Class of Trustworthy Pseudo-Random Number Generators

Jacques M. Bahi*, Jean-François Couchot*, Christophe Guyeux*and Qianxue Wang*
*University of Franche-Comte
Computer Science Laboratory LIFC, Belfort, France
Email:{jacques.bahi, jean-francois.couchot, christophe.guyeux, qianxue.wang}@univ-fcomte.fr

***Abstract*—With the widespread use of communication technologies, cryptosystems are therefore critical to guarantee security over open networks as the Internet. Pseudo-random number generators (PRNGs) are fundamental in cryptosystems and information hiding schemes. One of the existing chaos-based PRNGs is using chaotic iterations schemes. In prior literature, the iterate function is just the vectorial boolean negation. In this paper, we propose a method using Graph with strongly connected components as a selection criterion for chaotic iterate function. In order to face the challenge of using the proposed chaotic iterate functions in PRNG, these PRNGs are subjected to a statistical battery of tests, which is the well-known NIST in the area of cryptography.**

***Keywords*-Internet security; Chaotic sequences; Statistical tests; Discrete chaotic iterations.**

I. Introduction

Chaos and its applications in the field of secure communication have attracted a lot of attention in various domains of science and engineering during the last two decades. The desirable cryptographic properties of the chaotic maps such as sensitivity to initial conditions and random behavior have attracted the attention of researchers to develop new PRNG with chaotic properties. Recently, many scholars have made an effort to investigate chaotic PRNGs in order to promote communication security [5] [10] [14]. One of the existing chaos-based PRNGs is using chaotic iterations schemes.

A short overview of our recently proposed PRNGs based on Chaotic Iterations are given hereafter. In Ref. [1], 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 [7]. The chaotic behavior of CIs is exploited in [2], in order to obtain an unpredictable PRNG that depends on two logistic maps. 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 CIs, 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 [13]. Then, in the papers [3], [4], we have achieved to improve the speed of the former PRNG by replacing the two logistic maps: we used two XORshifts in [3], and ISAAC with XORshift in [4]. Additionally, we have shown that the first generator is able to pass DieHARD tests [11], whereas the second one can pass TestU01 [9].

In spite of the fact that all these previous algorithms are parametrized with the embed PRNG, they all iterate the same function namely, the vectorial boolean negation later denoted as $\neg$. It is then judicious to investigate whether other functions may replace the $\neg$ function in the above approach. In the positive case, the user should combine its own function and its own PRNGs to provide a new PRNG instance. 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.

The rest of this paper is organized in the following way. In the next section, some basic definitions concerning CIs are recalled. Then, our family of generators based on discrete CIs is presented in Section III with some improvements. Next, Section IV gives a characterization of functions whose iterations are chaotic. A practical note presents an algorithm allowing to generate some instances of such functions. These ones are then embedded in the algorithm presented in Sect. V where we show why generator of Sect. III is not convenient for them. In Section VI, various tests are passed with a goal to decide whether all chaotic functions are convenient in a PRNG context. The paper ends with a conclusion section where our contribution is summarized and intended future work is presented.

II. Discrete Chaotic Iterations: recalls

Let us denote by $[\![a;b]\!]$ the interval of integers: $\{a, a+1, \ldots, b\}$. A boolean system (BS) is a collection of n components. Each component $i \in [\![1;n]\!]$ takes its value x_i among the domain $\mathbb{B} = \{0,1\}$. A *configuration* of the system at discrete time t (also called at *iteration* t) is the vector $x^t = (x_1^t, \ldots, x_n^t) \in \mathbb{B}^n$.

The dynamics of the system is described according to a function $f : \mathbb{B}^n \to \mathbb{B}^n$ such that: $f(x) = (f_1(x), \ldots, f_n(x))$.

Let be given a configuration x. In what follows the configuration $N(i,x) = (x_1, \ldots, \overline{x_i}, \ldots, x_n)$ is obtained by switching the $i-$th component of x. Intuitively, x and $N(i,x)$ are neighbors. The discrete iterations of the f function are represented by the so called graph of iterations.

