FPGA acceleration of a pseudorandom number generator based on chaotic iterations

Xiaole Fang a,*, Qianxue Wang b, Christophe Guyeux c, Jacques M. Bahi c

a Land and Resources Technology Center of GuangDong Province, GuangZhou, 510075, China
b College of Automation, GuangDong University of Technology, GuangZhou, 510006, China
c FEMTO-ST Institute, UMR CNRS 6174, University of Franche-Comté, 25000, France

Abstract
As any well-designed information security application uses a very large quantity of good pseudorandom numbers, inefficient generation of these numbers can be a significant bottleneck in various situations. In previous research works, a technique that applies well-defined discrete iterations, satisfying the reputed Devaney’s definition of chaos, has been developed. It has been proven that the generators embedding these chaotic iterations (CIs) produce truly chaotic random numbers. In this new article, these generators based on chaotic iterations are redesigned specifically for Field Programmable Gate Array (FPGA) hardware, leading to an obvious improvement of the generation rate. Analyses illustrate that statistically perfect and chaotic random sequences are produced. Additionally, such generators can also be cryptographically secure. To show the effectiveness of the method, an application in the information hiding domain is finally proposed.

Keywords:
Internet security
Pseudorandom number generator
Discrete chaotic iterations
Cryptographical security
FPGA

1. Introduction

Pseudorandom number generators (PRNGs) are very important primitives widely used in numerous applications like numerical simulations or security. For instance, they are one of the most fundamental components that any cryptosystem has to embed, in order to generate encryption keys or keystreams in symmetric ciphers. Depending on the targeted application, these PRNGs must achieve requirements as speed, statistical quality, security, and so on. On the one hand, field programmable gate arrays (FPGAs) have been successfully used for realizing the speed requirement in pseudorandom sequence generation, due to their high parallelization capability (Bojanic et al., 2006; Danger et al., 2009; Tsoi et al., 2003). Advantages of such physical generation way encompass performance, design time, power consumption, flexibility, and cost. On the other hand, some researches have recently demonstrated the interest to use chaotic dynamical systems as PRNGs, among other things due to the unpredictability and distorted-like properties of such systems (Falciouli et al., 2005; Cecen et al., 2009, and Lee et al., 2004)). Their sensitivity to initial conditions and their broadband spectrum make them good candidates to generate sequences both secure and random. For instance, such chaos-based generators have been successfully used to strengthen optical communications (Larger and Dudley, 2010).

In this paper, which continues with the studies initiated in Bahi et al. (2013a), Bahi et al. (2009), Bahi et al. (2011), Bahi and Guyeux (2010a), and Bahi et al. (2012a), we intend to merge...
these two approaches by proposing a discrete chaos-based generator designed on FPGA. A short overview of our previous research is given thereafter. It has firstly been stated that a tool called chaotic iterations (CIs), used in distributed computing, satisfies the chaotic property as it is defined by Devaney (Devaney, 2003). The chaotic behavior of CIs has secondly been harnessed to obtain a class of unpredictable PRNGs (Bahi et al., 2009). This class receives two given generators as input and mix them with chaotic iterations, producing by doing so a sequence having a better random profile than the two inputs. Thirdly, in Bahi et al. (2011), two new versions of such “CIPRNGs” (chaotic iterations based pseudo-random number generators) have been proposed, involving respectively two logistic maps and two XORshifts (Marsaglia, 2003). Some efficient implementations on GPU, using another improvement of these generators, have been designed in Couturier and Guyeux (2013), leading to a very large quantity of pseudorandom numbers generated per second. Finally, in Bahi et al. (2012a), this family of CIPRNGs has been used with 10 well-known defective PRNGs, in order to show that they do not only inherit the chaotic properties of CIs, but also improve a lot the statistics of the inputted generators.

In this new research work, the proposition is to improve largely the efficiency of our formerly proposed generators, without any lack of chaos properties. To do so, another improved version of CIPRNG on Field Programmable Gate Array is proposed, and its cryptographical security is established in some cases. Furthermore, the most famous and important batteries of tests for evaluating PRNGs are applied on some generated numbers, to prove the randomness quality of this new proposal. Last, but not least, a potential use of this generator in cryptography is presented.

The remainder of this research work is organized as follows. Basic definitions concerning chaotic iterations and PRNGs are recalled in Section 2. The proposed design of our new generator based on CIs, its proof of cryptographical security, and its randomness quality are given in Section 3. The next section aims to describe how to use corresponding tools in order to design the generator in FPGA. Then, in Section 5, a potential use of this generator for digital information hiding is presented. The paper ends by a conclusion section where the proposal is summarized and intended future work is outlined.

