An optimization technique on pseudorandom generators based on chaotic iterations

Jacques M. Bahi, Xiaole Fang, and Christophe Guyeux*
FEMTO-ST Institute, UMR 6174 CNRS
University of Franche-Comté, Besançon, France
Email:{jacques.bahi, xiaole.fang, christophe.guyeux}@univ-fcomte.fr

***Abstract*—Internet communication systems involving cryptography and data hiding often require billions of random numbers. In addition to the speed of the algorithm, the quality of the pseudo-random number generator and the ease of its implementation are common practical aspects. In this work we will discuss how to improve the quality of random numbers independently from their generation algorithm. We propose an additional implementation technique in order to take advantage of some chaotic properties. The statistical quality of our solution stems from some well-defined discrete chaotic iterations that satisfy the reputed Devaney's definition of chaos, namely the chaotic iterations technique. Pursuing recent researches published in the previous International Conference on Evolving Internet (Internet 09, 10, and 11), three methods to build pseudorandom generators by using chaotic iterations are recalled. Using standard criteria named NIST and DieHARD (some famous batteries of tests), we will show that the proposed technique can improve the statistical properties of a large variety of defective pseudorandom generators, and that the issues raised by statistical tests decrease when the power of chaotic iterations increase.**

***Keywords*-Internet security; Pseudorandom Sequences; Statistical Tests; Discrete Chaotic Iterations; Topological Chaos.**

I. INTRODUCTION

Chaos has recently attracted more and more interests from researchers in the fields of mathematics, physics, and computer engineering, among other things due to its connection with randomness and complexity [9], [7]. In particular, various research works have recently regarded the possibility to use chaos in random number generation for Internet security. Indeed, the security of data exchanged through the Internet is highly dependent from the quality of the pseudorandom number generators (PRNGs) used into its protocols. These PRNGs are everywhere in any secure Internet communication: in the keys generation of any asymmetric cryptosystem, in the production of any keystream (symmetric cryptosystem), the generation of nonce, in the keys for keyed hash functions, and so on.

Numerous pseudorandom number generators already exist, but they are either secure but slow, or fast but insecure. This is why the idea to mix secure and fast PRNGs, to take benefits from their respective qualities, has emerged these last years [7], [1]. Chaotic dynamical systems appear as good candidates to achieve this mixture for optimization. Indeed, chaotic systems have many advantages as unpredictability or disorder-like, which are required in building complex sequences [12], [16]. This is why chaos has been applied to secure optical communications [13]. But chaotic systems of real-number or infinite bit representation realized in finite computing precision lead to short cycle length, non-ideal distribution, and other deflation of this kind. This is the reason of that chaotic systems on an infinite space of integers have been looked for these last years, leading to the proposition to use chaotic iterations (CIs) techniques to reach the desired goals. More precisely, we have proposed in INTERNET 2009 [4] to mix two given PRNGs by using chaotic iterations, being some particular kind of discrete iterations of a vectorial Boolean function. This first proposal has been improved in INTERNET 2010 [20] and INTERNET 2011 [3], to obtain a new family of statistically perfect and fast PRNGs. A short overview of these previous researches is given thereafter.

In [7], CIs have been proven to be a suitable tool for fast computing iterative algorithms on integers satisfying the topological chaotic property, as it has been defined by Devaney [10]. A first way to mix two given generators by using these chaotic iterations, called Old CIPRNGs, has been proposed in Internet 09 [4] and further investigated in [5], [2], [8]. It was chaotic and able to pass the most stringent batteries of tests, even if the inputted generators were defective. This claim has been verified experimentally, by evaluating the scores of the logistic map, XORshift, and ISAAC generators through these batteries, when considering them alone or after chaotic iterations. Then, in [20], a new version of this family has been proposed. This "New CIPRNG" family uses a decimation of strategies leading to the improvement of both speed and statistical qualities. Finally, efficient implementations on GPU using a last family called Xor CIPRNG have been designed in [6], showing that a very large quantity of pseudorandom numbers can be generated per second (about 20 Gsamples/s).

