CIPRNG: A VLSI Family of Chaotic Iterations Post-Processings for $\mathbb{F}_2$-Linear Pseudorandom Number Generation Based on Zynq MPSoC

Mohammed Bakiri, Jean-François Couchot, and Christophe Guyeux

***Abstract*—Hardware pseudorandom number generators are continuously improved to satisfy both physical and ubiquitous computing security system challenges. The main contribution of this work is to propose two post-processing modules in hardware, to improve the randomness of linear PRNGs while succeeding in passing the TestU01 statistical battery of tests. They are based on chaotic iterations and are denoted by CIPRNG-MC and CIPRNG-XOR. They have various interesting properties, encompassing the ability to improve the statistical profile of the generators on which they iterate. Such post-processing have been implemented on FPGA and ASIC without inferring any blocs (RAM or DSP). A comparison in terms of area, throughput, and statistical tests, is performed. The hardware pseudorandom number generation can reach a throughput/latency ratio equal to 8.5 Gbps for Zynq-FPGA and 10.9 Gbps for ASIC, being thus the fastest FPGA generators based on chaos that can pass TestU01. In particular, it is established that CIPRNG-XOR is 2.5 times faster and 5 times more efficient that almost all linear PRNGs who pass TestU01.**

***Index Terms*—Pseudorandom Number Generators, Discrete dynamical systems, Statistical Tests, Hardware Security, Applied Cryptography, System on Chip, FPGA.**

I. Introduction

DESPITE its long history, random generation still remains a hot topic, with the emergence of the so-called Random as Service or Entropy as Service [1] needs. It also becomes a key element in lightweight security cores in IoT devices. Finally, cloud services suffer when they have to generate multiple virtual machine instances (VM) from a golden image: they look like to have a very limited ability in randomness harvesting [2]. Despite the common use of these generators in many applications as described above, their integration into System on Chip becomes highly desirable, particularly for IoT and Smart Cards. Therefore, the practical purpose of current research works is to provide compact, high throughput, secure, and reconfigurable pseudorandom generators for hardware applications.

Let us recall that a random number generator algorithm is defined by the state space S of the generator, the transition mapping function f, the output extractor function g from a given state, and the seed x^0 [3]. The random output sequence is $y^1, y^2, \ldots$, where each y^t is generated by the two main steps described thereafter. The first step applies the transition function according to the recurrence $x^{t+1} = f(x^t)$, where x^t and x^{t+1} both belong to S. The mapping function f can be either an algorithm that deterministically produces random-like numbers in a discrete and finite state space. Such generators are denoted as pseudorandom number generators (PRNGs). Differently, f can be based on a physical source of entropy to produce randomness, thus making S a continuous space. The whole approach is thus called a True random number generator (TRNG). The second step consists in applying the function generator to the new internal state leading to the output y^t, that is, $y^t = g(x^t)$. There is a large variety of such recursive generators, which can be either linear or not, chaotic...

Random number generation is more studied in mathematics for software aspects, whereas hardware and semiconductor solutions are deeply investigated for true random generation. On the one hand, linear PRNGs are a special case of linear recurrence modulo 2 (that is, S is $\mathbb{F}_2$). Many research works and solutions are regularly proposed to increase their performance and statistical profile, and their linearity and security are investigated accordingly. Unfortunately, only a few of these linear PRNGs are analyzed in details at the hardware level, such as FPGA and ASIC. On the other hand, chaotic pseudorandom number generators (CPRNGs) are non-linear generators of the form: $x^0 \in \mathbb{R}$ and $x^{t+1} = f(x^t)$, where f is a chaotic map. They are an attractive application of the mathematical theory of chaos. Reasons explaining such an interest encompass their sensitivity to initial conditions, their unpredictability, and their ability of reciprocal synchronization [4]. Truly chaotic generators are a good demonstration of these characteristics: their period is infinite, hardware resources are compact, and statistical tests are often succeed quite reasonably [5]–[7].

One natural question that arises is: how can we inject disorder in a deterministic digital system, while respecting the mathematical definitions of chaos provided by Devaney [8] and Li-Yorke [9] on such finite state machines? An usual answer in digital embedded systems is to consider pseudo-chaotic generators instead of truly chaotic ones [6], [10]–[12]. In spite of the quality of the TRNG output based on a chaotic phenomenon, most of these techniques are however produced in a manner that is either slow (*i.e*, in a range of some Kbps to Mbps, to extract noise or jitter from a given component [13]) or costly (*e.g.*, extracting or measuring some noise using oscilloscope or laser [5], [14]). Additionally, to embed these TRNGs in a pure digital platform is an extreme challenge, where the main concern is calibration of the bias phenomenon coming from analog inputs. Digital TRNGs lead

M. Bakiri is from Centre des Techniques Avancées (CDTA), Alger, Algeria.
M. Bakiri, Couchot and Guyeux were with the Femto-ST Institute, UMR 6174 CNRS, Université de Bourgogne Franche-Comté, France.
E-mail: jean-francois.couchot@univ-fcomte.fr.
Manuscript received XXX; revised XXX.

thus to an uncontrollable uniformity and performance of the outputs compared to the theory. Conversely, chaotic PRNG appears as a convenient solution in SoC platforms such as Zynq based FPGA [15].

Additionally, these PRNGs have various drawbacks, particularly they fail statistical tests of linear complexity of their outputs. This work notably illustrates that 32-bits length internal state is sufficient to pass the linear complexity tests only if post-processing operations (permutations, for instance) are applied to scramble the output. Another solution that comes in mind is to enlarge the internal state space whilst conserving the same output length (32 bits). However such a second answer is contradictory with the objective of hardware implementation which is notably to keep resources as reduced as possible.

This article is an extended version of a paper accepted at Secrypt 2016, the 13*th* International Conference on Security and Cryptography [16]. We have reported the initial design and evaluation of Chaotic Iterations based PRNG (CIPRNG) as a possible post-processing for hardware PRNGs, demonstrating its benefits compared to other linear PRNGs. This proposal focuses on adding chaos (as mathematically defined by Devaney [8] and Li-Yorke [9]) on linear PRNGs as a post-processing, in which at each iteration, only a subset of components of the iteration vector is updated. In this article, we undertake a deeper evaluation of these linear PRNGs, encompassing statistical tests, throughput & latency, and Berlekamp-Massey algorithm [17] analyze. To the best of our knowledge, no paper has really deeply investigated hardware implementations of such linear PRNGs. Pseudo-chaos generators implemented on FPGA have been considered too in our investigations, which use various map functions like the so-called Logistic Map [18], the Timing Reseeding [19], or Differential Chaotic [20]. We also provide a detailed description about the SoC platform for implementation and randomness tests. In addition to the details of the CIPRNG-XOR (presented in [16]) this article provides two new Chaotic Iterations based post processes, namely Multi-Cycle CIPRNG-MC and Multi-Cycle Multi-Dimension (CIPRNG-MCMD). The underlying theory is emphasized since our proposal has been completely proven in the rigorous framework of chaos theory. Finally, we improve hardware aspects on FPGA by merging them in ASIC implementation using UMC-65nm Low Leakage Technology, which is done to compare area, throughput, and power consumption of investigated generators without inferring any blocks (DSP, RAM).

This extension of a conference article is organized as follows. Section II discusses hardware design (FPGAs and ASIC) and analysis a set of selected linear and chaotic pseudorandom number generators. Performance is regarded in Section III: frequency, area size, weaknesses, and computation complexity are investigated to select which linear PRNGs can be used for post-processing. Then, in Section IV, we present the mathematical topology foundation of chaotic iterations [21], while its application for PRNGs is detailed in Section V. We compare the implementation of both linear PRNGs and chaotic iterations on FPGAs using Zynq platform in the next section, while the ASIC implementation is discussed in Section VII. This article ends by statistical tests (Sec. VIII) and a conclusion section, in which our study is summarized and intended future work is outlined.