2. Definitions and terminologies

2.1. Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1:N]</td>
<td>{1,2,...,N}</td>
</tr>
<tr>
<td>(S^n)</td>
<td>the (n^{th}) term of a sequence (S=(S^1,S^2,...))</td>
</tr>
<tr>
<td>(v_i)</td>
<td>the (i^{th}) component of a vector: (v=(v_1,v_2,...,v_n))</td>
</tr>
<tr>
<td>(f^k)</td>
<td>(k^{th}) composition of a function (f)</td>
</tr>
<tr>
<td>strategy</td>
<td>a sequence which elements belong in ([1:N])</td>
</tr>
<tr>
<td>(S)</td>
<td>the set of all strategies</td>
</tr>
<tr>
<td>(X^n)</td>
<td>the set of sequences belonging into (X)</td>
</tr>
<tr>
<td>(C_n^k)</td>
<td>the binomial coefficient (\frac{n!}{k!(n-k)!})</td>
</tr>
<tr>
<td>(\Box)</td>
<td>the bitwise exclusive or</td>
</tr>
<tr>
<td>(+)</td>
<td>the integer addition</td>
</tr>
<tr>
<td>(&lt;\text{and}&gt;)</td>
<td>the usual shift operators ((x:d))</td>
</tr>
<tr>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>(n!)</td>
<td>the factorial (n!=n \times (n-1) \times \cdots \times 1)</td>
</tr>
<tr>
<td>(N^+)</td>
<td>the set of positive integers ([1,2,3,...])</td>
</tr>
<tr>
<td>(&amp;)</td>
<td>the bitwise AND</td>
</tr>
<tr>
<td>(\oplus)</td>
<td>the bitwise exclusive OR between two integers</td>
</tr>
</tbody>
</table>

2.2. Blum Blum Shub and XORshift

The Blum Blum Shub generator (Blum et al., 1986) (usually denoted by BBS) is defined by:

\[x^{n+1} = (x^n)^2 \mod m, \quad y^{n+1} = x^{n+1} \mod \log(m),\]

where \(m\) is the product of two prime numbers (these prime numbers need to be congruent to 3 modulus 4), and \(y^n\) is the returned binary sequence.

XORshift, on its part, is a category of very fast PRNGs designed by George Marsaglia (Marsaglia, 2003). Table 1 shows its working procedure (\(a,b,c\) are the shifting offsets).

2.3. Chaotic iterations

Definition 1. The set \(\mathbb{B}\) denoting \([0,1]\), let \(f: \mathbb{B}^N \times \mathbb{B}^N \rightarrow \mathbb{B}^N\) be an “iteration” function and \(S \subseteq \mathbb{B}\) be a strategy. Then, the so-called chaotic iterations are defined by (Terno and Robert, 1987):

\[
\begin{align*}
\{ x^0 \in \mathbb{B}^N, \\
\forall k \in \mathbb{N}^+, \forall \varepsilon \in [1:N], x^\varepsilon = f^{x^k} x^{k+1} - i \} \\
\text{if} S^i = j, \text{if} S^j = i.
\end{align*}
\]

In other words, at the \(n^{th}\) iteration, only the \(S^i\)-th cell is “iterated”. Note that in a more general formulation, \(S^n\) can be a subset of components and \(f(x^k)\) can be replaced by \(f(x^{k+1})\), where \(k < n\), describing for example delays transmission. For the general definition of such chaotic iterations, see, e.g., (Terno and Robert, 1987).

Chaotic iterations generate a set of vectors (Boolean vectors in this article), they are defined by an initial state \(x^0\), an iteration function \(f\), and a strategy \(S\) said to be a “chaotic strategy”.

Table 1 – XORshift algorithm.

| Input: \(x\) (a 64-bit word) | Output: \(r\) (a 64-bit word) |
| Parameters: \(a,b,c\) (integers) |
| \(x\) ← \(x \oplus (x<:a)\) |
| \(x\) ← \(x \oplus (x=>:b)\) |
| \(x\) ← \(x \oplus (x<:c)\) |
| \(r\) ← \(x\) |

An arbitrary round of XORshift
2.4. PRNGs based on CIs

Let us now recall some previously obtained results in the field of chaotic iteration based PRNGs.

2.4.1. CIPRNG(PRNG1,PRNG2) version 1

Let PRNG1 and PRNG2 be two given generators provided as input. Some chaotic iterations are fulfilled to generate a sequence \( \left( x^n \right)_{n\in\mathbb{N}} \in \left[ \mathbb{B}^N \right]^N \) of Boolean vectors, which are the successive states of the iterated system. Some of these vectors are randomly extracted and their components constitute the pseudorandom bit stream \( b \) (Bahi et al., 2009). Chaotic iterations are randomly extracted and their components constitute the pseudorandom bit stream (Bahi et al., 2011). First of all, some chaotic iterations have been completed to generate a sequence \( x^n \). Finally, some terms \( x^n \) are selected by a sequence \( m^n \), obtained using the PRNG1, as the pseudorandom bit sequence of our generator.

At each iteration, only the \( S^i \)-th component of state \( x^n \) is updated. Finally, some \( x^n \) are selected by a sequence \( m^n \), obtained using the PRNG1, as the pseudorandom bit sequence of our generator.

The basic design procedure of the first version of the CIPRNG generator is summed up in Algorithm 1. The internal state is \( x \), whereas \( a \) and \( b \) are computed by PRNG1 and PRNG2.

Algorithm 1. An arbitrary round of the Version 1 CI generator

| Input: | the internal state \( x \) (an array of \( N \) 1-bit words) |
| Output: | an array \( r \) of \( N \) 1-bit words |
| 1: | \( a \leftarrow PRNG1() \) |
| 2: | \( m \leftarrow a \mod 2 + c \) |
| 3: | while \( i = 0, \ldots, m \) do |
| 4: | \( b \leftarrow PRNG2() \) |
| 5: | \( S \leftarrow b \mod N \) |
| 6: | \( x_S \leftarrow \overline{x_S} \) |
| 7: | end while |
| 8: | \( r \leftarrow x \) |
| 9: | return \( r \) |

2.4.2. CIPRNG(PRNG1,PRNG2) version 2

The second version of the chaotic iterations based pseudorandom generators family is designed by the following process (Bahi et al., 2011). First of all, some chaotic iterations have to be done to generate a sequence \( \left( x^n \right)_{n\in\mathbb{N}} \in \left[ \mathbb{B}^N \right]^N \) of Boolean vectors, which are the successive states of the iterated system. Some of these vectors will be randomly extracted and their components constitute the pseudorandom bit stream will be constituted by their components. Such chaotic iterations are realized as follows.

