached. It can be noticed that zeroing sizes and percentages are rather proportional.

In case of authentication, even a small change of the carrier image (a crop by 10×10 pixels) leads to a really different extracted watermark. In this case, any attempt to alter the carrier image will be signaled, the image is well authenticated.

UNAUTHENTICATION		AUTHENTICATION	
Angle (degree)	Similarity	Angle (degree)	Similarity
2	97.31%	2	74.45%
5	94.02%	5	63.36%
10	89.98%	10	52.77%
25	80.84%	25	52.03%

Table. 2. Rotation attacks

7.6.0.5 Rotation attack

Let r_θ be the rotation of angle θ around the center $(128, 128)$ of the carrier image. So, the transformation $r_{-\theta} \circ r_\theta$ is applied to the watermarked image, which is altered as in Figure 7.10. The results in Table 2 have been obtained.

The same conclusion as above can be declaimed: this watermarking method satisfies the desired properties.

7.6.0.6 JPEG compression

A JPEG compression is applied to the watermarked image, depending on a compression level. Let us notice that this attack leads to a change of the representation domain (from spatial to DCT domain). In this case, the results in Table 3 have been obtained.

UNAUTHENTICATION		AUTHENTICATION	
Compression	Similarity	Compression	Similarity
2	85.92%	2	58.42%
5	70.45%	5	53.11%
10	64.39%	10	51.02%
20	53.94%	20	50.03%

Table. 3. JPEG compression attacks

A very good authentication through JPEG attack is obtained. As for the unauthentication case, the watermark still remains after a compression level equal to 10. This is a good result if we take into account the fact that we use spatial embedding.

7.6.0.7 Gaussian noise

Watermarked image can be also attacked by the addition of a Gaussian noise, depending on a standard deviation. In this case, the results in Table 4 have been obtained.

UNAUTHENTICATION		AUTHENTICATION	
Standard dev.	Similarity	Standard dev.	Similarity
1	81.00%	1	56.50%
2	77.18%	2	51.92%
3	67.01%	3	52.82%
5	59.28%	5	51.76%

Table. 4. Gaussian noise attacks

Once again we remark that good results are obtained, especially if we keep in mind that a spatial representation domain has been chosen.

Bibliography

[1] Delicious social bookmarking, http://delicious.com/. (Cited on page 97.)

[2] The frick collection, http://www.frick.org/. (Cited on page 97.)

[3] Java the programming language, http://www.java.com/en/. (Cited on page 100.)

[4] Verilog hdl. http://www.verilog.com/IEEEVerilog.html, 2008. Accessed: 30/09/2012. (Cited on page 91.)

[5] Introduction to the altera nios ii soft processor. http://coen.boisestate.edu/smloo/files/2011/11/, 2011. Accessed: 30/09/2012. (Cited on page 101.)

[6] Apostolis Argyris, Dimitris Syvridis, Laurent Larger, Valerio Annovazzi-Lodi, Pere Colet, Ingo Fischer, Jordi Garcia-Ojalvo, Claudio R. Mirasso, Luis Pesquera, and Alan K. Shore. Chaos–based communications at high bit rates using commercial fiber–optic links. *Nature*, 438:343–346, 2005. (Cited on page 1.)

[7] Jacques Bahi. Boolean totally asynchronous iterations. *Int. Journal of Mathematical Algorithms*, 1:331–346, 2000. (Cited on page 13.)

[8] Jacques Bahi, Jean-François Couchot, Christophe Guyeux, and Qianxue Wang. Class of trustworthy pseudo random number generators. In *INTERNET 2011, the 3-rd Int. Conf. on Evolving Internet*, pages ***–***, Luxembourg, Luxembourg, June 2011. To appear. (Cited on page 3.)

[9] Jacques Bahi, Xiaole Fang, and Christophe Guyeux. An optimization technique on pseudorandom generators based on chaotic iterations. In *INTERNET'2012, 4-th Int. Conf. on Evolving Internet*, pages 31–36, Venice, Italy, June 2012. (Cited on page 42.)