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

The remainder of this paper is organized in the following way. In Section II, some basic definitions concerning chaotic iterations are recalled. Then, four major classes of general PRNGs are presented in Section III. Section IV is devoted to two famous statistical tests suites. In Section V, various tests are passed with a goal to achieve a statistical comparison between our CIPRNGs and other existing generators. The paper ends with a conclusion and intended future work.

 ISBN: 978-1-61208-204-2

II. Chaotic Iterations Applied to PRNGs

In this section, we describe the CIPRNG implementation techniques that can improve the statistical properties of any generator. They all are based on CIs, which are defined below.

A. Notations

S^n → the n^{th} term of a sequence $S = (S^1, S^2, \ldots)$
v_i → the i^{th} component of a vector $v = (v_1, \ldots, v_n)$
f^k → k^{th} composition of a function f
strategy → a sequence which elements belong in $[\![1; \mathsf{N}]\!]$
$\mathbb{S}$ → the set of all strategies
$\mathbf{C}_n^k$ → the binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
$\oplus$ → bitwise exclusive or
$\ll$ and $\gg$ → the usual shift operators
$(\mathcal{X}, \mathrm{d})$ → a metric space
$LCM(a, b)$→ the least common multiple of a and b

B. Chaotic iterations

Definition 1 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 $x^0 \in \mathbb{B}^{\mathsf{N}}$, and

$$\forall n \in \mathbb{N}^*, \forall i \in [\![1; \mathsf{N}]\!], x_i^n = \begin{cases} x_i^{n-1} & \text{if } S^n \neq i \\ f(x^{n-1})_{S^n} & \text{if } S^n = i. \end{cases} \quad (1)$$

In other words, at the n^{th} iteration, only the S^n-th cell is "iterated".

C. The CIPRNG family

1) Old CIPRNG: Let $\mathsf{N} = 4$. Some chaotic iterations are fulfilled to generate a sequence $(x^n)_{n\in\mathbb{N}} \in \left(\mathbb{B}^4\right)^{\mathbb{N}}$ 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 [4]. Chaotic iterations are realized as follows. Initial state $x^0 \in \mathbb{B}^4$ is a Boolean vector taken as a seed and chaotic strategy $(S^n)_{n\in\mathbb{N}} \in [\![1, 4]\!]^{\mathbb{N}}$ is constructed with $PRNG_2$. Lastly, iterate function f is the vectorial Boolean negation. At each iteration, only the S^n-th component of state x^n is updated. Finally, some x^n are selected by a sequence m^n, provided by a second generator $PRNG_1$, as the pseudorandom bit sequence of our generator.

The basic design procedure of the Old CI generator is summed up in Algorithm 1. The internal state is x, the output array is r. a and b are those computed by $PRNG_1$ and $PRNG_2$.

Input: the internal state x (an array of 4-bit words)
Output: an array r of 4-bit words

1: $a \leftarrow PRNG_1()$;
2: $m \leftarrow a \ mod \ 2 + 13$;
3: **while** $i = 0, \ldots, m$ **do**
4: $\quad b \leftarrow PRNG_2()$;
5: $\quad S \leftarrow b \ mod \ 4$;
6: $\quad x_S \leftarrow \overline{x_S}$;
7: **end while**
8: $r \leftarrow x$;
9: return r;

Algorithm 1: An arbitrary round of the old CI generator

2) New CIPRNG: The New CI generator is designed by the following process [11]. First of all, some chaotic iterations have to be done to generate a sequence $(x^n)_{n\in\mathbb{N}} \in \left(\mathbb{B}^{32}\right)^{\mathbb{N}}$ 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 $x^0 \in \mathbb{B}^{32}$ is a Boolean vector taken as a seed and chaotic strategy $(S^n)_{n\in\mathbb{N}} \in [\![1, 32]\!]^{\mathbb{N}}$ is an *irregular decimation* of $PRNG_2$ sequence, as described in Algorithm 2.