Definition 1 (Graph of iterations) *In the oriented* graph of iterations*, vertices are configurations of $\mathbb{B}^n$ and there is an arc labeled i from x to $N(i,x)$ iff $f_i(x)$ is $N(i,x)$ (we consider 1-bit transitions).*

In the sequel, the *strategy* $S = (S^t)^{t \in \mathbb{N}}$ is the sequence of the components that may be updated at time t, S^t denotes the $t-$th term of the strategy S.

Let us now introduce two important notations. Δ is the *discrete Boolean metric*, defined by $\Delta(x,y) = 0 \Leftrightarrow x = y$, and the function F_f is defined for any given application $f : \mathbb{B}^n \to \mathbb{B}^n$ by

$$\begin{array}{rcl} F_f : [\![1;n]\!] \times \mathbb{B}^n & \to & \mathbb{B}^n \\ (s,x) & \mapsto & \left(x_j.\Delta(s,j) + f_j(x).\overline{\Delta(s,j)}\right)_{j \in [\![1;n]\!]}, \end{array}$$

where the point and the line above delta are multiplication and negation respectively. With such a notation, configurations are defined for times $t = 0, 1, 2, \ldots$ by:

$$\begin{cases} x^0 \in \mathbb{B}^n \text{ and} \\ x^{t+1} = F_f(S^t, x^t) \end{cases} \quad (1)$$

Finally, iterations of (1) can be described by the following system

$$\begin{cases} X^0 = ((S^t)^{t\in\mathbb{N}}, x^0) \in [\![1; n]\!]^{\mathbb{N}} \times \mathbb{B}^n \\ X^{k+1} = G_f(X^k), \end{cases} \quad (2)$$

such that

$$G_f\left((S^t)^{t\in\mathbb{N}}, x\right) = \left(\sigma((S^t)^{t\in\mathbb{N}}), F_f(S^0, x)\right),$$

where σ is the function that returns the strategy $(S^t)^{t\in\mathbb{N}}$ where the first term (*i.e.*, S^0) has been removed. In other words, at the t^{th} iteration, only the S^t-th cell is modified; the resulting strategy is the initial one where the first t terms have been removed.

A previous work [1] has shown a fine metric space such that iterations of the map G_f are chaotic in the sense of Devaney [7] when f is the negation function $\neg$. This definition consists of three conditions: topological transitivity, density of periodic points, and sensitive point dependence on initial conditions. Topological transitivity is established when, for any element, any neighborhood of its future evolution eventually overlap with any other given region. On the contrary, a dense set of periodic points is an element of regularity that a chaotic dynamical system has to exhibit. This regularity "counteracts" the effects of transitivity. Finally, a system is sensitive to initial conditions if future evolution of any point in its neighborhood are significantly different. This result theoretically implies the "quality" of the randomness.

The next section formalizes with chaotic iterations terms the PRNG algorithm presented in [2].

III. Chaos based PRNG

This section aims at formalizing a PRNG algorithm already presented in [2] and gives some improvements.

First of all, Let us intorduce *XORshift*, generator. Xorshift is a category of pseudorandom number generators designed by George Marsaglia [12] that repeatedly uses the transform of exclusive or on a number with a bit shifted version of itself. A XORshift operation is defined as follows.

Input: the internal state z (a 32-bits word)
Output: y (a 32-bits word)
$z \leftarrow z \oplus (z \ll 13)$;
$z \leftarrow z \oplus (z \gg 17)$;
$z \leftarrow z \oplus (z \ll 5)$;
$y \leftarrow z$;
return y;

Algorithm 1: An arbitrary round of XORshift algorithm

Then the design procedure of this generator is summed up in Algorithm 2.

Let be given a seed as the internal state x. This algorithm outputs a random configuration x'. It is based on the *XORshift*, generator which is called in two situations. The first one occurs while generating the parameter of the *reallocate* function that aims at computing the number k of time a function

Input: an initial state x^0 (n bits)
Output: a state x (n bits)
$x \leftarrow x^0$;
$k \leftarrow$ *reallocate*(*XORshift*() $\mod (2^n - 1)$);
$x \leftarrow$ *iterate_G*(neg, *XORshift*, k, x);
return x;

Algorithm 2: An arbitrary round of the (*XORshift*,*XORshift*) generator

has to be iterated. The second one occurs as a parameter of *iterate_G*, which executes the iterations of G as defined in (2), with $f = neg, S = XORshift, x$ as initial state, and k for the number of iterations.