- Initial state \( x^0 \in \mathbb{B}^N \) is a Boolean vector taken as a seed.
- Chaotic strategy \( \left( S^n \right)_{n\in\mathbb{N}} \in \left[ 1, N \right]^N \) is an irregular decimation of the PRNG2 sequence.

At each iteration, only the \( S^i \)-th component of state \( x^n \) is updated using the vectorial negation, as follows: \( x^i = x^{i-1} \) if \( i \neq S^i \), else \( x^i = \overline{x^{i-1}} \). Finally, some \( x^n \) are selected by a sequence \( m^n \) as the pseudorandom bit sequence of our generator, where \( \left( m^n \right)_{n\in\mathbb{N}} \in \mathbb{B}^N \) is computed using PRNG1.

The basic design procedure of this CIPRNG version 2 is summarized in Algorithm 2. The internal state is \( x \), while \( a \) and \( b \) are computed by the two inputted PRNGs. Finally, terms of the sequence \( m \) are defined as in Eq. (2).

\[
m^n = g_i(S^n) = \begin{cases} 
0 & \text{if } 0 \leq S^n < C_{32}^0, \\
1 & \text{if } C_{32}^0 \leq S^n < \sum_{i=0}^{32} C_{32}^i, \\
2 & \text{if } \sum_{i=0}^{1} C_{32}^i \leq S^n < \sum_{i=0}^{2} C_{32}^i, \\
\vdots \\
N & \text{if } \sum_{i=0}^{N-1} C_{32}^i \leq S^n < 1.
\end{cases}
\]

2.4.3. CIPRNG(PRNG1) version 3

Instead of updating only one cell at each iteration as in versions 1 and 2, we can choose a subset of components to update. Such a method 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 (Couturier and Guyeux, 2013):

\[
\begin{align*}
& \forall n \in \mathbb{N}, x^n = x^{0} - 1 \cdot S^n, \\
& x^n \in \left[ 0, 2^n - 1 \right] \setminus \{0\},
\end{align*}
\]

This mathematical rewriting can be understood as follows. The \( n \)-th term \( S^n \) of the sequence \( S \) (the PRNG1 provided as input), which is an integer of \( N \) binary digits, indicates the list of cells that must be updated in state \( x^n \) (an integer having \( N \) bits). More precisely, the \( k \)-th component of this state (a binary digit) is modified if and only if the \( k \)-th digit in the binary decomposition of \( S^n \) is equal to 1.

The single basic component presented in Eq. (3) is of ordinary use as a good elementary brick in various PRNGs. It corresponds to a discrete dynamical system in chaotic iterations.

Algorithm 2. An arbitrary round of the CIPRNG version 2

| Input: | the internal state \( x \) (\( N \) bits) |
| Output: | a state \( r \) of \( N \) bits |
| 1: | for \( i = 0, \ldots, N \) do |
| 2: | \( d_S \leftarrow 0 \) |
| 3: | end for |
| 4: | \( a \leftarrow PRNG1() \) |
| 5: | \( m \leftarrow g_1(a) \) |
| 6: | \( k \leftarrow m \) |
| 7: | while \( i = 0, \ldots, k \) do |
| 8: | \( b \leftarrow PRNG2() \mod N \) |
| 9: | \( S \leftarrow b \) |
| 10: | if \( d_S = 0 \) then |
| 11: | \( x_S \leftarrow \overline{x_S} \) |
| 12: | \( d_S \leftarrow 1 \) |
| 13: | else if \( d_S = 1 \) then |
| 14: | \( k \leftarrow k + 1 \) |
| 15: | end if |
| 16: | end while |
| 17: | \( r \leftarrow x \) |
| 18: | return \( r \) |
3. Introducing a new CIs-based PRNG

3.1. Presentation

It is possible to add more complexity in updating the subset at each iteration in Eq. (3) of CIPRNG version 3. When the updating function is the vectorial negation, this algorithm can be written as follows (Algorithm 3):

Algorithm 3. An arbitrary round of the new version of CIPRNGs

Input: the internal state \(x\) (N bits)
Output: a state \(r\) of N bits
1: for \(i = 1, \ldots, M\) do
2: \(S(i) \leftarrow PRNG2 \cdot x(i)\)
3: end for
4: \(T \leftarrow PRNG1()\)
5: \(r \leftarrow x \oplus g_2(S(\{1\}, S(\{2\}, \ldots, S(M))\))\)
6: return \(r\)

In the algorithm given above, \(S(\{1\}), S(\{2\}), \ldots, S(M)\) are M pseudorandom number sequences generated by \(M\) XORshifts, while \(T^n\) is obtained by using a cryptographically secure PRNG like the BBS one. \((l_1, l_2, \ldots, l_M) \in \{0, 1\}^M\) is the binary representation of the \(2^M\)-bit integer \(T^n\). \((T^n)\) is used to decimate the sequences \(S(\{1\}), S(\{2\}), \ldots, S(M)\), with a bitwise exclusive or (\(\oplus\)), according to the following decimation rule:

- if \(l_k = 0\), then \(S^n(k)\) is removed,
- else \(S^n(k)\) is used during the bitwise exclusive or computation.