Another time, at each iteration, only the S^n-th component of state x^n is updated, as follows: $x_i^n = x_i^{n-1}$ if $i \neq S^n$, 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 $PRNG_1$, where $\mathcal{M} \subset \mathbb{N}^*$ is a finite nonempty set of integers.

The basic design procedure of the New CI generator is summarized in Algorithm 2. The internal state is x, the output state is r. a and b are those computed by the two input PRNGs. Lastly, the value $g_1(a)$ is an integer defined as in Eq. 2.

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

Input: the internal state x (32 bits)
Output: a state r of 32 bits

1: **for** $i = 0, \ldots, N$ **do**
2: $\quad d_i \leftarrow 0$;
3: **end for**
4: $a \leftarrow PRNG_1()$;
5: $m \leftarrow f(a)$;
6: $k \leftarrow m$;
7: **while** $i = 0, \ldots, k$ **do**
8: $\quad b \leftarrow PRNG_2() \ mod \ \mathsf{N}$;
9: $\quad S \leftarrow b$;
10: $\quad$ **if** $d_S = 0$ **then**
11: $\quad\quad x_S \leftarrow \overline{x_S}$;
12: $\quad\quad d_S \leftarrow 1$;
13: $\quad$ **else if** $d_S = 1$ **then**
14: $\quad\quad k \leftarrow k + 1$;
15: $\quad$ **end if**
16: **end while**
17: $r \leftarrow x$;
18: return r;

Algorithm 2: An arbitrary round of the new CI generator

3) Xor CIPRNG: Instead of updating only one cell at each iteration as Old CI and New CI, we can try to choose a subset of components and to update them together. Such an attempt leads to a kind of merger of the two random sequences. When the updating function is the vectorial negation, this algorithm can be rewritten as follows [6]:

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus S^n, \end{cases} \quad (3)$$

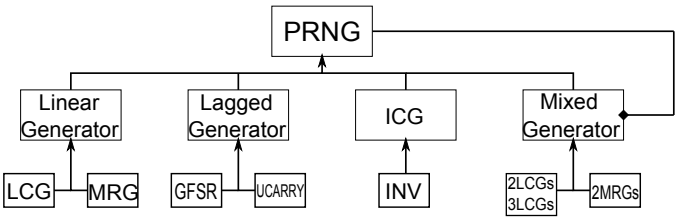

Figure 1: Ontological class hierarchy of PRNGs

The single basic component presented in Eq. 15 is of ordinary use as a good elementary brick in various PRNGs. It corresponds to the discrete dynamical system in chaotic iterations.

III. About some Well-known PRNGs

A. Introduction

Knowing that there is no universal generator, it is strongly recommended to test a stochastic application with a large set of different PRNGs [17]. They can be classified in four major classes: linear generators, lagged generators, inversive generators, and mix generators:

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

For instance, inversive generators are very interesting for verifying simulation results obtained with a linear congruential generator (LCG), because their internal structure and correlation behavior strongly differs from what LCGs produce. Since these generators have revealed several issues, some scientists refrain from using them. In what follows, chaotic properties will be added to these PRNGs, leading to noticeable improvements observed by statistical test. Let us firstly explain with more details the generators studied in this research work (for a synthetic view, see Fig. 1).

B. Details of some Existing Generators

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

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

$$x^n = (ax^{n-1} + c) \ mod \ m \tag{4}$$

where a, c, and x^0 must be, among other things, non-negative and less than m [19]. In what follows, 2LCGs and 3LCGs refer as two (resp. three) combinations of such LCGs. For further details, see [14].