Firstly, let us focus on the *reallocate* function, which is defined by:

$$\textit{reallocate}(k) = \begin{cases} 0 & \text{if} \quad 0 \leqslant k < \binom{n}{0} \\ 1 & \text{if} \quad \binom{n}{0} \leqslant k < \sum_{i=0}^{1} \binom{n}{i} \\ \vdots & \qquad\qquad \vdots \\ n & \text{if} \quad \sum_{i=0}^{n-1} \binom{n}{i} \leqslant k \leqslant 2^n - 1 \end{cases}$$

Formally, the set $[\![0, 2^n - 1]\!]$ is partionned into subsets $[\![\Sigma_{i=0}^{j}\binom{n}{0}, \Sigma_{i=0}^{j+1}\binom{n}{i}[\![$ where $j \in [\![0, n-1]\!]$. Each interval bound is a binomial coefficient: it gives the number of combinations of n things taken j. In our context, it is the number of configurations $(x_1, \ldots, x_n)$ that can be built by negating j elements among n. The function *reallocate* allows to compute a distribution on $[\![0, n]\!]$ that permits to reach configurations in $[\![0, 2^n - 1]\!]$ uniformly.

Let us present now the *iterate_G* function. It starts with computing the strategy S of lenght k as the result of a usual *sample* (not detailled here) function that selects k elements among n following a PRNG r given as the first parameter. The loop next reproduces k iterations of G_f as define in Equ. (2)

Input: a function f, a PRNG r, an iterations number k, a binary number x^0 (n bits)
Output: a binary number x (n bits)
$x \leftarrow x^0$;
S = *sample*(r, k, n);
for $i = 0, \ldots, k-1$ **do**
  $s \leftarrow S[i]$;
  $x \leftarrow F_f(s, x)$;
end
return x;

Algorithm 3: The *iterate_G* function.

Compared to work [2], this algorithm is:

- close to the formal iterations of G_f: strategy is explicitely computed and there are as many iterations as the number of executed loops.
- more efficient: in the previous work, loops are executed untill k distinct elements have been switched leading to possibly more iterations. In the opposite, the function *iterate_G* exactly executes k loops when k iterations are awaited. However, this improvement moves the problem into the *sample* function, which is classically tuned to

 ISBN: 978-1-61208-141-0

	100	10000	100000	1000000	1000000
Speed up	10%	7.8 %	8.8 %	8.1%	9.5%

Table I: Speed up improvement from Algorithm [2]

speed up its global behavior. In such a context we take a benefit of this improvement. Table I compares these two algorithms in terms of execution time with respect to the number of generated elements. The improvement is about 9%.

However as noticed in introduction, the whole (theoretical and practical) approach is based on the negation function. The following section studies whether other functions can theoretically replace this one.

IV. Characterizing and Computing Functions for PRNG

This section presents other functions that theoretically could replace the negation function $\neg$ in the previous algorithms.

In this algorithm and from the graph point of view, iterating the function G_f from a configuration x^0 and according to a strategy $(S^t)^{t\in\mathbb{N}}$ consists in traversing the directed iteration graph $\Gamma(f)$ from a vertex x^0 following the edge labelled with S^0, S^1, ... Obviously, if some vertices cannot be reached from other ones, their labels expressed as numbers cannot be output by the generator. The *Strongly connected component of* $\Gamma(f)$ (*i.e.*, when there is a path from each vertex to every other one), denoted by SCC in the following [6], is then a necessary condition for the function f. The following result shows this condition is sufficient to make iterations of G_f chaotic.

Theorem 1 (Theorem III.6, p. 91 in [8]) *Let f be a function from $\mathbb{B}^n$ to $\mathbb{B}^n$. Then G_f is chaotic according to Devaney iff the graph $\Gamma(f)$ is strongly connected.*

Any function such that the graph $\Gamma(f)$ is strongly connected is then a candidate for being iterated in G_f for pseudo random number generating. Thus, let us show how to compute a map f with a strongly connected graph of iterations $\Gamma(f)$.