II. Background of Linear and Chaotic PRNGs

A. $\mathbb{F}_2$ Linear PRNGs

Let $\mathbb{F}_2$ be the finite field of cardinality 2. Let us firstly recall that a common way to define a pseudorandom number generator is to consider two functions f and g, and a linear recurrence defined by

$$\begin{array}{ll} f : \mathbb{F}_2^{\mathsf{N}} \to \mathbb{F}_2^{\mathsf{N}} & g : \mathbb{F}_2^{\mathsf{N}} \to \mathbb{F}_2^{\mathsf{M}} \\ x^{t+1} = f(x^t) \quad \text{and} & y^t = g(x^t) \end{array}$$

where usually $\mathsf{N} > \mathsf{M}$, g is one way, x^0 is a seed provided by the user, and y^t is returned to the user.

Let us remind that linear PRNGs are a special case of linear recurrence modulo 2. Therefore, a linear PRNG of w bits can be defined by the following equations:

$$x^{t+1} = A \times x^t \tag{1}$$

$$y^t = B \times x^t \tag{2}$$

$$r^t = \sum_{\ell=l}^{w} y^t_{\ell-1}\ 2^{-\ell} \tag{3}$$

Equation (1) defines the function f, where $x^t = (x^t_0, \ldots, x^t_{k-1}) \in \mathbb{F}_2^k$ is the k-bit vector at step t and A is a $k \times k$ transition matrix with k-bit $\mathbb{F}_2$-vector.

Equations (2) and (3) define the function g, where $y^t = (y^t_0, \ldots, y^t_{w-1}) \in \mathbb{F}_2^k$ is the w-bit output vector at step t, and B is a $w \times k$ output transformation matrix with elements in $\mathbb{F}_2$. This latter produces the output bits that correspond to the internal RNG state, which is rewritten as $r^t \in [0,1]$: the output at step t. Let us provide some examples of such linear PRNGs.

Linear Feedback Shift Register (LFSR): Well-known examples of such generators are LFSR113 [22], LFSR258 [22], and Taus88 [23]. Look-up Table Shift Register (LUT-SR [24]) is another LFSR, in which authors propose to turn the use of LUT as a k-bit shift-register using Xilinx SRL32.

Linear Congruential Generators (LCGs): PCG32 [25] is an instance of improved LCG: it post-processes a permutation function (dropping bits using fixed and random rotations) to improve the randomness of the outputs. We can also evoke the MRG32$K3a$ generator [26] (further denotes as MRG32), which is a combined *Multiple Recursive Generator* (MRG), whose period is 2^{191}. KISS124 [27] is another 64-bits (2^{124} period) combined PRNG that calls 3 PRNGs: a 64-bit MWC (*Multiply-With-Carry*), the XOR64, and finally a LCG.

Twisted Generalized Feedback Shift Register (TGFSR): They are based on matrix linear recurrence of n sequence words, each containing w-bits:

$$x^{k+n} = x^{k+m} \oplus (((x^k \,\&\, \underline{S_{MSB}}) \mid (x^{k+1} \,\&\, \overline{S_{LSB}})) \times A), \tag{4}$$

where $0 \leqslant m \leqslant n$. Mersenne Twister [28], Well512 [29], and TT800 [30] generators are special cases of TGFSR, which use BRAM memory to READ/WRITE the tree words.

Xorshift generators: XOR64 [31] and XOR128, with a period of 2^{64} and 2^{128} respectively, are examples of these generators. We can also cite the XOR64* generators [32], which scramble the result of a Xorshift using a 64-bit multiplication, leading to a period of 2^{1024} and 1024-bit state. Finally, XOR128+ [33] proposes a generator of 128 states based on two XOR64 and the sum of the new and previous generated outputs, with a period of 2^{128}.

B. Chaotic PRNG

This section presents a state-of-art of pseudo-chaotic generators which have already been implemented on FPGA.

Differential Chaotic PRNG. Authors in [34] propose a digitized implementation of a nonlinear chaotic oscillator system in Rössler format [20]. They solve the Lörenz hyperchaos with other differential systems as the Chen [35] and Elwakil [36] ones, using an approximated Euler numerical approach. An implementation and optimization of this Lörenz equation are given in [37], in which is used again an Euler approximation with less area but same range of throughput. Finally, authors of [38] have implemented the so-called Oscillator Frequency Dependent Negative Resistors (OFDNR) [39], while using the same Euler approximation.

Chaotic Mapping PRNG. Two different chaotic maps are in general considered: the logistic map [18] and the Hénon map [40]. In [41], the authors deploy the facilities of Matlab DSP System Toolbox software to implement various ranges of logistic map with various lengths, namely from 16 to 64 bits, where the resources are dependent on the precision (from 24 to 53 bits). Then, authors of [38] compare the logistic map [41] results recalled previously with the Hénon ones. Additionally, these authors propose two optimized versions of chaotic logistic map in [42], in which they pipeline the multiplication operations and synchronize them, while adding some delays into each stage, in order to ensure a parallel execution of sequences. Finally, in [43], four different chaotic maps are implemented in FPGA, namely the so-called Bernoulli, Chebychev [44], Tent, and Cubic chaotic maps. The implementation is done with and without FPGA's DSP blocks for the multiplication operations.

Chaotic based Timing Reseeding (CTR). This main concept [19] was first implemented in FPGA [45]. Instead of initializing the chaotic PRNG with a new seed, the seed can be selected by masking the current state x^{t+1} at a specific time. They optimize in [45] the arithmetic operators as multiplication with Carry Lookahead Adder, while the authors of [46] mix the output from the PRNG with an auxiliary generator y^{t+1} to improve statistical tests.

III. Quantifying Hardware Performance of PRNGs

A. Methodology

Previously presented hardware PRNGs are evaluated regarding their randomness, which can be done using statistical tests. The objective of such tests is to evaluate whether the output of a given RNG can be separated or not from a truly random sequence obtained, for instance, by rolling a dice. Such tests are usually grouped in batteries, like NIST [47], and TestU01 [48] ones.

More precisely, the US National Institute of Standard and Technologies has its own battery called NIST SP800 − 22, see [47]. It is constituted by 15 different statistical tests. The *binary* sequence to evaluate must have a fixed length N, such that $10^3 < N < 10^7$. Then, for each statistical test, a set of s sequences is produced by the RNG under consideration, and p-values are obtained. They all need to be larger than 0.0001 to reasonably consider the associated sequences as uniformly distributed and secure according to the NIST opinion.

TestU01, for its part, is currently the most complete and stringent battery of tests for RNGs [48], which groups more than 516 tests inside 7 sub-batteries. In this section, we focus on three major sub-batteries, that encompass 319 tests and which are specific to PRNGs. They are, namely, the *SmallCrush*, *Crush*, and *BigCrush* batteries of tests. *Big Crush* is the most difficult sub-battery in TestU01. This latter uses approximately 2^{38} pseudorandom numbers and applies 160 statistical tests (it computes 160 p-values, that must belong to $[0.001, 0.999]$ in order to pass the considered test).

All the aforementioned linear PRNGs have been implemented on FPGA using Zybo board and Xilinx Vivado tools. The underlying design methodology relies on the use of two high levels of implementation, namely the traditional Register-Transfer Level (RTL) flow and the High-Level Synthesis (HLS [49]). After applying our experiments, we have obtained that almost all PRNGs pass NIST test but only PCG32, MRG32, and XOR64* generators can pass the Big-Crush of TestU01, the most stringent part of this battery, which is coherent with the literature. Obtained test results have shown that a particular and common test called linearity complexity is very frequently failed. This behavior is explained in the next section.