In brief, the produced output sequence \(x^n\), based on chaotic iterations, is updated by a bitwise exclusive or of an irregular decimation of \(S(\{1\}), S(\{2\}), \ldots, S(M)\), according to the bits of \(T^n\).

The M terms \(S(\{1\}), \ldots, S^n(M)\) of the n-th iterate of sequences \(S(\{1\}), S(\{2\}), \ldots, S(M)\) are integers of N bits. Each term \(T^n\) of sequence T is an integer having M binary digits. Such a \(T^n\) contains the list of cells to update in the state \(x^n\) of the system, which is also an integer of N bits. The way to update these cells is given by the function \(g_2(S^n(\{1\}), S^n(\{2\}), \ldots, S^n(M), T^n)\), which is defined by Algorithm 4. Indeed, each bit in \(T^n\) determines whether its corresponding \(S^n(i)\) is used in the bitwise exclusive or computation defining \(x^n\). More precisely, the \(k\)-th binary digit of \(x^n(k)\) is modified if and only if the \(k\)-th digit in the binary decomposition of \(g_2(S^n(\{1\}), S^n(\{2\}), \ldots, S^n(M), T^n)\) is equal to 1.

Algorithm 4. The \(g_2(\{S^n(\{1\}), S^n(\{2\}), \ldots, S^n(M), T^n)\) function

Input: sequences \(S^n(\{1\}), \ldots, S^n(M)\), and \(T^n\)
Output: a state \(r\) (N bits)
1: \(r \leftarrow 0\)
2: \(M \leftarrow \text{size of } T^n\)
3: for \(i = 1, \ldots, M\) do
4: if \(T^n \& (2^{i-1}) \neq 0\) then
5: \(r \leftarrow r \oplus S^n(i)\)
6: end if
7: end for
8: return \(r\)

3.2. Security analysis

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

Definition 2. A pseudorandom number generator \(G\) is cryptographically secure if for any probabilistic polynomial time algorithm \(D\), for any positive polynomial \(p\), and for all sufficiently large \(m\)’s,

\[|Pr[D(G(U_m))] - Pr[D(U_m)]| < \frac{1}{p(m)},\]

where \(U_m\) is the uniform distribution over \([0,1]^m\) and the probabilities are taken over \(U_m\), \(U_{\text{Gm}}\) 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 \(f\) into any polynomial function \(f\) satisfying \(f(m) > m\). It has been proven in Couturier and Guyeux (2013) that, if the inputted generator is cryptographically secure, then the CIPRNG version 3 is also cryptographically secure. The proof for the updated version of CIPRNG given by Algorithm 3 is given thereafter.

The generation schema developed in Algorithm 3 is based on \(M + 1\) pseudorandom generators. Let \(H_1, H_2, \ldots, H_M\) be the PRNGs that are used to update the bits of the internal state, and \(l\) be the PRNG that determines which \(H_i\) is available in this updating round. We may assume, without loss of generality, that for any string \(S_j(0)\) of size \(N\), the size of \(H_j(S_j(0))\) is \(kN\). Then any string \(T_0\) of size \(M\) has \(l(T_0)\) with \(kM\), where \(k > 2\). It means that \(l(\text{NM}) = kM\) and \(l(M) = kM\).

Let \(S_1(1), S_1(2), \ldots, S_1(M), \ldots, S_M(1), S_M(2), \ldots, S_M(M)\), and \(T_{1}, \ldots, T_{k}\) be the \(M + 1\) string sequences (the strings \(S\) have \(N\) bits whereas strings \(T\) are constituted by \(M\) bits). Then \(H_j(S_j(0)) = S_j(k)\) and \(l(T_{0}) = T_{1} \ldots T_{k}\) that is, \(H_i(S_i(0))\) are the concatenation of \(S_i(0)\) and \(T_i\). Using this formulation, the generator \(X\) defined in Algo.3 is the algorithm that maps any string
$x^0 g_x(S^1(1), S^1(2), ..., S^1(M), T^1)$

of length $M + NM + N$ into the string:

$x^0 @ g_x(S^1(1), S^1(2), ..., S^1(M), T^1), x^0 @ g_x(S^1(1), S^1(2), ..., S^1(M), T^1)@ g_x(S^1(1), S^1(2), ..., S^1(M), T^1), ...$

$x^0 @ g_x(S^1(1), S^1(2), ..., S^1(M), T^1)$.

In particular, one have $l_d(M + NM + N) = kn = l_d(M)$, where $kn \geq M + NM + N$.

We announce that if the PRNG $I$ is cryptographically secure, then the new one $X$ provided by Algorithm 3 is also cryptographically secure.

Proposition 1. If $I$ is a cryptographically secure PRNG, then $X$ is also a cryptographically secure PRNG.

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

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

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

1. Decompose $w$ into $w_1, ..., w_k$.
2. Pick a string $y$ of size $N$ uniformly at random.
3. Pick $M$ strings of size $kn$: $u_1, ..., u_M$.
4. Decompose each $u(j)$ into $u_0, ..., u_{l_d(j)}$.
5. Define $t_i = \Theta^{j-1}_{j-0}((w_j @ j) & 1) \times u_j(1)$. By construction, one has for every $t_i$,

$$D'(w) = D(\varphi_y(t_i)), \quad (5)$$

where $y$ is randomly generated. Moreover, for each $y$, $\varphi_y$ is injective: if

$$(y @ t_1)@ t_2, ... @ (y @ t_{l-1}) = (y @ t_1')@ t_2', ... @ (y @ t_{l-1}')$$

then for every $1 \leq j \leq k$, $y @ t_{i-1}' = y @ t_{i-1}'$. It follows, by direct induction, that $t_i = t_i'$.