We first consider the negation function $\neg$. The iteration graph $\Gamma(\neg)$ is obviously strongly connected: since each configuration $(x_1,\ldots,x_n)$ may reach one of its n neighbors, there is then a bit by bit path from any $(x_1,\ldots,x_n)$ to any $(x'_1,\ldots,x'_n)$. Let then Γ be a graph, initialized with $\Gamma(\neg)$, the algorithm iteratively does the two following stages:

1) select randomly an edge of the current iteration graph Γ and
2) check whether the current iteration graph without that edge remains strongly connected (by a Tarjan algorithm [15], for instance). In the positive case the edge is removed from G,

until a rate r of removed edges is greater than a threshold given by the user.

Formally, if r is close to 0% (*i.e.*, few edges are removed), there should remain about $n\times 2^n$ edges (let us recall that 2^n is the amount of nodes). In the opposite case, if r is close to 100%, there are left about 2^n edges. In all the cases, this step returns the last graph Γ that is strongly connected. It is not then obvious to return the function f whose iteration graph is Γ.

Function f	$f(x)$, for x in $(0,1,2,\ldots,15)$	Rate
$\neg$	(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0)	0%
ⓐ	(15,14,13,12,11,10,9,8,7,6,7,4,3,2,1,0)	2.1%
ⓑ	(14,15,13,12,11,10,9,8,7,6,5,4,3,2,1,0)	4.1%
ⓒ	(15,14,13,12,11,10,9,8,7,7,5,12,3,0,1,0)	6.25%
ⓓ	(14,15,13,12,9,10,11,0,7,2,5,4,3,6,1,8)	16.7%
ⓔ	(11,2,13,12,11,14,9,8,7,14,5,4,1,2,1,9)	16.7%
ⓕ	(13,10,15,12,3,14,9,8,6,7,4,5,11,2,1,0)	20.9%
ⓖ	(13,7,13,10,11,10,1,10,7,14,4,4,2,2,1,0)	20.9%
ⓗ	(7,12,14,12,11,4,1,13,4,4,15,6,8,3,15,2)	50%
ⓘ	(12,0,6,4,14,15,7,15,11,1,14,2,7,4,7,9)	75%

Table II: Functions with SCC graph of iterations

However, such an approach suffers from generating many functions with similar behavior due to the similarity of their graph. More formally, let us recall the graph isomorphism definition that resolves this issue. Two directed graphs Γ_1 and Γ_2 are *isomorphic* if there exists a permutation p from the vertices of Γ_1 to the vertices of Γ_2 such that there is an arc from vertex u to vertex v in Γ_1 iff there is an arc from vertex $p(u)$ to vertex $p(v)$ in Γ_2.

Then, let f be a function, $\Gamma(f)$ be its iteration graph, and p be a permutation of vertices of $\Gamma(f)$. Since $p(\Gamma(f))$ and $\Gamma(f)$ are isomorphic, then iterating f (*i.e.*, traversing $\Gamma(f)$) from the initial configuration c amounts to iterating the function whose iteration graph is $p(\Gamma(f))$ from the configuration $p(c)$. Graph isomorphism being an equivalence relation, the sequel only consider the quotient set of functions with this relation over their graph. In other words, two functions are distinct if and only if their iteration graph are not isomorphic.

Table II presents generated functions that have been ordered by the rate of removed edges in their graph of iterations compared to the iteration graph $\Gamma(\neg)$ of the boolean negation function $\neg$.

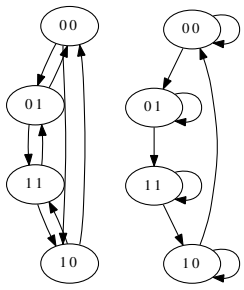
For instance let us consider the function ⓖ from $\mathbb{B}^4$ to $\mathbb{B}^4$ defined by the following images: $[13,7,13,10,11,10,1,10,7,14,4,4,2,2,1,0]$. In other words, the image of 3 (0011) by ⓖ is 10 (1010): it is obtained as the binary value of the fourth element in the second list (namely 10). It is not hard to verify that $\Gamma(ⓓ)$ is SCC. Next section gives practical evaluations of these functions.

