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 [50]. 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 [30, 36, 43]. 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) [23]; 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) [45, 29]. 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 [33] and widely used for secure chaos communications [1]. 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 [23]. 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 [19, 17, 34]. 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 [40]. Hence in [22], 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) [32], 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 [12]. 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. [21, 13]. 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. [27].

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 [20]. 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 [42], NIST (National Institute of Standards and Technology of the U.S. Government) and DieHARD suites [39, 25], and Comparative test parameters [28]. 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 [44].

1.1 Research Background and Significance

1.2 Related work

In [20, 6], 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 [18]. Indeed, we have obtained this PRNG by combining chaotic iterations and two generators based on the logistic map in [48]. 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 [39].

Then, in the papers [7, 8], we have achieved to improve the speed of the former PRNG by replacing the two logistic maps: we used two XORshifts in [7], and ISAAC with XORshift in [8]. Additionally, we have shown that the first generator is able to pass DieHARD tests [7], whereas the second one can pass TestU01 [8].

In [47, 4], which is an extension of [48], 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 [3], 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 [5]. 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,...}

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. [39]

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), [11] 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 [35, 38]:

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 [41] 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 [14, 49, 31, 15, 24].

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 [18], 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 $d(f^n(x), f^n(y)) \geqslant \varepsilon$.

  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 [37]:

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

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.* [2]). For the general definition of such chaotic iterations, see, e.g. [37].

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 [16] (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 [?] 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:     $r \leftarrow 0$;
4: else
5:     $r \leftarrow 1$;
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 [26]. 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 [?]. 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 $i \in \{0,\dots,255\}$ as:

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

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** $i = 0, \dots, 255$ **do**
4:     $s \leftarrow x_i$;
5:     $a \leftarrow f(a,i) + x_{(i+128) \; mod \; 256}$;
6:     $x_i \leftarrow a + b + x_{(x \gg 2) \; mod \; 256}$;
7:     $r_i \leftarrow s + x_{(x_i \gg 10) \; mod \; 256}$;
8:     $b \leftarrow r_i$;
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. Chaotic iterations are realized as follows. Initial state $x^0 \in \mathbb{B}^{\mathsf{N}}$ is a boolean vector

Algorithm 5 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

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

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:

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

or the following integers:

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

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 $\mathcal{M}$ = {k, k+1} ($k \geqslant 3\mathsf{N}$ is recommended).

Chapter 3

Development of Version 1 CI Algorithm

In this chapter, some studies for Version 1 CI are deepen.

3.1 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 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...

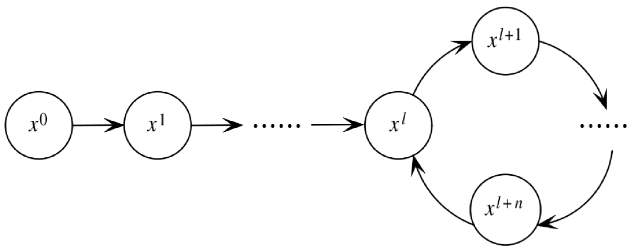


Figure 3.1: A pseudo orbit of a digital chaotic system

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}

- 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)...

3.2 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.

3.2.1 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.

$$\left\{\begin{array}{l} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}}, T \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ C = S^n \& 1 + 3 * N \\ w^0 = T^m \; mod \; N, w^1 = T^{m+1} \& 3, ...w^{C-1} = T^{m+C-1} \& 3 \\ d^n = (1 \ll w^0) \oplus (1 \ll w^1) \oplus ...(1 \ll w^{C-1}) \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus d^n, \end{array}\right. \quad (1)$$

3.2.2 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,

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

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.