Consider also for each $u(j) \in [kN]$ the function $\varphi_u$ from $[kN]$ into $[kN]$ mapping $w = w_1, ..., w_k$ (each $w_i$ has length $M$) to:

$$\Theta^{j-1}_{j-0}((w_1 @ j) & 1) \times u_j(1) \Theta^{j-1}_{j-0}((w_2 @ j) & 1) \times u_j(1) \Theta^{j-1}_{j-0}((w_3 @ j) & 1) \times u_j(1) \Theta^{j-1}_{j-0}((w_4 @ j) & 1) \times u_j(1) ...$$

$$\Theta^{j-1}_{j-0}((w_k @ j) & 1) \times u_j(1)$$

The $u(j)$ is generated by $H(j)$ PRNG and $\varphi_u$ is injective, so if $\Theta^{j-1}_{j-0}((w_1 @ j) & 1) \times u_j(1) \Theta^{j-1}_{j-0}((w_2 @ j) & 1) \times u_j(1) \Theta^{j-1}_{j-0}((w_3 @ j) & 1) \times u_j(1) \Theta^{j-1}_{j-0}((w_4 @ j) & 1) \times u_j(1) ...$$

then $u_1 = u'_1$. So, according to Eq. (5):

$$D(w) = D(\varphi_y(w)) \quad (6)$$

Furthermore, using Eq. (6), one has $Pr[D(U_{km}) = 1] = Pr[D(\varphi_y(U_{km})) = 1]$ and, therefore,

$$Pr[D(U_{km}) = 1] = Pr[D(U_{km}) = 1]. \quad (7)$$

Now, using Eq. (6) again, one has for every $x$,

$$D'(l(x)) = D(\varphi_y(l(x))), \quad (8)$$

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

$$Pr[D'(l(U_{km})) = 1] = Pr[D(X(U_{MN,N}, N)) = 1]. \quad (9)$$

Using Eq. (9) minus Eq. (7), one can deduce that there exists a polynomial time probabilistic algorithm $D'$, a positive polynomial $p$, such that for all $k_0$ there exists $M + NM + N \geq k_0$ satisfying

$$Pr[D'(l(U_{km})) = 1] - Pr[D'(U_{km}) = 1] \geq \frac{1}{p(M + NM + N)}$$

proving that $I$ is not secure, which is a contradiction.

3.3. Efficient and cryptographically secure PRNG based on chaotic iterations

Table 2 describes an efficient, chaotic, and cryptographically secure PRNG having a perfect statistical profile. It can be divided into two parts, as explained below.

The first part is based on Algorithm 3. This part is very suitable for FPGA as it can be easily arranged to be processed in parallel. Additionally, as stated in the previous section, this new version of our CIPRNG can be configured to be cryptographically secure. To obtain such a secured CIPRNG, $M + 1$ generators must be provided as inputs, one of them being cryptographically secure. In the proposed design, the BBS

Table 2 - Efficient pseudorandom generator designed for FPGAs.

<table>
<thead>
<tr>
<th>Input: $x$ (a 32-bit word)</th>
<th>Output: $r$ (a 32-bit word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$ -- XORshift10;</td>
<td>$t_2$ -- XORshift20;</td>
</tr>
<tr>
<td>$t_4$ -- bbs0;</td>
<td></td>
</tr>
<tr>
<td>if $t_4$ then $x = x @ (t_1 @ 0)$;</td>
<td>if $t_4$ then $x = x @ (t_1 @ 0)$;</td>
</tr>
<tr>
<td>$t_4$ -- $x @ (t_1 @ 0)$;</td>
<td></td>
</tr>
<tr>
<td>if $t_4$ then $x = x @ (t_1 @ 0)$;</td>
<td>if $t_4$ then $x = x @ (t_1 @ 0)$;</td>
</tr>
<tr>
<td>$r = x$;</td>
<td></td>
</tr>
<tr>
<td>return $r$;</td>
<td></td>
</tr>
</tbody>
</table>

An arbitrary round of the algorithm
secured generator has been chosen due to its simplicity. Due to its slowness, this BBS is used to compute the T sequence of Algorithm 3. The size of $m$ is 32 bits. It is well known that the $\log(\log(m))$ least significant bits can be securely extracted at each iteration of the BBS (Vitini et al., 1998). So we set $M = 3$, leading to the selection of 2 XORshifts playing the role of $S$. They are denoted XORshift1 and XORshift2 in Table 2. Each XORshift output is separated into two 32 bits blocks, leading to four 32 bits numbers. Three of them (namely, the two 32 bits blocks of XORshift1 and the first one of XORshift2) are controlled by the bits outputted by the BBS according to Algorithm 3. The last 32 bits block, on its part, is used in the second stage of the algorithm. More precisely, if one bit in the bbs output is 0, then the corresponding 32 bits number is not used during the exclusive or processing, whereas it is considered if this BBS bit is 1.

According to our experiments, the sole first part of the algorithm cannot produce a statistically perfect output. It is not contradictory with Proposition 1, as the cryptographically secure property is an asymptotic one. Following the approach detailed in Bahl et al. (2012a), we have used the chaotic iterations to improve the statistical behavior of the proposed generator. Hence, the second stage of the algorithm consists in using the last 32 bits block of XORshift2 to realize Eq. (3) on the output of the first part. By doing so, we obtain in Table 2 a generator being both chaotic and cryptographically secure (Couturier and Guyeux, 2013).