V. Modifying the PRNG Algorithm

A coarse attempt could directly embed each function of table II in the *iterate_G* function defined in Algorithm 3. Let us show the drawbacks of this approach on a more simpler example.

Let us consider for instance n is two, the negation function on $\mathbb{B}^2$, and the function f defined by the list $[1,3,0,2]$ (i.e., $f(0,0)=(0,1), f(0,1)=(1,1), f(1,0)=(0,0)$, and $f(1,1)=(1,0)$) whose iterations graphs are represented in Fig. 1. The two graphs are strongly connected and thus the vectorial negation function should theoretically be replaced by the function f.

In the graph of iterations $\Gamma(\neg)$ (Fig. 1a), let us compute the probability $P^t_{\neg}(X)$ to reach the node X in t iterations from the node 00. Let X_0, X_1, X_2, X_3 be the nodes 00, 01, 10 and 11. For $i\in[\![0,3]\!]$, $P^1_{\neg}(X_i)$, are respectively equal to 0.0, 0.5, 0.0, 0.5. In two iterations $P^2_{\neg}(X_i)$ are 0.5, 0.0, 0.5, 0.0. It is obvious to establish that we have $P^{2t}(X_i)=P^0(X_i)$



(a) Negation (b) $(1, 3, 0, 2)$

Figure 1: Graphs of Iterations

Name	Deviation	Suff. number of it.
ⓐ	8.1%	167
ⓑ	1%	105
ⓒ	18%	58
ⓓ	1%	22
ⓔ	24%	19
ⓕ	1%	14
ⓖ	20%	6
ⓗ	45.3%	7
ⓘ	53.2%	14

Table III: Deviation with Uniform Distribution

and $P^{2t+1}(X_i) = P^1(X_i)$ for any $t \in \mathbb{N}$. Then in k or $k+1$ iterations all these probabilities are equal to 0.25.

Let us apply a similar reasoning for the function f defined by $[1, 3, 0, 2]$. In its iterations graph $\Gamma(f)$ (Fig. 1b), and with X_i defined as above, the probabilities $P_f^1(X_i)$ to reach the node X_i in one iteration from the node 00 are respectively equal to 0.5, 0.5, 0.0, 0.0. Next, probabilities $P_f^2(X)$ are 0.25, 0.5, 0.25, 0.0. Next, $P_f^3(X)$ are 0.125, 0.375, 0.375, 0.125. For each iteration, we compute the average deviation rate R^t with 0.25 as follows.

$$R^t = \frac{\Sigma_{i=0}^{3} \mid P_f^t(X_i) - 0.25 \mid}{4}.$$

The higher is this rate, the less the generator may uniformly reach any X_i from 00. For this example, it is necessary to iterate 14 times in order to observe a deviation from 0.25 less than 1%. A similar reasoning has been applied for all the functions listed in Table II. The table III summarizes their deviations with uniform distribution and gives the smallest iterations number the smallest deviation has been obtained.

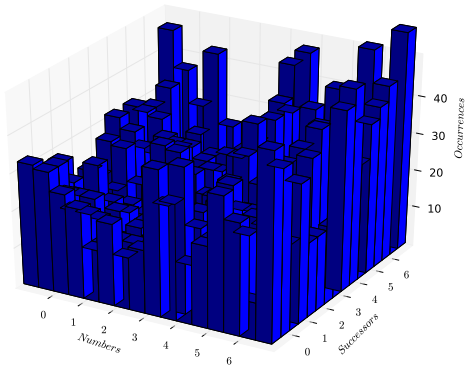
With that material we present in Algorithm 4 the method that allows to take any chaotic function as the core of a pseudo random number generator. Among the parameters, it takes the number b of minimal iterations that have to be executed to get a uniform like distribution. For our experiments b is set with the value given in the third column of Table III.

Compared to the algorithm 2 parameters of this one are the function f to embed and the smallest number of time steps G_f is iterated. First, the number of iterations is either b or $b+1$ depending on the value of the *XORshift* output (if the next value . Next, a loop that iterates G_f is executed.