The first next subsections focus on 4 criteria, namely: the linear complexity, the jump complexity [50], the arithmetic operators, and the throughput. Note that the linear and jump complexities are only studied in the linear PRNG case as (1) chaotic PRNGs are not linear, and (2) all these chaotic generators can successfully pass the NIST, which embeds these two complexity tests. Concerning the latter, we further remark that they are currently studied in the literature only for hardware optimization: the novelty of chaotic PRNGs in Table II lies solely in this optimization, and no deeper theoretical study are performed on them. Additionally, they should need to be combined with physical sources to pass the TestU01 battery, which *stricto sensu* transform them in TRNGs. And such TRNGs become too slow to be evaluated with a so stringent battery [11].

B. Linear Complexity

For a given k-length finite binary sequence in $\mathbb{F}_2^k$ issued from a RNG, its linear complexity L_k is defined as the degree of the shortest characteristic polynomial of the LFSR that can generate the same sequence. Intuitively, non linearity is observed when this degree L_k is small. Fig. 1 presents the linear complexity profiles of some PRNGs when applying the

Berlekamp-Massey algorithm. PCG32 and XOR64*, which can pass the whole TestU01, have the linear complexity property. Conversely, other PRNGs like XOR64, WELL512, TT800, and LUT-SR, fail to exhibit such a property.

At this point, we can wonder whether there is any relation between linear complexity and other parameters like the space (resources) used in FPGA. To answer this question, TestU01 computes another parameter named *Jump Computation*.

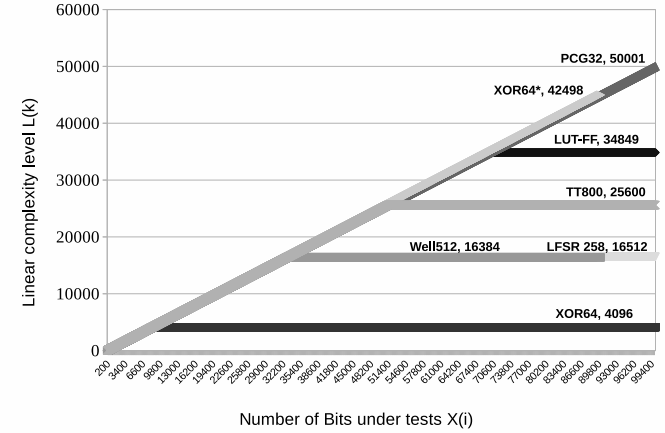

Fig. 1: Linear Complexity profiles $L_k(x_i)$ using Berlekamp-Massey algorithm

C. Jump Complexity

TestU01 battery additionally calculates the number of jumps that occur in the linear complexity for each local subsequence. This number of jumps represents how many bits must be added to the sequence to increase its linear complexity. It has been proven [50] that ideal PRNGs have to perform jumps symmetric to the $k/2$-line as in a perfect linear complexity, with maximum jump heights of $k/4$, and close to $\lfloor (k+1)/2 \rfloor$ for each k-length sequence.

Lets us first illustrate some of these properties using Fig. 2. We compute the linear complexity profiles of the first 32 bits ($k = 32$) of generators LFSR258, XOR64*, and PCG32 using the Berlekamp-Massey algorithm, where the complexity level $L_k(x_i)$ is expressed as follow: $L_1(x_i), L_2(x_i), \ldots, L_{k-1}(x_i)$, with $L_1(x_i) = L(x_1)$ and $L_2(x_i) = L(x_1, x_2)$ ... Each of these PRNGs performs a jumps symmetric to the $k/2$-line as illustrated in Fig. 2. Let us however explain some differences within these jumps. We first notice that the $L_k(x_0, x_1, x_2, x_3)$ is stable for LFSR258 and XOR64$*$. When we add x_4 to compute $L_k(x_0, x_1, x_2, x_3, x_4)$, LFSR258 jumps from 1 to 4 whereas XOR64* is still stable. PCG32, for its part, is stable for less bits and jump by 2 levels in the same interval, where the first jump happens on the x_7 and with more than 8 levels for XOR64*.

Let us consider a stream of random bits $x_i = x_0, x_1, \ldots, x_n$, in which the perfect jump is the difference between two successive linear complexity level L_k applied to x_i and that satisfy $0 < L_k(x_i) - L_k(x_{i-1}) \leq 2$ (*e.g.*, PCG32 has $L_k(x_0, x_1) - L_k(x_0) = (1-1) = 0$ and $L_k(x_0, x_1, x_2) - L_k(x_0, x_1) = (3-1) = 2$ ...).

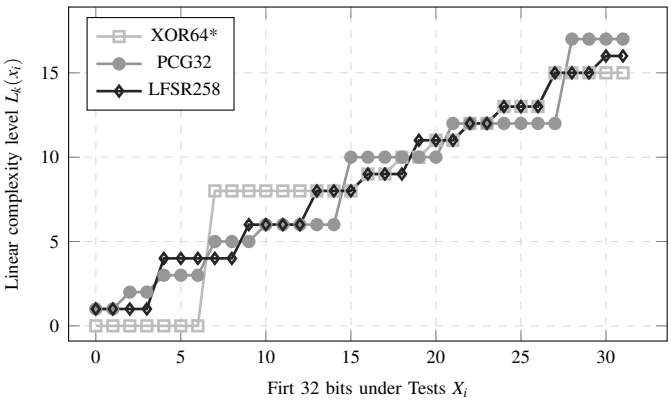

Fig. 2: Jump Computation for 32 bits of random: number of jump < 2 lead to a perfect $\lfloor (k+1)/2 \rfloor$) for k-sequences

Regarding FPGAs, these jumps determine how much resources are required in order to have a perfect complexity profile. For illustration purposes, some of these PRNG jumps have been computed in Fig. 3, by starting from the linear complexity profile L_k illustrated in Fig. 2. More precisely, we computed the jump complexity of 200 linear complexity degrees $L_k(x)$ (k = 200 bits = 6 words), on the one hand for XOR64* and PCG32 that can pass TestU01, and on the other hand for XOR32, TT800, and LUT-SR, who failed this battery.

Let us take XOR64* and LUT-SR as demonstrators of each category from Fig. 3. The aforementioned 200 complexity linear levels illustrate that XOR64* needs a minimum of 52-bits jump to perform a symmetric $k/2$-line (maximum jump heights of $k/4$). However, only 38 jumps are perfect (< 2), where $L_k(x)$ can possibly be repeated between jumps. In addition, we consider stable situations where no jump has occurred (streams of repeated $L(x) = L(x-1)$), where unstable jump is repeated only once. Indeed, we conclude that useful bits are the minimum unique bits, which does not present any form of stability in complexity profile L_k.

We can see that LUT-ST is 4 perfect jumps lower in total than XOR64*. The latter will be propagated for a long period of time, which conducts to a less useful bit contribution for passing linear tests. It is more obvious for XOR32, which confirms the need to another process to face this issue. Indeed, PRNGs that fail to pass TestU01 have the lowest number of useful bits and of perfect jumps, when compared to successful ones. Note that XOR64* uses a multiplication as a kind of output scrambling. PCG32 has the same perspective in its multiplication use, so why it has less useful bits at the end while passing linearity test? To answer this question, we can focus on Fig. 1, which illustrates the existence of stability in linear complexity starting from shorter periods of time.

Some periods can be long, as in the case of PCG32 for instance. When the PRNGs are running, the states space used are constant for any operation. Such property is obvious in 32-bit LCG generators like the PCG32. The latter deploys 32-bit multiplications (64-bits state), but a 36-bits state is required to pass TestU01 with a 32-bit output. This fact means a loss of information that can create a new jump in complexity. This is why PCG32 applies a permutation function to scramble the

weak least significant bits (LSBs) after the multiplication.