3.3 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 3.2, this generator based on Version 1 CI can turn to be cryptographically secure

```
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

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 [46], 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.

Chapter 4

Version 2 CI Algorithm

4.1 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 4.2) 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 4.4). 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 4.3). So, the generator returns the following values:
Bits:

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

or States:

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

4.2 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 [44].

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 $y^0 = t$.

4.3 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 $m = i$:

$$P(m^n = i) = \frac{C_N^i}{2^N} \tag{1}$$

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

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

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:

$$m^n = g_2(y^n) = \begin{cases} N \text{ if } 0 \leqslant \frac{y^n}{2^{32}} < \frac{C_N^0}{2^N}, \\ N-1 \text{ if } \frac{C_N^0}{2^N} \leqslant \frac{y^n}{2^{32}} < \sum_{i=0}^{1} \frac{C_N^i}{2^N}, \\ N-2 \text{ if } \sum_{i=0}^{1} \frac{C_N^i}{2^N} \leqslant \frac{y^n}{2^{32}} < \sum_{i=0}^{2} \frac{C_N^i}{2^N}, \\ \vdots \quad \vdots \\ 0 \text{ if } \sum_{i=0}^{N-1} \frac{C_N^i}{2^N} \leqslant \frac{y^n}{2^{32}} < 1. \end{cases} \tag{3}$$

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 4.1(a) and Figure 4.1(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 4.1(c), whereas its uniformity is further justified by the tests presented in Section **??**.

4.4 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 $b = 142\underline{2}3341421\underline{122}34...$ 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 4.2 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 4.1, in which r means "reset" and the integers which are underlined in sequence b are discarded.

4.5 New CI(XORshift, XORshift) 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 2. Lastly, N is a constant defined by the user.

Algorithm 6 An arbitrary round of the new CI generator

Input: the internal state x (N bits)

Output: a state r of N bits

```
1:  for $i = 0, \dots, N$ do
2:      $d_i \leftarrow 0$
3:  $a \leftarrow PRNG1()$
4:  $m \leftarrow f(a)$
5:  $k \leftarrow m$
6:  while $i = 0, \dots, k$ do
7:      $b \leftarrow PRNG2()\ mod\ \mathsf{N}$
8:      $S \leftarrow b$
9:      if $d_S = 0$ then
10:         $x_S \leftarrow \overline{x_S}$
11:         $d_S \leftarrow 1$
12:     else if $d_S = 1$ then
13:         $k \leftarrow k + 1$
    $r \leftarrow x$ 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 **??** 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

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

4.6 Illustrative Example

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 2 can be adapted to this example as follows:

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

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 $(1, 0, 1, 1)$.

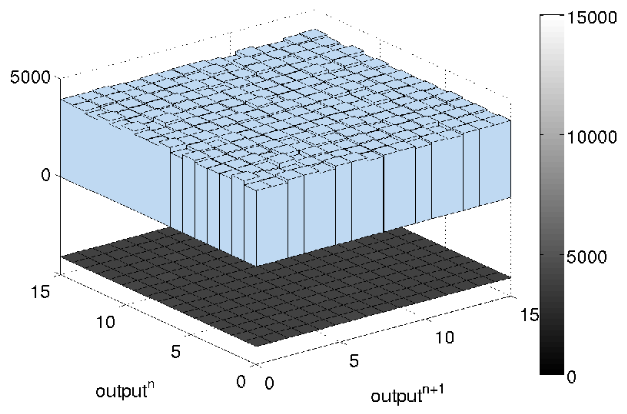
In this simulation, $m = 0, 4, 2, 2, 3, 4, 1, 1, 2, 3, 0, 1, 4, ...$ Additionally, b is computed with a XORshift generator too, but with another seed. We have found $b = 1, 4, 2, 2, 3, 3, 4, 1, 1, 4, 3, 2, 1, ...$

Chaotic iterations are made with initial state x^0, vectorial logical negation f_0, and strategy S. The result is presented in Table 4.1. 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...

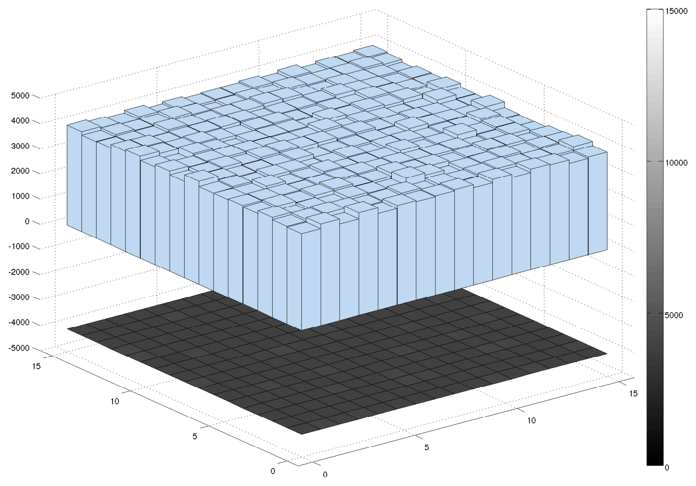
m	0	4	2	2
k	0	4 +1	2	2 +1
b		1 4 2 $\underline{2}$ 3	3 4	1 $\underline{1}$ 4
d	r	$r\begin{pmatrix}1\\0\\0\\0\end{pmatrix}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}\begin{pmatrix}1\\1\\0\\1\end{pmatrix}\quad\begin{pmatrix}1\\1\\1\\1\end{pmatrix}$	$r\begin{pmatrix}0\\0\\1\\0\end{pmatrix}\begin{pmatrix}0\\0\\1\\1\end{pmatrix}$	$r\begin{pmatrix}1\\0\\0\\0\end{pmatrix}\quad\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$
S		1 4 2 3	3 4	1 4
x^0	x^0	x^4	x^6	x^8
0	0	$\xrightarrow{1} 1$ 1	1	$\xrightarrow{1} 0$ 0
1	1	$\xrightarrow{2} 0$ 0	0	0
0	0	$\xrightarrow{3} 1$ 1	$\xrightarrow{3} 0$ 0	0
0	0	$\xrightarrow{4} 1$ 1	$\xrightarrow{4} 0$ 0	$\xrightarrow{4} 1$ 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: $x^0, x^4, x^6, x^8... = 4, 11, 8, 1...$