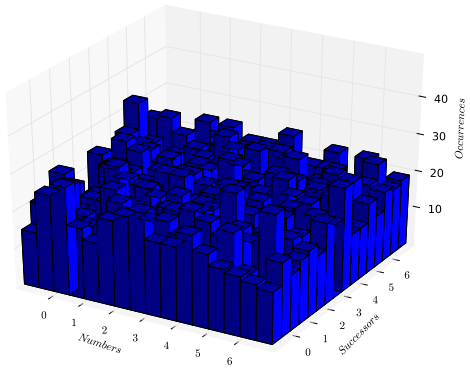
In this example, n and b are equal to 4 for easy understanding. The initial state of the system x^0 can be seeded by the decimal part of the current time. For example, the current time in seconds since the Epoch is 1237632934.484088, so $t = 484088$. $x^0 = t \mod 16$ in binary digits, then $x^0 = 0100$. m and S can now be computed from *XORshift*.

Input: a function f, an iteration number b, an initial state x^0 (n bits)
Output: a state x (n bits)
$x \leftarrow x^0$;
$k \leftarrow b + (\textit{XORshift}() \mod 2)$;
for $i = 0, \ldots, k-1$ **do**
  $s \leftarrow \textit{XORshift}() \mod n$;
  $x \leftarrow F_f(s, x)$;
end
return x;

Algorithm 4: modified PRNG with various functions



(a) Function ⓔ



(b) Function ⓕ

Figure 2: Repartition of function outputs.

- f = [14,15,13,12,11,10,9,8,7,6,5,4,3,2,1,0]
- k = 4, 5, 4,...
- s = 2, 4, 2, 3, , 4, 1, 1, 4, 2, , 0, 2, 3, 1,...

Chaotic iterations are done with initial state x^0, the mapping function f, and strategy s^1, s^2... The result is presented in Table IV. Let us recall that sequence k gives the states x^t to return: x^4, x^{4+5}, x^{4+5+4}... Successive stages are detailed in Table IV.

To illustrate the deviation, Figures 2a and 2b represent the simulation outputs of 5120 executions with b equal to 40 for ⓔ and ⓕ respectively. In these two figures, the point (x, y, z) can be understood as follows. z is the number of times the value x has been succedded by the value y in the considered generator. These two figures explicitly confirm that outputs of functions ⓕ are more uniform that these of the function ⓔ. In the former each number x reaches about 20 times each number y whereas in the latter, results vary from 10 to more that 50.

 ISBN: 978-1-61208-141-0

k	4					5						4				
s	2	4	2	3		4	1	1	4	2		0	2	3	1	
	$f(4)$	$f(0)$	$f(0)$	$f(4)$		$f(6)$	$f(7)$	$f(15)$	$f(7)$	$f(7)$		$f(2)$	$f(0)$	$f(4)$	$f(6)$	
	1	1	1	1		1	**1**	**0**	1	1		1	1	1	**1**	
f	**0**	1	**1**	0		0	0	0	0	**0**		1	**1**	0	0	
	1	1	1	**1**		0	0	0	0	0		**0**	1	**1**	0	
	1	**0**	0	1		**1**	0	0	**0**	0		1	0	1	1	
x^0					x^4						x^9					x^{13}
4	0	0	4	6	6	7	15	7	7	7	2	0	4	6	14	14
0					0		$\xrightarrow{1}$ 1	$\xrightarrow{1}$ 0			0				$\xrightarrow{1}$ 1	1
1	$\xrightarrow{2}$ 0		$\xrightarrow{2}$ 1		1					$\xrightarrow{2}$ 0	0		$\xrightarrow{2}$ 1			1
0				$\xrightarrow{3}$ 1	1						1	$\xrightarrow{3}$ 0		$\xrightarrow{3}$ 1		1
0		$\xrightarrow{4}$ 0			0	$\xrightarrow{4}$ 1			$\xrightarrow{4}$ 0		0					0

Table IV: Application example

VI. Experiments

A convincing way to prove the quality of the produced sequences is to confront them with the NIST (National Institute of Standards and Technology) Statistical Test Suite SP 800-22 [13]. This is a statistical package consisting of 15 tests that focus on a variety of different types of non-randomness that could occur in a (arbitrarily long) binary sequences produced by a pseudo-random number generators.