This algorithm has a very similar design than the efficient CIPRNG version GPU presented in Couturier and Guyeux (2013), which has successfully passed the stringent TestU01 battery of statistical tests (L’Ecuyer and Simard, 2012). However, in the GPU version, no BBS is used to determine which bits in the most significant binary block of size 32 of XORshift will be used in the process. Moreover, the number of used XORshifts is not the same in this cryptographically secured generalization of the generators formerly described in Couturier and Guyeux (2013). Table 4 shows the test results of the proposed CIPRNG against the NIST battery (Rukhin et al., 2011). Results of XORshift and BBS are provided too, for the sake of comparison. Let us remark that, even though the Blum Mulos Shub generator is cryptographically secure (which is a property independent from the chosen modulo $m$), the very small value chosen for $m$ makes it unable to pass the NIST battery. Obviously, best statistical performances are obtained using the proposed CIPRNG. Furthermore, this CIPRNG can also pass both the DieHARD (Marsaglia, 2012) and TestU01 test suites. These results have been obtained on an INTEL i5 dual core computer, with 2.3 GHz CPUs and 4 GB of RAM memory. As detailed in Table 3, only less than 7 h have been required for this generator to finish the biggest test (namely, the BigCrush in TestU01). So it runs much faster than CIPRNG versions 1 and 2, and its speed is close to CIPRNG version 3.

### 4. PRNG design using a FPGA

In order to take benefits from the power of FPGAs, a computation needs to be spread into several independent blocks of threads that can be computed simultaneously. Indeed, performance on FPGAs is directly related to the number of threads and of logistical elements that are used during computation. It decreases when then number of branching instructions (if, while, etc.) increases. Having these rules in mind, it is possible to build a Verilog-HDL (Verilog hdl, 2008) program similar to the algorithm presented in Table 2, leading to a pseudorandom numbers generation with chaotic properties on FPGA. This generator contains three PRNG objects that use the exclusive or operation: two XORshifts and a BBS. Their processing are described thereafter.

#### 4.1. Design of XORshift

The structure of XORshift designed in Verilog-HDL is shown in Fig. 1a. There are four inputs:

- the first one is the initial state, which costs 64 bits of register units,
- the remainder ones are used to define the shift operations.

Let us remark that, in FPGA, the shift operation costs nothing, as it simply consists in using different bit cells of the input. We can thus conclude that there are $64 - s1 + 64 - s2 + 64 - s3 = 192 - s1 - s2 - s3$ required logic gates elements for XORshifts processing.

#### 4.2. Design of BBS

Fig. 1b gives the proposed design of the BBS generator in FPGAs. There are two inputs of 32 bits, namely $b$ and $m$.

### Table 3 – Three CIPRNG Versions in BigCrush of TestU01 test suite.

<table>
<thead>
<tr>
<th>PRNGs</th>
<th>Time to finish</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBS</td>
<td>43 h 33 min</td>
<td>Fails</td>
</tr>
<tr>
<td>XORshift</td>
<td>5 h 34 min</td>
<td>Fails</td>
</tr>
<tr>
<td>CIPRNG Version 1(XORshift,XORshift)</td>
<td>35 h 22 min</td>
<td>Fails</td>
</tr>
<tr>
<td>CIPRNG Version 2(XORshift,XORshift)</td>
<td>19 h 47 min</td>
<td>Passes</td>
</tr>
<tr>
<td>CIPRNG Version 3(XORshift)</td>
<td>6 h 6 min</td>
<td>Fails</td>
</tr>
<tr>
<td>New CIPRNG (BBS,XORshift × 2)</td>
<td>6 h 36 min</td>
<td>Passes</td>
</tr>
</tbody>
</table>

### Table 4 – NIST SP 800-22 test results ($p_0$).

<table>
<thead>
<tr>
<th>Method</th>
<th>CIPRNG</th>
<th>XORshift</th>
<th>BBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (Monobit)</td>
<td>0.372881</td>
<td>0.145326</td>
<td>0.32435</td>
</tr>
<tr>
<td>Frequency Test within a Block</td>
<td>0.772324</td>
<td>0.028817</td>
<td>0.00000</td>
</tr>
<tr>
<td>Runs</td>
<td>0.464244</td>
<td>0.739918</td>
<td>0.00000</td>
</tr>
<tr>
<td>Longest Run of Ones in a Block</td>
<td>0.183231</td>
<td>0.554420</td>
<td>0.00000</td>
</tr>
<tr>
<td>Binary Matrix Rank</td>
<td>0.251300</td>
<td>0.236810</td>
<td>0.00000</td>
</tr>
<tr>
<td>Discrete Fourier Transform (Spectral)</td>
<td>0.324348</td>
<td>0.514124</td>
<td>0.00000</td>
</tr>
<tr>
<td>Non-overlapping Template Matching</td>
<td>0.235439</td>
<td>0.512363</td>
<td>0.00000</td>
</tr>
<tr>
<td>Overlapping Template Matching</td>
<td>0.633420</td>
<td>0.595549</td>
<td>0.00000</td>
</tr>
<tr>
<td>Universal Statistical</td>
<td>0.082345</td>
<td>0.122325</td>
<td>0.00000</td>
</tr>
<tr>
<td>Linear Complexity</td>
<td>0.987524</td>
<td>0.249284</td>
<td>0.00000</td>
</tr>
<tr>
<td>Serial ($m = 10$)</td>
<td>0.093482</td>
<td>0.495847</td>
<td>0.04355</td>
</tr>
<tr>
<td>Approximate Entropy ($m = 10$)</td>
<td>0.334353</td>
<td>0.000000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Cumulative Sums (Cusum)</td>
<td>0.797792</td>
<td>0.074404</td>
<td>0.00000</td>
</tr>
<tr>
<td>Random Excursions</td>
<td>0.343226</td>
<td>0.507812</td>
<td>0.00000</td>
</tr>
<tr>
<td>Random Excursions Variant</td>
<td>0.182837</td>
<td>0.289594</td>
<td>0.00000</td>
</tr>
<tr>
<td>Success</td>
<td>15/15</td>
<td>14/15</td>
<td>2/15</td>
</tr>
</tbody>
</table>