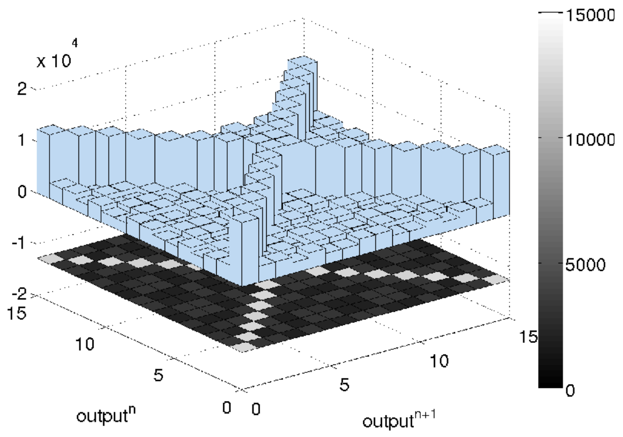
Table 4.1: Example of New CI(XORshift,XORshift) generation



(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 4.1: Histogram and intensity maps

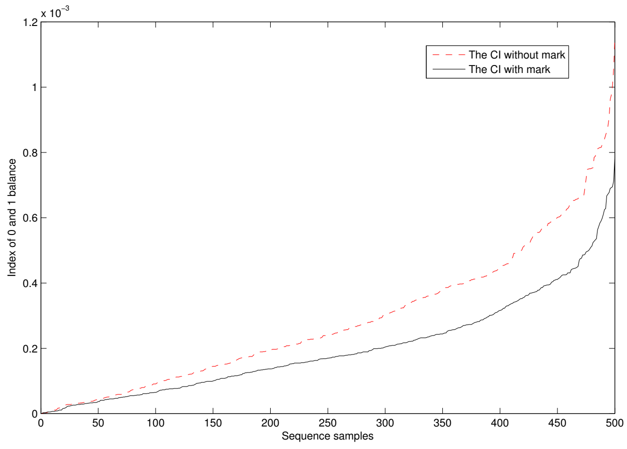


Figure 4.2: Balance property

CHAPTER 5

Version 3 LUT CI(XORshift,XORshift) algorithms

5.1 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 new 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 equivalency is to reduce the complexity of the proposed PRNG. Indeed the new CI(S,m) generator can be written as:

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

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.

5.2 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 8. 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 8 Generation of sequence b^n

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

Table 5.1: 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

This Algorithm 8 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 2 as follows.

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

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 2 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, a table can be given containing all the images of the integers into $[\![0, 15]\!]$ (see Table 5.1 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 9, which return a table $lut1$ such that $m^n = lut1[b^n]$.

Algorithm 9 The LUT-1 table generation

```
1: for $j = 0...N$ do
2:    $i = 0$
3:    while $i < C_N^j$ do
4:       $lut1[i] = j$
5:       $i = i + 1$
6: Return $lut1$
```

5.3 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:

$$w^n = XORshift2()\ mod\ C_N^m \tag{3}$$

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.

Algorithm 10 $LUT21$ procedure

```
1:  Procedure LUT21($M, N, b, v, c$)
2:  $count \leftarrow c$
3:  $value \leftarrow v$
4:  if $count == M$ then
5:      $lut2[M][num] = value$
6:      $num = num + 1$
7:  else
8:      for $i = b....N$ do
9:          $value = value + 2^i$
10:         $count = count + 1$
11:         Call recurse LUT21($M, N, i + 1, value, count$)
12:         $value = v$
13:         $count = c$
14: End Procedure
```

An example of such a LUT is shown in Table 5.2, when Algorithm 11 gives a concrete procedure to obtain such tables. This procedure makes recursive calls to the function $LUT21$ defined in Algorithm 10. 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. These parameters should be initialized as 0. For instance, the LUT presented in Table 5.2 is the $lut2$ obtained in Algorithm 10 with $N = 4$.