For all 15 tests, the significance level α was set to 1%. If a p-value is greater than 0.01, the keystream is accepted as random with a confidence of 99%; otherwise, it is considered as non-random. For each statistical test, a set of p-values is produced from a set of sequences obtained by our generator (i.e., 100 sequences are generated and tested, hence 100 p-values are produced).

Empirical results can be interpreted in various ways. In this paper, we check whether $\mathbb{P}_T$ (P-values of p-values), which arise via the application of a chi-square test, were all higher than 0.0001. This means that all p-values are uniformly distributed over (0, 1) interval as expected for an ideal random number generator.

Table V shows $\mathbb{P}_T$ of the sequences based on discrete chaotic iterations using different "iteration" functions. If there are at least two statistical values in a test, the test is marked with an asterisk and the average value is computed to characterize the statistical values. Here, NaN means a warning that test is not applicable because of an insufficient number of cycles. Time (in seconds) is related to the duration needed by each algorithm to generate a 10^8 bits long sequence. The test has been conducted using the same computer and compiler with the same optimization settings for both algorithms, in order to make the test as fair as possible.

Firstly, the computational time in seconds has increased due to the growth of the sufficient iteration numbers, as precised in Table III. For instance, the fastest generator is ⓖ since each new number generation only requires 6 iterations. Next, concerning the NIST tests results, best situations are given by ⓑ, ⓓ and ⓕ. In the opposite, it can be observed that among the 15 tests, less than 5 ones are a successful for other functions. Thus, we can draw a conclusion that, ⓑ, ⓓ, and ⓕ are qualified to be good PRNGs with chaotic property. NIST tests results are not a surprise: ⓑ, ⓓ, and ⓕ have indeed a deviation less than 1% with the uniform distribution as already precised in Table III. The rate of removed edge in the graph $\Gamma(\neg)$ is then not a pertinent criteria compared to the deviation with the uniform distribution property: the function ⓐ whose graph $\Gamma(ⓐ)$ is $\Gamma(\neg)$ without the edge $1010 \to 1000$ (*i.e.*, with only one edge less than $\Gamma(\neg)$) has dramatic results compared to the function ⓕ with many edges less.

Let us then try to give a characterization of convenient function. Thanks to a comparison with the other functions, we notice that ⓑ, ⓓ, and ⓕ are composed of all the elements of $[\![0; 15]\!]$. It means that ⓑ, ⓓ, and ⓕ, and even the vectorial boolean negation function are arrangements of $[\![0; 2^n]\!]$ ($n = 4$ in this article) into a particular order.

VII. Conclusion

In this work, we first have formalized the PRNG already presented in a previous work. It results a new presentation that has allowed to optimize some part and thus has led to a more efficient algorithm. But more fundamentally, this PRNG closely follows iterations that have been proven to be topological chaotic.

By considering a characterization of functions with topological chaotic behavior (namely those with a strongly connected graph of iterations), we have computed a new class of PRNG based on instances of such functions. These functions have been randomly generated starting from the negation function. Then an a posteriori analysis has checked whether any number may be equiprobabilistically reached from any other one.

The NIST statistical test has confirmed that functions without equiprobabilistical behavior are not good candidates for being iterated in our PRNG. In the opposite, the other ones have topological chaos property and success all the NIST tests. To summarize the approach, all our previous approaches were based on only one function (namely the negation function) whereas we provide now a class of many trustworthy PRNG.

Future work are mainly twofold. We will firstly study sufficient conditions to obtain functions with the two properties of equiprobability and strongly connectivity of its graph of iterations. With such a condition any user should choose its own trustworthy PRNG. Dually, we will continue the evaluation of randomness quality by checking other statistical series like DieHard[11], TestU01 [9]...on newly generated