[10] Jacques Bahi, Xiaole Fang, and Christophe Guyeux. State-of-the-art in chaotic iterations based pseudorandom numbers generators application in information hiding. In *IHTIAP'2012, 1-st Workshop on Information Hiding Techniques for Internet Anonymity and Privacy*, pages 90–95, Venice, Italy, June 2012. (Cited on page 95.)

[11] Jacques Bahi, Xiaole Fang, Christophe Guyeux, and Qianxue Wang. Evaluating quality of chaotic pseudo-random generators. application to information hiding. *IJAS, International Journal On Advances in Security*, *(*):***–***, 2011. Accepted manuscript. To appear. (Cited on pages 3 and 95.)

[12] Jacques Bahi, Xiaole Fang, Christophe Guyeux, and Qianxue Wang. On the design of a family of CI pseudo-random number generators. In *WICOM'11, 7th Int. IEEE Conf. on Wireless Communications, Networking and Mobile Computing*, pages ***–***, Wuhan, China, September 2011. To appear. (Cited on page 3.)

[13] Jacques Bahi and Christophe Guyeux. Topological chaos and chaotic iterations, application to hash functions. *Neural Networks (IJCNN2010)*, pages 1–7, July 2010. (Cited on pages 3 and 99.)

[14] Jacques Bahi, Christophe Guyeux, and Qianxue Wang. Improving random number generators by chaotic iterations. application in data hiding. In *ICCASM 2010, Int. Conf. on Computer Application and System Modeling*, pages V13–643–V13–647, Taiyuan, China, October 2010. IEEE. to appear. (Cited on pages 3 and 95.)

[15] Jacques Bahi, Christophe Guyeux, and Qianxue Wang. A pseudo random numbers generator based on chaotic iterations. application to watermarking. In *WISM 2010, Int. Conf. on Web Information Systems and Mining*, volume 6318 of *LNCS*, pages 202–211, Sanya, China, October 2010. (Cited on pages 3 and 95.)

[16] Jacques M. Bahi, Raphaël Couturier, Christophe Guyeux, and Pierre-Cyrille Héam. Efficient and cryptographically secure generation of chaotic pseudorandom numbers on gpu. *CoRR*, abs/1112.5239, 2011. (Cited on pages 38, 39, 42 and 91.)

[17] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management part 1: General. In *NIST Special Publication 800-57, August 2005, National Institute of Standards and Technology. Available at http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf*, 2005. (Cited on page 8.)

[18] Thomas Beth, Dejan E. Lazic, and A. Mathias. *Cryptanalysis of Cryptosystems Based on Remote Chaos Replication*, pages 318–331. CRYPTO '94. Springer-Verlag, London, UK, 1994. (Cited on page 2.)

[19] Eli Biham. *Cryptanalysis of the chaotic-map cryptosystem suggested at EUROCRYPT'91*, pages 532–534. EUROCRYPT'91. Springer-Verlag, Berlin, Heidelberg, 1991. (Cited on page 2.)

[20] P. M. Binder and R. V. Jensen. Simulating chaotic behavior with finite-state machines. *Physical Review A*, 34(5):4460–4463, 1986. (Cited on page 12.)

[21] M. Blank. Discreteness and continuity in problems of chaotic dynamics. *Translations of Mathematical Monographs*, 161, 1997. (Cited on page 12.)

[22] Lenore Blum, Manuel Blum, and Michael Shub. A simple unpredictable pseudorandom number generator. *SIAM Journal on Computing*, 15:364–383, 1986. (Cited on page 13.)

[23] Slobodan Bojanic, Gabriel Caffarena, Slobodan Petrovic, and Octavio Nieto-Taladriz. Fpga for pseudorandom generator cryptanalysis. *Microprocessors and Microsystems*, 30(2):63 – 71, 2006. (Cited on page 91.)

[24] S. Cecen, R. M. Demirer, and C. Bayrak. A new hybrid nonlinear congruential number generator based on higher functional power of logistic maps. *Chaos, Solitons and Fractals*, 42:847–853, 2009. (Cited on page 2.)