Algorithm 11 LUT-2 generation

```
1: for $i = 0....N$ do
2:     Call LUT21($i, N, 0, 0, 0$)
3: Return lut2
```

Table 5.2: 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		

5.4 LUT CI(XORshift,XORshift) Algorithm

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

$$x^n = x^{n-1} \wedge \mathcal{S}^n. \tag{4}$$

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 8 and $w^n = XORshift2()\ mod\ C_N^m$. An iteration of this generator is written in Algorithm 12. Let us finally remark that the two inputted XORshift can be replaced by any other operating PRNG.

Algorithm 12 LUT CI (XORshift,XORshift) algorithm

```
1: $c = n\ mod\ \lfloor 32/N \rfloor$
2: if $c = 0$ then
3:     $a \leftarrow XORshift1()$
4: $b^n \leftarrow (a \gg (N \times c))\&(2^N - 1)$
5: $m^n = lut1[b^n]$
6: $d^n = XORshift2()$
7: $w^n = b^n\ mod\ C_N^m$
8: $S^n = lut2[m][w]$
9: $x = x \wedge S^n$
10: Return $x$
```

5.5 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 9 provides the LUT-1 depicted in Table 5.1. The first XORshift generator has returned $y = 0, 11, 7, 2, 10, 4, 1, 0, 3, 9, \ldots$. By using this LUT, we obtain

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

Table 5.3: Example of a LUT CI(XORshift,XORshift) generation

$m = 0, 3, 2, 1, 2, 1, 1, 0, 1, 2, ...$ Then the Algorithm 11 is computed, leading to the LUT-2 given by Table 5.2.

So chaotic iterations of Algorithm 12 can be realized, to obtain in this example: 0100100101010001... or 4,9,5,1...

Bibliography

[1] 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.)

[2] Jacques Bahi. Boolean totally asynchronous iterations. *Int. Journal of Mathematical Algorithms*, 1:331–346, 2000. (Cited on page 11.)

[3] 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.)

[4] 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 page 3.)

[5] 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.)

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

[7] 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 page 3.)

[8] 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 page 3.)

[9] 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. (Not cited.)

[10] J. Banks, J. Brooks, G. Cairns, and P. Stacey. On devaney's definition of chaos. *Amer. Math. Monthly*, 99:332–334, 1992. (Not cited.)

[11] 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 6.)

[12] 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.)

[13] 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.)

[14] 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 10.)

[15] M. Blank. Discreteness and continuity in problems of chaotic dynamics. *Translations of Mathematical Monographs*, 161, 1997. (Cited on page 10.)

[16] Lenore Blum, Manuel Blum, and Michael Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15:364–383, 1986. (Cited on page 11.)

[17] 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.)

[18] R. L. Devaney. *An Introduction to Chaotic Dynamical Systems*. Redwood City: Addison-Wesley, 2nd edition, 1989. (Cited on pages 3 and 10.)

[19] 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.)

[20] 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.)

[21] 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.)

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

[23] 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.)

[24] 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 10.)

[25] G. Marsaglia. Diehard battery of tests of randomness. Florida State University, 1995. (Cited on page 3.)

[26] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003. (Cited on page 12.)

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

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

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

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

[31] J. Palmore and C. Herring. Computer arithmetic, chaos and fractals. *Physica D*, 42:99–110, 1990. (Cited on page 10.)

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

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

[34] 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.)

[35] 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 7.)

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

[37] F. Robert. *Discrete Iterations: A Metric Study*, volume 6 of *Springer Series in Computational Mathematics*. 1986. (Cited on page 11.)

[38] M. J. B. Robshaw. Stream ciphers, 1995. (Cited on page 7.)

[39] 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 and 5.)

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

[41] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, Oktober 1949. (Cited on page 8.)

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

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

[44] 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 23.)

[45] 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.)

[46] 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 page 22.)

[47] 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.)

[48] 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.)

[49] D. D. Wheeler. Problems with chaotic cryptosystems. *Cryptologia*, XIII(3):243–250, 1989. (Cited on page 10.)

[50] 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.)