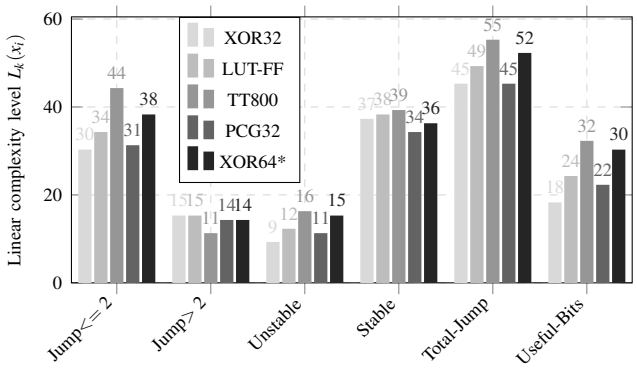

Fig. 3: Jump computation before TestU01 of 200 linear complexity Level: a) Perfect Jump = $[0 < L(k) - L(k-1) \leq 2]$, b) other Jump =$[L(k) - L(k-1) > 2]$, c) Unstable Jump= $[L(k) - L(k-1) \neq L(k)]$, d) stable jump= $[L(k) - L(k-1) = L(k-1)]$, e) Useful bits=$[L(k) - L(k-1) = 1]$

Let us now consider the XOR64* generators, which also use 64-bit multiplications. Their linear complexity is close to the perfect one. The key difference here is the permutation function used for multiplication. In LCG family, this is the main function applied to perform an uniform scrambling operation, whereas in XOR64*, they are deployed to inject bias in randomness.

On the other hand, we can notice the uniform distribution of Mersenne Twister, with an unique maximum perfect jump. But it has the largest stable jumps, that will finally be stable once and for all. This indicates the limitation of tempering unit (similar to XOR32 or LFSR) in terms of performance of transition unit.

At this point, the issue that may be worth mentioning is that most of the chaotic PRNGs reviewed in this paper are not answering the targeted question. That is, how can we inject disorder in a deterministic digital system, in order to respect on such finite state machines the mathematical definitions of chaos provided by Devaney [8] and Li-Yorke [9]? Indeed, passing the NIST, which is the most usual way to evaluate a PRNG, can be put into default: some generators of pour quality can successfully pass these tests.

To solve this issue, we will see in Section VI-B the usefulness of chaotic iterations as a post-processing replacing this tempering unit (see CIPRNG-MCMD).

D. Experimental Results

The aforementioned PRNGs have been implemented in our Zynq platform (Fig. 4). Both categories (linear and chaotic PRNGs) are analyzed in terms of hardware resources and throughput/latency. The analysis is based on Xilinx Vivado v16.3 tools with the default configuration and without any optimization. Additionally, the FPGA target was Zybo Zynq-7000 ARM/FPGA SoC Trainer Board from digilent (125Mhz).

Hardware resources: The size and performance of the PRNG depend on both the word length (addressing the LUT increases the table exponentially) and their binary representations, regarding *dynamic range* (DR) and precision ($DR_{fxpt} = r^n - 1$ where r is in binary format (Radix-2) and n is the number of digits in fixed-point precision). The aforementioned PRNGs in this section have a fixed DR and internal space of 32 or 64 bits.

Reviewing Table I, LUT-SR, Taus88, and XOR64 require the lowest amount of area resource. Conversely, combined PRNGs like KISS124 and MRG32, LCG, and TGFSR families have large area consumption due to their implementation of arithmetic multiplication with complex logic as DSP. let us take for instance the KISS127 of $DR = 2^{64}$ as an example, which is implemented with DA (KISS-DA) or DSP blocks (KISS-DSP). It is clear from Table I that disabling DSP will induce a huge area extension and a drop in frequency while presenting the same latency (even though DSP blocks can be a convenient alternative and an additionnal resource for ASIC applications). To sum up, chaotic PRNGs have approximately the same use of hardware resources than linear PRNGs (logistic map implementations have proven to be the lowest among them).

Throughput and Latency: Let us recall two proprieties based on the frequency, which are namely the latency and the throughput. Latency is the number of iterations required to compute a new output from a given input. The throughput, for its part, is the number of iterations needed to produce new output or to consume a new input. Note that the throughput delay can be equal to the latency, which lead us to use the throughput/latency value to estimate the real throughput of the PRNG. In the other hand, the HLS flow schedules automatically cycle-by-cycle the algorithms as a finite state machine. Therefore, the synthesis tool adds one cycle to process the input and a second one to generate the final output.

Latency and throughput in the RTL and HLS flows can be formalized as follows.

$$\begin{array}{rcl} Design\ Latency & : & [Delay\ from\ Input\ to\ Output] \\ OutputLatency & : & [Delay\ for\ each\ Output] \\ Throughput & : & \left[\dfrac{Output\ Size}{Output\ Latency}\right] \end{array} \quad (5)$$

On the one hand, for 32 bit linear generators (resp. for 64 bits ones), Taus88 and LUT-SR with LFSR113 (resp. XOR64 and LFSR258) have the largest throughput performance, while for chaotic PRNGs, Bernoulli [43] and logistic map using DSP [42] with CPRNG based timing reseeding [45] have the larger throughput. On the other hand, two implementations of Mersenne Twister generators have been designed with and without the seed, respectively denoted as MT_WS and MT_NS. We have remarked that, when considering the seed as a function, frequency is reduced to less than 200MHz compared to the case where it is not present. Therefore, to increase performances, most PRNGs do not include the seed internally (and a software is used).

To put it in a nutshell, if we take the ratio of area/throughput as main criterion, we are balancing between high performance as XOR64, LFSR113 for linear PRNG (resp., Bernoulli [43] and logistic map [42] for chaotic PRNG) and the ability to pass statistical tests (PCG32 and XOR64*), which is not surprising. Another result is that combining PRNGs leads to

a performance decrease in hardware level. Such combinations do not take into account the Chaotic Iterations post-processing, which appears as promising [21]. Indeed, chaotic PRNGs outperform the linear ones in terms of throughput performance, but they are not able to pass the TestU01 statistical battery (see Table II). This lack is the reason to be of this contribution: we propose a hardware chaos-based post-processing module to improve the randomness of linear PRNGs. By doing so, and conversely to the other chaotic PRNG, these post-processed generators behave chaotically while succeeding in passing the TestU01 test. Effects of such a post-processing on performances at hardware level are detailed in the following sections.

IV. Chaotic Iterations: the theory

The generators (or, more exactly, the post-treatment over existing generators) we propose in this document are theoretically formalized by the so-called chaotic iterations (CIs), and their performances are directly related to the topological properties of these CIs. The latter are investigated in this section, while the relations with our generators are detailed in the next one.

Let f be a map from $\mathbb{B}^{\mathsf{N}}$ to itself, and let us introduce the following functions:

- $\sigma : \mathcal{P}(\llbracket 1,\mathsf{N}\rrbracket)^{\mathbb{N}} \longrightarrow \mathcal{P}(\llbracket 1,\mathsf{N}\rrbracket)^{\mathbb{N}}, (S^t)_{t\in\mathbb{N}} \mapsto (S^{t+1})_{t\in\mathbb{N}}$.
- the initial function, defined by $i : \mathcal{P}(\llbracket 1,\mathsf{N}\rrbracket)^{\mathbb{N}} \longrightarrow \mathcal{P}(\llbracket 1,\mathsf{N}\rrbracket), (S^t)_{t\in\mathbb{N}} \mapsto S^0$
- and $F_f : \mathcal{P}(\llbracket 1;\mathsf{N}\rrbracket) \times \{0,1\}^{\mathsf{N}} \longrightarrow \{0,1\}^{\mathsf{N}}$,

$$(P,E) \longmapsto \left(E_j.\delta(j,P) + f(E)_j.\overline{\delta(j,P)}\right)_{j\in\llbracket 1;\mathsf{N}\rrbracket}$$

where $\mathbb{B} = \{0,1\}$, $\mathcal{P}(X)$ is the set of subsets of X, $X^{\mathbb{N}}$ is the set of sequences whose elements belong to X, and $\delta(j,P) = 1$ if $j \in P$, else $\delta(j,P) = 0$. For $\mathsf{N} \in \mathbb{N}^*$, let $\mathcal{X} = \mathcal{P}\left(\llbracket 1;\mathsf{N}\rrbracket^{\mathbb{N}}\right) \times \{0,1\}^{\mathsf{N}}$, with the distance between two points $X = (S,E), Y = (\check{S},\check{E})$ as follows:

$$d(X,Y) = d_e(E,\check{E}) + d_s(S,\check{S}), \tag{6}$$

where

$$\begin{cases} d_e(E,\check{E}) &= \displaystyle\sum_{k=1}^{\mathsf{N}} \delta(E_k,\check{E}_k) \text{ is the Hamming distance,} \\ d_s(S,\check{S}) &= \dfrac{9}{\mathsf{N}} \displaystyle\sum_{k=1}^{\infty} \dfrac{|S^k \Delta S^k|}{10^k}. \end{cases} \tag{7}$$

where $|X|$ is the cardinality of a set X and $A\Delta B$ is for the symmetric difference, defined for sets A, B as $A\Delta B = (A \setminus B) \cup (B \setminus A)$.