[25] J. L. Danger, S. Guilley, and P. Hoogvorst. High speed true random number generator based on open loop structures in fpgas. *Microelectron. J.*, 40(11):1650–1656, November 2009. (Cited on page 91.)

[26] R. L. Devaney. *An Introduction to Chaotic Dynamical Systems*. Redwood City: Addison-Wesley, 2nd edition, 1989. (Cited on pages 3 and 12.)

[27] M. Falcioni, L. Palatella, S. Pigolotti, and A. Vulpiani. Properties making a chaotic system a good pseudo random number generator. *arXiv*, nlin/0503035, 2005. (Cited on page 2.)

[28] Christophe Guyeux and Jacques Bahi. Hash functions using chaotic iterations. *Journal of Algorithms & Computational Technology*, 4(2):167–182, 2010. (Cited on pages 2 and 3.)

[29] Toshiki Habutsu, Yoshifumi Nishio, Iwao Sasase, and Shinsaku Mori. *A secret key cryptosystem by iterating a chaotic map*, pages 127–140. EUROCRYPT'91. Springer-Verlag, Berlin, Heidelberg, 1991. (Cited on page 2.)

[30] R. J. Jenkins. Isaac. *Fast Software Encryption*, pages 41–49, 1996. (Cited on page 15.)

[31] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1998. (Cited on page 48.)

[32] Anthony J.C. Ladd. A fast random number generator for stochastic simulations. *Computer Physics Communications*, 180(11):2140–2142, 2009. (Cited on page 79.)

[33] Laurent Larger and John M. Dudley. Nonlinear dynamics: Optoelectronic chaos. *Nature*, 465(7294):41–42, 05 2010. (Cited on page 2.)

[34] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749, 1988. (Cited on page 82.)

[35] Pierre L'ecuyer. Comparison of point sets and sequences for quasi-monte carlo and for random number generation. *SETA*, pages 1–17, 2008. (Cited on page 1.)

[36] S. Li, G. Chen, and X. Mou. On the dynamical degradation of digital piecewise linear chaotic maps. *Bifurcation an Chaos*, 15(10):3119–3151, 2005. (Cited on page 12.)

[37] G. Marsaglia. Diehard battery of tests of randomness. Florida State University, 1995. (Cited on pages 3, 43 and 83.)

[38] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003. (Cited on page 14.)

[39] R. Matthews. On the derivation of a chaotic encryption algorithm. *Cryptologia*, 8:29–41, January 1984. (Cited on page 2.)

[40] A. Menezes, Paul C. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997. (Cited on pages 3 and 46.)

[41] Thomas E. Murphy and Rajarshi Roy. Chaotic lasers: The world's fastest dice. *Nature Photonics*, 2:714 – 715, 2008. (Cited on page 1.)

[42] Y. Nakashima, R. Tachibana, and N. Babaguchi. Watermarked movie soundtrack finds the position of the camcorder in a theater. *IEEE Transactions on Multimedia*, 2009. Accepted for future publication Multimedia. (Cited on page 97.)

[43] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000. (Cited on page 1.)

[44] J. Palmore and C. Herring. Computer arithmetic, chaos and fractals. *Physica D*, 42:99–110, 1990. (Cited on page 12.)

[45] Louis Pecora and Thomas Carroll. Synchronization in chaotic systems. *Physical Review Letters*, 64:821–824, 1990. (Cited on page 2.)

[46] Louis M. Pecora and Thomas L. Carroll. Synchronization in chaotic systems. *Physical Review Letters*, 64(8):821–824, 1990. (Cited on page 1.)

[47] L. Po-Han, C. Yi, P. Soo-Chang, and C. Yih-Yuh. Evidence of the correlation between positive lyapunov exponents and good chaotic random number sequences. *Computer Physics Communications*, 160:187–203, 2004. (Cited on page 2.)