2) MRG: This module implements multiple recursive generators (MRGs), based on a linear recurrence of order k, modulo m [19]:

$$x^n = (a^1 x^{n-1} + \ ... \ + a^k x^{n-k}) \ mod \ m \tag{5}$$

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

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

$$\begin{aligned} x^n &= (x^{n-r} + x^{n-s} + c^{n-1}) \ mod \ m, \\ c^n &= (x^{n-r} + x^{n-s} + c^{n-1})/m, \end{aligned} \tag{6}$$

the SWB generator, having the recurrence:

$$\begin{aligned} x^n &= (x^{n-r} - x^{n-s} - c^{n-1}) \ mod \ m, \\ c^n &= \begin{cases} 1 & \text{if } (x^{i-r} - x^{i-s} - c^{i-1}) < 0 \\ 0 & \text{else,} \end{cases} \end{aligned} \tag{7}$$

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

$$\begin{aligned} x^n &= (a^1 x^{n-1} \oplus \ ... \ \oplus a^r x^{n-r} \oplus c^{n-1}) \ mod \ 2^w, \\ c^n &= (a^1 x^{n-1} \oplus \ ... \ \oplus a^r x^{n-r} \oplus c^{n-1}) \ / \ 2^w. \end{aligned} \tag{8}$$

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

$$x^n = x^{n-r} \oplus x^{n-k} \tag{9}$$

5) INV: Finally, this module implements the nonlinear inversive generator, as defined in [19], which is:

$$x^n = \begin{cases} (a^1 + a^2/z^{n-1}) \ mod \ m & \text{if } z^{n-1} \neq 0 \\ a^1 & \text{if } z^{n-1} = 0. \end{cases} \tag{10}$$

IV. Statistical tests

Considering the properties of binary random sequences, various statistical tests can be designed to evaluate the assertion that the sequence is generated by a perfectly random source. We have performed some statistical tests for the CIPRNGs proposed here. These tests include NIST suite [18] and DieHARD battery of tests [15]. For completeness and for reference, we give in the following subsection a brief description of each of the aforementioned tests.

A. NIST statistical tests suite

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

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

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

 ISBN: 978-1-61208-204-2

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

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

B. DieHARD battery of tests

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

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

V. Results and discussion

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

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

1) **Single CIPRNG**: The PRNGs involved in CI computing are of the same category.
2) **Mixed CIPRNG**: Two different types of PRNGs are mixed during the chaotic iterations process.
3) **Multiple CIPRNG**: The generator is obtained by repeating the composition of the iteration function as follows: $x^0 \in \mathbb{B}^{\mathsf{N}}$, and $\forall n \in \mathbb{N}^*, \forall i \in [\![1; \mathsf{N}]\!]$,

$$x_i^n = \begin{cases} x_i^{n-1} & \text{if } S^n \neq i \\ \forall j \in [\![1; \mathsf{m}]\!], f^m(x^{n-1})_{S^{nm+j}} & \text{if } S^{nm+j} = i. \end{cases} \quad (11)$$

m is called the *functional power*.

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

A. Tests based on the Single CIPRNG

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

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

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

B. Tests based on the Mixed CIPRNG

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

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus PRNG_1 \oplus PRNG_2, \end{cases} \quad (12)$$

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

$$n_{SXORCI} = \begin{cases} n_P & \text{if } x^0 = x^{n_P} \\ 2n_P & \text{if } x^0 \neq x^{n_P}. \end{cases} \quad (13)$$

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

$$n_{XXORCI} = \begin{cases} LCM(n_{P1}, n_{P2}) & \text{if } x^0 = x^{LCM(n_{P1}, n_{P2})} \\ 2LCM(n_{P1}, n_{P2}) & \text{if } x^0 \neq x^{LCM(n_{P1}, n_{P2})}. \end{cases} \quad (14)$$

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

- **Matrix Rank 32x32.** A random 32x32 binary matrix is formed, each row having a 32-bit random vector. Its rank is an integer that ranges from 0 to 32. Ranks less than 29 must be rare, and their occurences must be pooled with those of rank 29. To achieve the test, ranks of 40,000 such random matrices are obtained, and a chisquare test is performed on counts for ranks 32,31,30 and for ranks ≤ 29.
- **COUNT-THE-1's TEST** Consider the file under test as a stream of bytes (four per 2 bit integer). Each byte can contain from 0 to 8 1's, with probabilities

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

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

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

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

1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte: 0,1, or 2 yield A, 3 yields B, 4 yields C, 5 yields D and 6,7, or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (over-lapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test: Q5-Q4, the difference of the naive Pearson sums of $(OBS - EXP)^2/EXP$ on counts for 5- and 4-letter cell counts.