Consider $G_f(S,E) = (\sigma(S), F_f(i(S),E))$. Chaotic iterations are defined by the following discrete dynamical system [51]:

$$\begin{cases} X^0 = (S,x^0) \in \mathcal{X}, \\ \forall t \in \mathbb{N}, X^{t+1} = G_f(X^t). \end{cases} \tag{8}$$

The asynchronous iteration graph associated with f is the directed graph $\Gamma(f)$ defined by: the set of vertices is $\mathbb{B}^{\mathsf{N}}$; for all $x \in \mathbb{B}^{\mathsf{N}}$ and $i \subset \llbracket 1;\mathsf{N}\rrbracket$, the graph $\Gamma(f)$ contains an arc from x to $F_f(i,x)$ labeled by subset i. We have previously established that [52], [53]:

Proposition 1 *If $\Gamma(f)$ is strongly connected, then G_f is strongly transitive: for any couple $(x,y) \in \mathcal{X}$ and for all neighborhood V of x, there is $z \in V$ and $n \in \mathbb{N}$ such that $f^n(z) = y$.*

Thus it is chaotic according to Devaney [8], i.e., it is

1) Transitive: *For each couple of open sets $A,B \subset \mathcal{X}$, there exists $k \in \mathbb{N}$ such that $f^{(k)}(A) \cap B \neq \varnothing$.*
2) Regular: *Periodic points are dense in $\mathcal{X}$.*
3) Sensible to the initial conditions: $\exists \varepsilon > 0$, $\forall x \in \mathcal{X}$, $\exists y \in \mathcal{X}$, $\exists n > 0 \in \mathbb{N}$, *such that* $d(x,y) < \varepsilon$ *and* $d(f^{(n)}(x), f^{(n)}(y)) \geqslant \varepsilon$.

We start to further investigate the disordered behavior of chaotic iterations, on which our generator is based, with the following result:

Proposition 2 *Let us consider f such that the graph $\Gamma(f)$ is strongly connected. Then, for all open ball $\mathcal{B}$ of $\mathcal{X}$, we can find an iteration $n \in \mathbb{N}$ such that $G_f^n(\mathcal{B}) = \mathcal{X}$.*

Proof Given $x \in \mathcal{X}$ and $r > 0$, let us recall that the open ball $\mathcal{B}(x,r)$ is the set $\{y \in \mathcal{X} \mid d(x,y) < r\}$. For these x and r, we intend to show that $\exists N_0 \in \mathbb{N}$ such that $G_f^{N_0}(\mathcal{B}(x,r)) = \mathcal{X}$. Without limitation, we can assume that $r < 1$, because if $r' < min(1,r)$ satisfies $\exists N_0 \in \mathbb{N}$ s.t. $G_f^{N_0}(\mathcal{B}(x,r')) = \mathcal{X}$, then as $\mathcal{B}(x,r') \subset \mathcal{B}(x,r)$, we can a fortiori deduce that $G_f^{N_0}(\mathcal{B}(x,r)) = \mathcal{X}$.

Consider the point $y^{(0)} = (\{1\},\{1\},\{1\},\ldots),(0,\ldots,0))$ of $\mathcal{X}$. As the iteration graph is strongly connected, then G_f is strongly transitive. So there is a point $x^{(0)}$ in the neighborhood $\mathcal{B}(x,r)$ of x and an integer $n^{(0)}$ such that $G_f^{n^{(0)}}(x^{(0)}) = y^{(0)}$. This point $x^{(0)}$ is necessarily of the following form:

- Being inside $\mathcal{B}(x,r)$ with $r < 1$, its second coordinate (the Boolean vector) must be the same than x, due to the Hamming distance in d. In other words, $x_2^{(0)} = x_2$.
- Let $n_0 = -\lfloor \log_{10}(r) \rfloor$. Having regard to the definition of d and as $x^{(0)} \in \mathcal{B}(x,r)$, we necessarily have an equality between the n_0 first terms of the sequence x_1 and the n_0 first terms of the sequence $x_1^{(0)}$.
- As $G_f^{n^{(0)}}\left(x^{(0)}\right) = y^{(0)}$, it is a necessity that after $n^{(0)}$ shifts of the sequence of $x^{(0)}$, we obtain the sequence $(\{1\},\{1\},\{1\},\ldots)$ of $y^{(0)}$.
- Finally, terms of the sequence of $x^{(0)}$ between positions n_0+1 and $n^{(0)}$ are the ones required for f to transform the Boolean vector of $x^{(0)}$ to the one of $y^{(0)}$ (this is the path to follow in Γ_f, to reach $y_2^{(0)}$ starting from $x_2^{(0)}$).

Let us now consider a point $Y^{(0)}$ of the form:

- its Boolean vector $Y_2^{(0)}$ is equal to $y_2^{(0)}$;
- its sequence $Y_1^{(0)}$ is of any kind.

Then the point $X^{(0)}$ defined by:

1) $X_2^{(0)} = x_2^{(0)}$: same Boolean vector than $x^{(0)}$;
2) $\forall k \in \llbracket 0,n_0\rrbracket, X_{1,k}^{(0)} = x_{1,k}^{(0)}$: the n_0+1 first terms of subset sequences of $X^{(0)}$ and $x^{(0)}$ are equal;
3) $\forall k \in \llbracket n_0+1, n^{(0)}\rrbracket, X_{1,k}^{(0)} = x_{1,k}^{(0)}$: idem for the $n^{(0)} - n_0$ following ones;
4) $\forall k > n^{(0)}, X_{1,k}^{(0)} = Y_{1,k-n^{(0)}}^{(0)}$: the last terms in sequence of $X^{(0)}$ are the whole terms of sequence of $Y^{(0)}$;

TABLE I: FPGA implementation of linear PRNG in term of: Area, Speed, and Statistical tests

		Linear PRNG															
Familly		LFSR				xorshift				TGFRS				LCG			
PRNG		LFSR113	Taus88	LFSR258	LUT-SR	XOR128	XOR64	XOR128+	XOR64*	MT_WS[b]	MT_NS[c]	Well512	TT800	KISS-DA	KISS-DSP	MRG[a]	PCG32[a]
Output Rang (n)		32	32	64	32	32	64	64	64	32	32	32	32	64	64	64	32
AREA	LUT	79	68	171	64	36	55	131	298	523	184	94	184	271	2038	1055	345
	FF	162	130	386	64	194	130	194	390	120	179	108	483	746	1277	1359	418
	RAM	0	0	0	0	0	0	0	10	2	2	0	6	0	0	0	10
	DSP	0	0	0	0	0	0	0	4	3	0	2	2	7	0	8	0
	Total Area (LUT+FF)*8	2072	1584	4456	576	1840	1480	2600	5504	5144	3272	1616	5336	8136	26520	19312	6104
SPEED	Frequences (Mhz)	443,26	448,63	396,98	609	429,36	457,45	250,81	231	118	462	213	169	154	112	175	179
	Design Latency	2	2	2	2	2	2	2	21	3	2	5	4	9	9	14	20
	Output Latency	1	1	1	1	1	1	1	21	1	1	5	4	9	9	14	20
	Throughput/Latency (Gbps)	14,18	14,35	12,70	19.5	13,73	14,63	8,02	0.7	3.8	13.2	1.3	1.3	1.1	0.8	0.8	0,286
TESTS	NIST (16 Tests)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	NO	PASS	PASS	PASS	PASS
	TestU01 (319 Tests)	NO	NO	NO	NO	NO	NO	NO	PASS	NO	NO	NO	NO	NO	NO	PASS	PASS

[a] HLS Implementation, [b] MT_WS: Mersenne Twister with Seed, [c] MT_NS: Mersenne Twister without Seed.