 ISBN: 978-1-61208-141-0

Method	ⓐ	ⓑ	ⓒ	ⓓ	ⓔ	ⓕ	ⓖ	ⓗ	ⓘ
Frequency (Monobit) Test	0.00000	0.45593	0.00000	0.38382	0.00000	0.61630	0.00000	0.00000	0.00000
Frequency Test within a Block	0.00000	0.55442	0.00000	0.03517	0.00000	0.73991	0.00000	0.00000	0.00000
Cumulative Sums (Cusum) Test*	0.00000	0.56521	0.00000	0.19992	0.00000	0.70923	0.00000	0.00000	0.00000
Runs Test	0.00000	0.59554	0.00000	0.14532	0.00000	0.24928	0.00000	0.00000	0.00000
Test for the Longest Run of Ones in a Block	0.20226	0.17186	0.00000	0.38382	0.00000	0.40119	0.00000	0.00000	0.00000
Binary Matrix Rank Test	0.63711	0.69931	0.05194	0.16260	0.79813	0.03292	0.85138	0.12962	0.07571
Discrete Fourier Transform (Spectral) Test	0.00009	0.09657	0.00000	0.93571	0.00000	0.93571	0.00000	0.00000	0.00000
Non-overlapping Template Matching Test*	0.12009	0.52365	0.05426	0.50382	0.02628	0.50326	0.06479	0.00854	0.00927
Overlapping Template Matching Test	0.00000	0.73991	0.00000	0.55442	0.00000	0.45593	0.00000	0.00000	0.00000
Maurer's "Universal Statistical" Test	0.00000	0.71974	0.00000	0.77918	0.00000	0.47498	0.00000	0.00000	0.00000
Approximate Entropy Test	0.00000	0.10252	0.00000	0.28966	0.00000	0.14532	0.00000	0.00000	0.00000
Random Excursions Test*	NaN	0.58707	NaN	0.41184	NaN	0.25174	NaN	NaN	NaN
Random Excursions Variant Test*	NaN	0.32978	NaN	0.57832	NaN	0.31028	NaN	NaN	NaN
Serial Test* (m=10)	0.11840	0.95107	0.01347	0.57271	0.00000	0.82837	0.00000	0.00000	0.00000
Linear Complexity Test	0.91141	0.43727	0.59554	0.43727	0.55442	0.43727	0.59554	0.69931	0.08558
Success	5/15	15/15	4/15	15/15	3/15	15/15	3/15	3/15	3/15
Computational time	66.0507	47.0466	32.6808	21.6940	20.5759	19.2052	16.4945	16.8846	19.0256

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

functions.

REFERENCES

[1] J. M. Bahi and C. Guyeux. Topological chaos and chaotic iterations, application to hash functions. In *WCCI'10, IEEE World Congress on Computational Intelligence*, pages 1–7, Barcelona, Spain, July 2010. Best paper award.

[2] J. M. Bahi, C. Guyeux, and Q. Wang. A novel pseudo-random generator based on discrete chaotic iterations. In *INTERNET'09, 1-st Int. Conf. on Evolving Internet*, pages 71–76, Cannes, France, August 2009.

[3] J. M. Bahi, C. Guyeux, and Q. 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.

[4] J. M. Bahi, C. Guyeux, and Q. 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.

[5] S. Behnia, A. Akhavan, A. Akhshani, and A. Samsudin. A novel dynamic model of pseudo random number generator. *Journal of Computational and Applied Mathematics*, 235(12):3455–3463, 2011.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 3rd ed. edition, 2009.

[7] R. L. Devaney. *An Introduction to Chaotic Dynamical Systems*. Redwood City: Addison-Wesley, 2nd edition, 1989.

[8] C. Guyeux. *Le désordre des itérations chaotiques et leur utilité en sécurité informatique*. PhD thesis, Université de Franche-Comté, 2010.

[9] P. L'Ecuyer and R. J. Simard. Testu01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), 2007.

[10] N. Liu. Pseudo-randomness and complexity of binary sequences generated by the chaotic system. *Communications in Nonlinear Science and Numerical Simulation*, 16(2):761–768, 2011.

[11] G. Marsaglia. Diehard: a battery of tests of randomness. *1414203*, 1996.

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

[13] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute of Standards and Technology, April 2010.

[14] Fuyan Sun and Shutang Liu. Cryptographic pseudo-random sequence from the spatial chaotic map. *Chaos, Solitons & Fractals*, 41(5):2216–2219, 2009.

[15] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

 ISBN: 978-1-61208-141-0