[48] Bart Preneel, B. Preneel, Elisabeth Oswald, Alex Biryukov, E. Oswald, Bart Van Rompay, Sean Murphy, Louis Granboulan, Juliette White, Emmanuelle Dottax, S. Murphy, Alex Dent, J. White, Eli Biham, Elad Barkan, Orr Dunkelman, Markus Dichtl, Stefan Pyka, Markus Schafheutle, Håvard Raddum, Matthew Parker, Pascale Serf, E. Biham, E. Barkan, O. Dunkelman, J. -j. Quisquater, Mathieu Ciet, Francesco Sica, Lars Knudsen, M. Parker, and H. Raddum. Nessie d20 - nessie security report, 2003. (Cited on page 9.)

[49] David R.C. and Hill. Urng: A portable optimization technique for software applications requiring pseudo-random numbers. *Simulation Modelling Practice and Theory*, 11(7?C8):643 – 654, 2003. (Cited on page 81.)

[50] Gallager R.G. *Principles of Digital Communication*. Cambridge Univ. Press, 2008. (Cited on page 1.)

[51] F. Robert. *Discrete Iterations: A Metric Study*, volume 6 of *Springer Series in Computational Mathematics*. 1986. (Cited on page 13.)

[52] M. J. B. Robshaw. Stream ciphers, 1995. (Cited on page 9.)

[53] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, San Vo, Andrew Rukhin, Juan Soto, Miles Smid, Stefan Leigh, Mark Vangel, Alan Heckert, James Dray, and Lawrence E Bassham Iii. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2001. (Cited on pages 3, 7, 43 and 83.)

[54] H. G. Schuster. Deterministic chaos an introduction. *Physik Verlag*, Weinheim, 1984. (Cited on page 2.)

[55] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, Oktober 1949. (Cited on page 10.)

[56] Richard Simard and Université De Montréal. Testu01: A software library in ansi c for empirical testing of random number generators., 2002. (Cited on pages 3, 42, 43, 82 and 83.)

[57] D.R Stinson. *Cryptography: Theory and Practice*. CRC Press, 1995. (Cited on page 1.)

[58] K. H. Tsoi, K. H. Leung, and P. H. W. Leong. Compact fpga-based true and pseudo random number generators. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '03, pages 51–, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 91.)

[59] M. S. Turan, A Doganaksoy, and S Boztas. On independence and sensitivity of statistical randomness tests. *SETA 2008*, LNCS 5203:18–29, 2008. (Cited on pages 3 and 24.)

[60] A. Uchida, K. Amano, I. Inoue, K. Hirano, S. Naito, H. Someya, I. Oowada, T. Kurashige, M. Shiki, S. Yoshimori, K. Yoshimura, and P. Davis. Fast physical random bit generation with chaotic semiconductor lasers. *Nature Photonics*, 2:728 – 732, 2008. (Cited on page 1.)

[61] S. M. Ulam and J. V. Neumann. On combination of stochastic and deterministic processes. *Amer. Math. Soc.*, 53:1120, 1947. (Cited on page 14.)

[62] F. Montoya Vitini, J. Monoz Masque, and A. Peinado Dominguez. Bound for linear complexity of bbs sequences. *Electronics Letters*, 34:450–451, 1998. (Cited on pages 22 and 41.)

[63] Qianxue Wang, Jacques Bahi, Christophe Guyeux, and Xaole Fang. Randomness quality of CI chaotic generators. application to internet security. *INTERNET10*, pages 125–130, September 2010. (Cited on page 3.)

[64] Qianxue Wang, Christophe Guyeux, and Jacques Bahi. A novel pseudo-random generator based on discrete chaotic iterations for cryptographic applications. *INTERNET '09*, pages 71–76, 2009. (Cited on page 3.)

[65] D. D. Wheeler. Problems with chaotic cryptosystems. *Cryptologia*, XIII(3):243–250, 1989. (Cited on page 12.)

[66] Bin B. Zhu. Multimedia encryption. In Wenjun Zeng, Heather Yu, and Ching-Yung Lin, editors, *Multimedia Security Technologies for Digital Rights Management*, pages 75–109. Academic Press, Burlington, 2006. (Cited on page 1.)