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

C. Tests based on the Multiple CIPRNG

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

$$\begin{cases} x^0 \in [\![0, 2^{\mathsf{N}} - 1]\!], S \in [\![0, 2^{\mathsf{N}} - 1]\!]^{\mathbb{N}} \\ \forall n \in \mathbb{N}^*, x^n = x^{n-1} \oplus S^{nm} \oplus S^{nm+1} \ldots \oplus S^{nm+m-1}, \end{cases} \quad (15)$$

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

D. Results Summary

We can summarize the obtained results as follows.

1) The CIPRNG method is able to improve the statistical properties of a large variety of PRNGs.
2) Using different PRNGs in the CIPRNG approach is better than considering several instances of one unique PRNG.
3) The statistical quality of the outputs increases with the functional power m.

VI. Conclusion and Future Work

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

In future work, we will try to enlarge this study, by considering a larger variety of tests. The CIPRNG's chaotic behavior will be deepened by using some specific tools provided by the mathematical theory of chaos. Finally, a large variety of Internet usages, as cryptography and data hiding, will be considered for applications.

References

[1] J. M. Bahi and C. Guyeux. A new chaos-based watermarking algorithm. In *SECRYPT 2010, International conference on security and cryptography*, pages 1–4, Athens, Greece, 2010. To appear.

 ISBN: 978-1-61208-204-2

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

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

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

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

[2] J. M. Bahi and C. Guyeux. Topological chaos and chaotic iterations, application to hash functions. *WCCI'10: 2010 IEEE World Congress on Computational Intelligence*, Accepted paper, 2010.

[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 72–77, Luxembourg, Luxembourg, June 2011. To appear.

[4] Jacques Bahi, Christophe Guyeux, and Qianxue 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.

[5] 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.

[6] Jacques M. Bahi, Raphael Couturier, Christophe Guyeux, and Pierre-Cyrille Heam. Efficient and cryptographically secure generation of chaotic pseudorandom numbers on gpu. *http://arxiv.org/abs/1112.5239*, December 2011. Submitted already.

[7] Jacques M. Bahi and Christophe 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.

[8] Jacques M. 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.

[9] 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.

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

[11] Christophe Guyeux and Jacques Bahi. An improved watermarking algorithm for internet applications. In *INTERNET'2010. The 2nd Int. Conf. on Evolving Internet*, pages 119–124, Valencia, Spain, September 2010.

[12] Yue Hu, Xiaofeng Liao, Kwok wo Wong, and Qing Zhou. A true random number generator based on mouse movement and chaotic cryptography. *Chaos, Solitons & Fractals*, 40(5):2286–2293, 2009.

[13] L. Larger and J.M. Dudley. Nonlinear dynamics Optoelectronic chaos. *Nature*, 465(7294):41–42, 05 2010.

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

[15] George Marsaglia. Diehard: a battery of tests of randomness. *http://www.stat.fsu.edu/pub/diehard/*, 1996.

[16] L. De Micco, C.M. Gonzaez, H.A. Larrondo, M.T. Martin, A. Plastino, and O.A. Rosso. Randomizing nonlinear maps via symbolic dynamics. *Physica A: Statistical Mechanics and its Applications*, 387(14):3373–3383, 2008.

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

[18] NIST Special Publication 800-22 rev. 1. A statistical test suite for random and pseudorandom number generators for cryptographic applications. August 2008.

[19] Richard Simard and Université De Montréal. Testu01: A software library in ansi c for empirical testing of random number generators. software users guide. *ACM Transactions on Mathematical Software*, 2002.

[20] Qianxue Wang, Jacques Bahi, Christophe Guyeux, and Xiaole Fang. Randomness quality of CI chaotic generators. application to internet security. In *INTERNET'2010. The 2nd Int. Conf. on Evolving Internet*, pages 125–130, Valencia, Spain, September 2010. IEEE Computer Society Press. Best Paper award.

 ISBN: 978-1-61208-204-2