TABLE II: FPGA implementation of chaotic PRNG in term of: Area, Speed, and Statistical tests

		Chaotic PRNG						
		[41]	[38]	[42]	[45]	[46]	[34]	[43]
PRNG		Logistic	Logistic-Hénon-FNDR[d]	Logistic	Timing Reseeding	Timing Reseeding	LRZ[f]- Chen - ELW[d]	B[c]- CH[f]- T[g]
Output Rang (n)		32	32	32	32	32	32	32
AREA	LUT	66	48-539-208	313	*** [e]	*** [e]	287-284-265	*** [e]
	FF	32	32-32-96	842	*** [e]	*** [e]	96-97-97	*** [e]
	DSP	4	*** [e]	16	*** [e]	*** [e]	8-0-0	0
	Total Area (LUT+FF)*8	784	640-4568-4568	9240	*** [e]	11903	3064-13968	*** [e]
SPEED	Frequency (Mhz)	76.1	151.1-58.2-183	233	200	200	53.53-122-126.7	265.9-118.7-111.8
	Design Latency	*** [e]	*** [e]	8 to 16	*** [e]	*** [e]	*** [e]	*** [e]
	Output Latency	1	1	1	1	1	1	1
	Throughput/Latency (Gbps)	2.435	4.835-1.862-5.856	7.5	6.4	6.4	1.71-3.9-4.06	8.5-3.798-3.577
TESTS	NIST (16 Tests)	PASS	PASS	PASS	PASS	PASS	NO	PASS
	TestU01 (312 Tests)	NO	NO	NO	NO	NO	NO	NO

[a] ***: No Information, [b] FNDR: Frequency Dependent Negative Resistors. [c] LRZ: Lörenz. [d] EWL: Elwakil. [e] B: Bernoulli, [f] CH: Chebychev, [g] T: Tent.

is such that:

- $X^{(0)} \in \mathcal{B}(x,r)$, due to the two first items above;
- the Boolean vector of $G_f^{n^{(0)}}\left(X^{(0)}\right)$ is the one of $Y^{(0)}$, due to the third item;
- the sequence of $G_f^{n^{(0)}}\left(X^{(0)}\right)$, which is the one of $X^{(0)}$ after $n^{(0)}$ shifts, is too the sequence of $Y^{(0)}$, due to the forth item.

In other words, for each $Y^{(0)}$ in $\mathcal{X}$ that has $(0,0,\ldots,0)$ as Boolean vector, we have found a point $X^{(0)}$ in $\mathcal{B}(x,r)$ and $n^{(0)} \in \mathbb{N}$ such that $G_f^{n^{(0)}}\left(X^{(0)}\right) = Y^{(0)}$.

We now proceed similarly for points having $(0,\ldots,0,1)$ as Boolean vector. Consider now the point $y^{(1)} = (\{\{1\},\{1\},\{1\},\ldots),(0,\ldots,0,1)) \in \mathcal{X}$. For the same reasons than previously, there exists a point $x^{(1)}$ of $\mathcal{B}(x,r)$ and an integer $n^{(1)}$ such that $G_f^{n^{(1)}}(x^{(1)}) = y^{(1)}$. Let us now consider a point $Y^{(1)}$ of the form:

- its Boolean vector $Y_2^{(1)}$ is equal to $y_2^{(1)}$;
- its sequence $Y_1^{(1)}$ is of any kind.

Then the point $X^{(1)}$ defined by:

1) $X_2^{(1)} = x_2^{(1)}$;
2) $\forall k \in [\![0,n_1]\!], X_{1,k}^{(1)} = x_{1,k}^{(1)}$;
3) $\forall k \in [\![n_1+1,n^{(1)}]\!], X_{1,k}^{(1)} = x_{1,k}^{(1)}$;
4) $\forall k > n^{(1)}, X_{1,k}^{(1)} = Y_{1,k-n^{(1)}}^{(1)}$;

is such that $X^{(1)} \in \mathcal{B}(x,r)$ and $G_f^{n^{(1)}}\left(X^{(1)}\right) = Y^{(1)}$: any point of $\mathcal{X}$ having $(0,\ldots,0,1)$ as Boolean vector can be reached from $\mathcal{B}(x,r)$ with $n^{(1)}$ iterations of G_f.

This process can be extended accordingly until the point $y^{(2^{\mathsf{N}}-1)} = (\{\{1\},\{1\},\{1\},\ldots),(1,\ldots,1,1))$, which leads to the definition of $n^{(2^{\mathsf{N}}-1)}$, of $x^{(2^{\mathsf{N}}-1)}$, of $Y^{(2^{\mathsf{N}}-1)}$, and finally of $X^{(2^{\mathsf{N}}-1)}$.

At this stage, we can claim that, for all y of $\mathcal{X}$, it is possible to find $x' \in \mathcal{B}(x,r)$ and a certain integer $N \in \{n^{(0)},\ldots,n^{(2^{\mathsf{N}}-1)}\}$ such that $G_f^N(x') = y$. The last issue to solve is that the iteration number N depends on the Boolean vector y_2, which should not be the case.

Let us consider $N_0 = max(\{n^{(k)}, k = 0,..,2^{\mathsf{N}}-1\})$. In each sequence of subsets $X_1^{(k)}, k \in [\![0,2^{\mathsf{N}}-1]\!]$, it is possible to incorporate $N_0 - k$ times the empty set $\varnothing$ between terms $X_{1,n^{(k)}}^{(k)}$ and $X_{1,N_0}^{(k)}$, in such a way that:

$$Y^{(k)} = G_f^{n^{(k)}}\left(X^{(k)}\right) = G_f^{n^{(k)}+1}\left(X^{(k)}\right) = \ldots = G_f^{N_0}\left(X^{(k)}\right),$$

which is equivalent to be on a treadmill once reaching the target $Y^{(k)}$ and until having iterated N_0 times. Thanks to that, for all $y \in \mathcal{X}$, it is possible to find $x' \in \mathcal{B}(x,r)$ such that $G_f^{N_0}(x') = y$, which is the expected result.

Therefore, however small the starting open ball, we finish to reach the whole $\mathcal{X}$ space by iterating G_f. Using this result, we can deduce the following proposition related to chaos.