Register $b$ stores the state of the system at each time (after the square computation). $m$ is also a register that saves the value of $M$, which must not change. Another register $b_{\text{extend}}$ is used to combine $b$ to a data having 64 bits, with a view to avoid overflow. After the last computation, the three LSBs from the output are returned. Let us remark that a BBS is performed at each time unit.

### 4.3. Design of the chaotic iterations

Two XORshifts and one BBS are connected together, in order to constitute the proposed CIPRNG. As detailed in Fig. 1c, the three bits of the BBS output are switches for the corresponding 32 bits XORshift outputs. Every round of the processing costs 2 time units to be performed: in the first clock, the four PRNGs are processed in parallel, whereas in the second one, the results of these generators are combined with the current state of the system, in order to produce the output of 32 bits.

EP2C8Q208C8 FPGA from Altera company’s CYCLONE II series has been used in our experiments. By default, its working frequency is equal to 50 MHz. However, it is possible to increase it until 400 MHz by using the phase-lock loop (PLL) device. In that situation, the CIPRNG designed on this FPGA can produce about 6,400 Mbits per second (that is, 400 MHz ÷ 2(times) × 32(bits)), see Fig. 3, while using 3,652 of the 8,256 logic elements in EP2C8Q208C8 (as depicted in Fig. 2). Let us recall that the CIPRNG version 1 has already been implemented in FPGAs (Bahi et al., 2013b) by using the same platform (6114 logic elements cost and 16 bits output). However, the proposed new scheme shows better source usage and a larger output size.

In the next section, an application of this CSPRNG designed on FPGA in the information hiding security fields is detailed, to show that this hardware pseudorandom generator is ready to use.

### 5. An information hiding application

Information hiding has recently become a major information security technology, especially with the increasing importance and widespread distribution of digital media through the Internet (Wu and Guan, 2007). It includes several techniques like digital watermarking. The aim of digital watermarking is to embed a piece of information into digital documents, such as pictures or movies. This is for a large panel of reasons, such as: copyright protection, control utilization, data description, content authentication, and data integrity. For these reasons, many different watermarking schemes have been proposed in recent years. Digital watermarking must have essential characteristics, including: security, imperceptibility, and robustness. Chaotic methods...
have been proposed to encrypt the watermark before embedding it in the carrier image for these security reasons. In this paper, a watermarking algorithm based on the chaotic PRNG presented above is given, as an illustration of use of this PRNG based on chaotic iterations.

5.1. Most and least significant coefficients

Let us firstly recall the definitions of most and least significant coefficients, as they have been formerly introduced in (Guyeux et al., 2010; Bahi and Guyeux, 2010b).

Definition 3. For a given image, the most significant coefficients (in short MSCs), are coefficients that allow the description of the relevant part of the image, i.e., its most rich part (in terms of embedding information), through a sequence of bits.

For example, in a spatial description of a grayscale image, a definition of MSCs can be the sequence constituted by the first three bits of each pixel as shown in Fig. 4b. In a discrete cosine frequency domain description, each 8 × 8 block of the carrier image is mapped to a list of 64 coefficients. The energy of the image is contained in the firsts of them. After binary conversion, the first fourth coefficients of all these blocks can constitute a possible sequence of MSCs.

Definition 4. By least significant coefficients (LSCs), we mean a translation of some insignificant parts of a medium in a sequence of bits (insignificant can be understand as: “which can be altered without sensitive damages”).

These LSCs can be for example, the last three bits of the gray level of each pixel, in the case of a spatial domain watermarking of a grayscale image, as in Fig. 4c.

Discrete cosine, Fourier, and wavelet transform can be used to define LSCs and MSCs, in the case of frequency domain watermarking, among other possible choices. Moreover, these definitions are not limited to image media, but can easily be extended to the audio and video media as well.

LSCs are used during the embedding stage: some of the least significant coefficients of the carrier image will be chaotically chosen and replaced by the bits of the mixed watermark. With a large number of LSCs, the watermark can be inserted more than once and thus the embedding will be more secure and robust, but also more detectable.

The MSCs are only useful in the case of authentication: encryption and embedding stages depend on them. Hence, a coefficient should not be defined both as an MSC and a LSC; the latter can be altered, while the former is needed to extract the watermark. For a more rigorous definition of such LSCs and MSCs (see, e.g., Bahi et al., 2012b).

Fig. 2 – 3,652 of the 8,256 logic elements cost in EP2C8Q208C8 FPGA board.

Fig. 3 – Outputs of each component in clock step units.

Fig. 4 – Spatial MSCs and LSCs of Lena.

(a) Lena

(b) MSCs of Lena

(c) LSCs of Lena
5.2. Stages of the algorithm

We recall now a formerly introduced watermarking scheme, which consists of two stages: (1) mixture of the watermark and (2) its embedding (Bahi and Guyeux, 2010c).