Proposition 3 *General chaotic iterations G_f are topologically mixing: for all couple of nonempty open sets U and V, there is $n_0 \in \mathbb{N}$ such that $\forall n \geqslant n_0$, $G_f^n(U) \cap V \neq \varnothing$.*

PROOF Let us consider U and V, two disjoint nonempty open sets of $\mathcal{X}$. U being a nonempty open set, we can find $x \in \mathcal{X}$ and $r > 0$ such that $\mathcal{B}(x,r) \subset U$. Due to Proposition 2, $\exists n_0$ s.t. $G_f^{n_0}(\mathcal{B}(x,r)) = \mathcal{X}$. As $\mathcal{B}(x,r) \subset U$, we have too $G_f^{n_0}(U) = \mathcal{X}$, and so $G_f^{n_0}(U) \cap V \neq \varnothing$. Let us consider $Y \in G_f^{n_0}(U) \cap V$. It exists $X^{(0)} \in U$ such that $G_f^{n_0}(X^{(0)}) = Y$. The point $X^{(1)}$ defined by:

- $X_2^{(1)} = X_2^{(0)}$;
- $\forall k \leqslant n_0, X_{1,k}^{(1)} = X_{1,k}^{(0)}$: the two sequences start by the same terms;
- $X_{1,n_0+1}^{(1)} = \varnothing$: we insert an empty set at position n_0+1 in sequence $X_1^{(1)}$;
- $\forall k > n_0, X_{1,k}^{(1)} = X_{1,k-1}^{(0)}$;

is such that $G_f^{n_0+1}(X^{(1)}) = Y$, and so $G_f^{n_0+1}(U) \cap V \neq \varnothing$. Similarly, by incorporating l empty sets between positions n_0+1 and n_0+l inside the sequence of $X^{(0)}$, we are able to define a point $X^{(l)}$, which is such that $G_f^{n_0+l}(X^{(l)}) = Y$, proving so $G_f^{n_0+l}(U) \cap V \neq \varnothing$. This inequality being valid for all $l > 0$, we can deduce the topological mixing of G_f.

Proposition 4 *When considering the vectorial negation for f, the general chaotic iterations satisfy the Knudsen's definition of chaos [54]: they are sensible to the initial condition and they have a dense orbit.*

PROOF The sensibility to the initial condition of G_f has already been stated in [52]. We are then left to construct a point $x \in \mathcal{X}$ such that the set $\{G_f^n(x) \mid n \in \mathbb{B}\}$ is dense in $\mathcal{X}$: iterations of $G_f^n(x)$ must be as close as possible to any point $y \in \mathcal{X}$.

Let us denote by $s_0, s_1, \ldots, s_{2^{\mathsf{N}}-1}$ the list of each subset of $[\![1,\mathsf{N}]\!]$: $s_0 = \varnothing$, $s_1 = \{\mathsf{N}\}$, $s_2 = \{\mathsf{N}-1\}$, $s_3 = \{\mathsf{N}-1, \mathsf{N}\}$, ..., $s_{2^{\mathsf{N}}-1} = \{1,2,...,\mathsf{N}\}$. Let us now consider a point $y \in \mathcal{X}$. Its Boolean vector y_2 can be associated to a given s_k, namely the subset of $[\![1,\mathsf{N}]\!]$ that contains the coordinates of 1's in y_2. The first term $y_{1,0}$ of sequence y_1, for its part, is a given $s_{k'}$, while the second term $y_{1,1}$ is too a given $s_{k''}$, with $k, k', k'' \in [\![0, 2^{\mathsf{N}}-1]\!]$.

Let us now remark that, when iterating G_f on the point $((s_k, s_{k'}, s_{k'}, s_k, ...), (0,0,\ldots,0))$, with f the vectorial negation:

- We start on the Boolean vector $(0,0,\ldots,0)$;
- As s_k indicates the 1's in vector y and we use the vectorial negation, we thus have, after one iteration of G_f:
  - the Boolean vector $(0,0,\ldots,0)$ is changed in y_2;
  - the sequence is shifted of one position, so it now starts by $(s_{k'}, s_{k'}, s_k, ...)$.

  Having the same Boolean vector and the same first term in the sequence, we are thus at a distance lower than 10^{-1} to y after one iterate.
- Iterating G_f another time switches the binary digits in positions $s_{k'}$ in the Boolean vector, while shifting the sequence so that it becomes $(s_{k'}, s_k, ...)$.
- Iterating twice G_f will operate a second time the negation on Boolean digits at position $s_{k'}$ and s_k,

and so the Boolean state is $(0,0,\ldots,0)$ again after these 4 iterations. To sum up, iterating four times starting from $((s_k, s_{k'}, s_{k'}, s_k, ...), (0,0,\ldots,0))$ will first move the system at a distance 10^{-1} to y, and then come back to $(0,0,\ldots,0)$ after shifting 4 times the sequence.

Let us now consider the point:

$$\begin{aligned}(\,&(s_0, s_0, s_0, s_0, \quad s_0, s_1, s_1, s_0, \ldots,\\ &s_0, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_0, \quad s_1, s_0, s_0, s_1, \quad s_1, s_1, s_1, s_1, \ldots,\\ &s_1, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_1, \ldots, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, \ldots),\\ &(0,0,\ldots,0))\end{aligned}$$

By iterating G_f on it, we will be at one time at 10^{-1} of any point of $\mathcal{X}$, while recovering the null Boolean vector at each 4 iterates. Continuing the process with patterns of length 6, 8, 10, etc., will define a unique point x whose iterates are as close as possible to any point of $\mathcal{X}$, leading to a dense orbit.

V. CHAOTIC ITERATIONS AS PRNGS POST-PROCESSING

A. CIPRNG Multi-Cycle

As described in the previous section, the general chaotic iterations receives an integer sequence as input (and the first internal state, a binary vector), and it produces a sequence of binary vectors. In other words, chaotic iterations translate a sequence in another sequence. This is a way to obtain a new pseudorandom number generator from a former one. Both the kind of inputted generator and the iteration function f are parameters of this post-treatment, while the first vector x^0 and the first term S^0 act as seeds. As the latter are the initial condition of the discrete dynamical system of Eq. (8), to choose f such that this dynamical system behaves chaotically seems to be interesting in a pseudorandom generation context. In other words, we hope that chaos bring by the iteration function will lead to a more disordered output $(x^t)_{t\in\mathbb{N}}$ than the input $(S^t)_{t\in\mathbb{N}}$. Even if there is, *stricto sensu*, no theoretical relation between randomness and chaos (similarly, there is no relation between security and chaos), numerous simulations have illustrated [53] that, due to chaos, the output sequence is in general more random than the input one, according to the number of statistical tests they can pass.

Such chaotic iterations based post-treatment over existing PRNGs can be designed as follows. As we need to generate a sequence $(S^t)_{t\in\mathbb{N}}$ of subsets of $[\![1,\mathsf{N}]\!]$, we can consider two input generators, both producing numbers in $[\![1,\mathsf{N}]\!]$. The aim of the first generator is to provide, at each iterate t, the size of the subset S^t, while the second generator produces the content of S^t. This way to post-operate over the input generators is what we called CIPRNG Multi-Cycles.

The basic design procedure of this latter is summarized in Algorithm 1. The internal state is x, the output state is r. The internal values a and b are computed by the two input PRNGs. Lastly, the value $g_1(a)$ is an integer defined as in Eq. (9). To do so, a sequence d_s $(= (d^1, d^2, \ldots, d^N) \in \{0,1\}^N)$ called a *irregular decimation* is provided for the second generator b, which insures that we do not have two successive permutations of the same bit within a given iteration. This latter will update the i-th bit of b at iteration m^t, and by using the strategy, if and only if $d^{b^i} \neq 1$, otherwise it is discarded. For instance, let us

consider the input $x = \{x^1, x^2, x^3, x^4\}$, the number of iterations $m^t = \{4,3,4,1\}$, and $b = \{2,3,1,1,\mathbf{4,4},3,1,2,3,\mathbf{2,2,4,4},1,2\}$. Due to the first value of m^t, we have to iterate 4 times b, and then 3 and 4 times. We then have to operate the decimation on b so that we will not modify twice a same component in a given iteration: this leads to the strategy $S = \{\{2,3,1,4\}\{4,3,1\}\{2,3,2,4\}\dots\}$ extracted from b. As can be seen, the duplicated entry 2,2,4,4 has been decimated to 2,4, while it is not the case for the first 4,4, as according to m^0 this duplication falls between two iterates. This constraint explains the general form of m^t provided in Eq. (9).