5.2.1. Watermark mixture

Firstly, for security reasons, the watermark can be encrypted before its embedding into the image. A common way to achieve this stage is to use the bitwise exclusive or (XOR), for instance between the watermark and the PRNG described previously. In this paper and similarly to (Bahi and Guyeux, 2010c), we will use another mixture scheme based on chaotic iterations. Its chaotic strategy, defined with our PRNG, will be highly sensitive to the MSCs in the case of an authenticated watermarking, as stated in Bahi and Guyeux (2010a).

5.2.2. Watermark embedding

Some LSCs will be substituted by all bits of the possibly mixed watermark. To choose the sequence of LSCs to be altered, a number of integers, lesser than or equal to the number N of LSCs corresponding to a chaotic sequence \( U^k \), is generated from the chaotic strategy used in the mixture stage. Thus, the \( U^k \)-th least significant coefficient of the carrier image is substituted by the \( k \)-th bit of the possibly mixed watermark. In the case of authentication, such a procedure leads to a choice of the LSCs that are highly dependent on the MSCs.

5.2.3. Extraction

The chaotic strategy can be regenerated, even in the case of an authenticated watermarking because the MSCs have not been changed during the embedding stage. Thus, the altered LSCs can be found, the mixed watermark can be rebuilt, and the original watermark can be obtained again. If the watermarked image is attacked, then the MSCs will change. Consequently, in the case of authentication and due to the high sensitivity of the embedding sequence, the LSCs designed to receive the watermark will be completely different. Hence, the result of the recovery will have no similarity with the original watermark: authentication is reached.

5.3. The FPGA setting

The 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs is applied in this information hiding specific application. Nios II incorporates many enhancements over the original Nios architecture, making it more suitable for a wider range of embedded computing applications, from DSP to system-control (Introduction to the alter, 2011).

In this application, the NIOS II system can read the image from the HOST computer side. After processing, the results are also transmitted back into the host system using NIOS II. Our NIOS II uses the most powerful version the CYCLONE II can support (namely, the NIOS II/f one). 4 KB on chip memory and 16 MB SDRAM are set, and the PLL device is used to enhance the clock frequency from 50 to 200 MHz. Finally, the data connection bus NIOS II system and generator works in 32 bits.

5.4. Results

To prove the efficiency and the robustness of the proposed algorithm, some attacks are performed on some chaotically watermarked images. For each attack, a similarity percentage with the original watermark is computed. This percentage is the number of equal bits between the original and the extracted watermark, shown as a percentage. A result lesser than or equal to 50% implies that the image has probably not been watermarked.

5.4.1. Cropping attack

In this kind of attack, a watermarked image is cropped. In this case, the results in Table 5 have been obtained. In Fig. 5, the decrypted watermarks are shown after a crop of 50 pixels and after a crop of 10 pixels, in the authentication case.

By analyzing the similarity percentage between the original and the extracted watermark, we can conclude that in the case of unauthentication, the watermark still remains after a cropping attack. The desired robustness is reached. It can be noticed that cropping sizes and percentages are rather proportional. In the case of authentication, even a small change of the carrier image (a crop by 10 × 10 pixels) leads to a really different extracted watermark. In this case, any attempt to alter the carrier image will be signaled, thus the image is well authenticated.

5.4.2. Rotation attack

Let \( r_\theta \) be the rotation of angle \( \theta \) around the center (128,128) of the carrier image. So, the transformation \( r_{-\theta} \) is applied to

![Image](a) Unauthenticated (10 × 10)  (b) Authentication (10 × 10)  (c) Unauthenticated (50 × 50)

Fig. 5 – Extracted watermark after a cropping attack (zoom ×2).
the watermarked image. The results in Table 5 have been obtained. The same conclusion as above can be declared.

5.4.3 Gaussian noise
A watermarked image can be also attacked by the addition of a Gaussian noise, depending on a standard deviation. In this case, the results in Table 5 are obtained, which are quite satisfactory again.

6. Conclusion and future work
In this article, a pseudorandom generator proposed in a former research work has been improved in terms of efficiency and randomness. We have also shown that this hardware generator can be cryptographically secure. By using a BBS generator and due to a new approach in the way the CIPRNG version 3 uses its strategies, the generator based on chaotic iterations works faster and is more secure. This new CIPRNG is able to finish BigCrush in less than 7 h when considering software implementation, and to reach 6000 Mbps in FPGA hardware. These considerations enable us to claim that this CIPRNG(BBS, XORshift) offers a sufficient speed and level of security for a whole range of applications where secure generators are required as cryptography and data hiding.

In future work, we will continue to explore new strategies and iteration functions. The chaotic behavior of the proposed generator will be deepened by using the various tools provided by the mathematical theory of chaos. Additionally a probabilistic study of its security will be done. Lastly, new applications in computer science will be proposed, like watermarking in electronic maps.

REFERENCES

Bahi J, Guyeux C. Topological chaos and chaotic iterations, application to hash functions. In: IJCNN’10, Int. Joint Conf. on Neural Networks, joint to WCCI’10, IEEE World Congress on Computational Intelligence, Barcelona, Spain, July 2010. p. 1–7. [best paper award]


Devaney RL. An introduction to chaotic dynamical systems. 2nd ed. 2003 [Westview Pr (Short Disc)].


Marsaglia G. Diehard: a battery of tests of randomness: http:// stat.fsu.edu/~geo/diehard.html. [accessed 02.01.12].