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

Algorithm 1 CIPRNG-MC proposal. At each iteration, only the S^t-th component of state x^t is updated, as follows: $x_i^{t+1} = x_i^t$ if $i \neq S^t$, else $x_i^{t+1} = \overline{x_i^t}$.

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

procedure CIPRNG-MC(x, r)
  for $i = 0,\dots,N$ **do**
    $d_i \leftarrow 0$
  $a \leftarrow PRNG_1()$
  $m \leftarrow g(a)$
  while $i = 0,\dots,m$ **do**
    $b \leftarrow PRNG2()$ *mod* N
    $S \leftarrow b$
    if $d_S = 0$ **then**
      $x_S \leftarrow \overline{x_S}$
      $d_S \leftarrow 1$
    else
      $m \leftarrow m + 1$
  return ($r \leftarrow x$)

Such CIPRNG-MC, which is a sub-category of our CIPRNG post-treatment, can be summarized as follows [52]. x^0 is the initial Boolean vector of size N, and $(S^t)_{t\in\mathbb{N}}$ is the sequence resulted from the irregular decimation of (m,b), as described previously. We suppose that this latter produces numbers belonging into $[\![0, 2^{\mathsf{N}} - 1]\!]$. Operating G_f with the vectorial negation on such sequences can be directly rewritten as follows [52]:

$$x^{t+1} = x^t \otimes S^t, \tag{10}$$

where S^t is expressed in the base-2 numeral system as a binary vector of size N, while $\otimes$ is the bitwise XOR operation over binary vectors. In other words, CIPRNG-MC is equal to the chaotic iterations with the vectorial negation and the decimation S of the two inputted generators m and b. Note that, most of the time, we need to iterate the second generator more than the cardinality of S^t, as we can obtain twice the same number. This weakness in the decimation process is at the origin of our second proposal, namely the CIPRNG-XOR.

B. CIPRNG-XOR

Conversely to CIPRNG Multi-Cycles, this CIPRNG-XOR only needs one inputted generator. It operates on it using the vectorial negation. We have established in [55] that G_f satisfies various properties of chaos with such iteration function, one of them being the notion of chaos according to Devaney, which is studied in this article. Another interesting property proven in the aforementioned article is that, if the inputted generator is cryptographically secure, then the resulted CIPRNG-XOR generator, obtained after post-processing, still present this property. Once again, such CIPRNG-XOR is a sub-category of our CIPRNG post-treatment. If we consider again that x^0 is the initial Boolean vector of size N, and $(S^t)_{t\in\mathbb{N}}$ is the sequence generated by the inputted generator (producing numbers belonging into $[\![0, 2^{\mathsf{N}} - 1]\!]$), then operating G_f with the vectorial negation on such sequences can obviously be rewritten as Eq. (10) [52]. We found again a direct equivalence between chaotic iterations using the vectorial negation and this CIPRNG-XOR. The main requirement is to prevent the machine from working in silos, by taking at each iterate a new input from the outside world (an entropy source like a physical white noise or some digits in the CPU temperature, can be considered for instance). By doing so, the finite state machine does not necessarily enter into a loop: a same state can be visited twice, but with two completely different future evolution, depending on the inputs the machine receives.

Algorithm 2 presents details of this approach where 3 PRNGs are embedded to compute the strategy. In the updated version we implemented, two inputted PRNGs of 64 bits denoted by x_i and y_i are used for defining the chaotic strategy S. Furthermore, we added a third inputted set generator z_i of 32 bits for more complexity. The z_i generator will pick randomly a subset of the inputs at each iteration as described in Equation 10, in which only the $\log(\log(n))$ least significant bits (in this case, 3 bits) are used.

Algorithm 2 CIPRNG-XOR proposal: it randomly picks a subset of the inputs at each iteration, whose index is contained in the first term of the strategy

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

procedure CIPRNG-MC(x, r)
  $u_i \leftarrow PRNG1$,
  $y_i \leftarrow PRNG2$,
  $z_i \leftarrow PRNG3$
  if $(z_i \,\&\, 1) \neq 0$ **then**
    $x \leftarrow x \otimes (u_i \,\&\, 0x0FFFFFFFF)$
  if $(z_i \,\&\, 2) \neq 0$ **then**
    $x \leftarrow x \otimes (u_i \gg 32)$
  if $(z_i \,\&\, 4) \neq 0$ **then**
    $x \leftarrow x \otimes (y_i \,\&\, 0x0FFFFFFFF)$
  $r \leftarrow x \otimes (y_i \gg 32)$
  return r

VI. FPGA Implementation Based on Zynq Platform

A. General Presentation

Xilinx Zynq-7000 Extensible Processing Platform (EPP) [15] is a silicon system on chip (SoC) for FPGAs,

which has been proposed by Xilinx. This SoC deploys the latest technologies of ARM processors with a large set of peripherals (DDR, PCI, etc.). This latter is defined as Peripheral System (PS), which is a sub-system with ARM. The full FPGA is the Programmable Logic (PL) that is connected with PS through an AXI bus interface.

Fig. 4 shows the detailed hardware architecture of our system used to integrate and test CIPRNGs. The AXI-PRNG interconnect can handle many PRNGs/CIPRNGs at the same time and it activates the one that is currently tested. This interconnect component is re-configurable using the firmware, which deploys two GPIO IPs for this task. GPIO-0 is used to select one PRNG at a time, and GPIO-1 is used for the data burst size of the PRNG. For instance, all PRNGs implemented in HLS or RTL, including the AXI-PRNG interconnect, are AXI Stream Interface, while the CPU is Memory-Mapped Interface. Additionally to CPU, the AXI-DMA engines, which oversee the data transaction between the slave and master IPs, deploy the receiver channel Slave to Memory Map (S2MM) connected to a slave port, and the transmitter channel Memory-Map to Slave (MM2S) connected with the master. Final outputs are displayed in an external terminal via the UART protocol.

We have used the Zybo board ($XC7Z010-1CLG400C$) as a prototype kit for experiments such that the clock is configured at 125Mhz. The total space of the logic part (PL) on Zybo board is: 2,982 LUT (19%), 4,071 FF (11%), 7 DSPs, and 3 memories respectively.

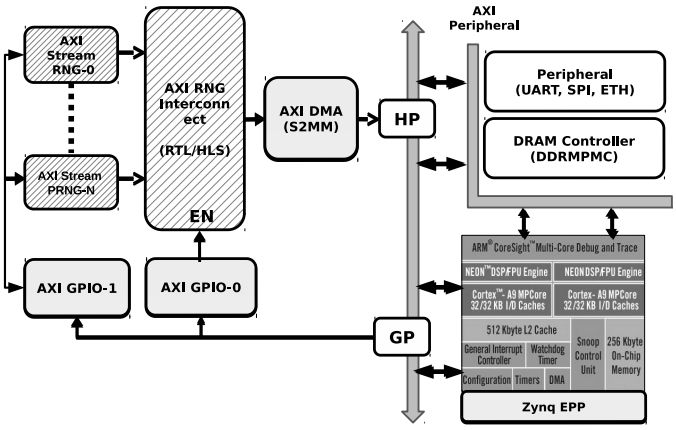

Fig. 4: PRNG platform based on Zynq FPGA

B. Global Comparison

As stated previously, the objective is to determine the performance of CIPRNG implementation in terms of area (space) and throughput (speed). The Xilinx tool calculates all resources used in FPGA as logic gates, LUT, Flip-Flop (register), additionally to DSP and memory blocks. Despite the fact that Xilinx calculates the area by counting slices (1 Slice = 4× LUT + 2× FF+interconnection), it uses the same LUT of 6-inputs for all its technologies (Virtex5, Virtex6, Virtex7, and Zynq). Hence, for our area comparison, we only calculated LUT and FF as $[(LUT+FF)\times 8]$, since DSPs and RAM memories are hard blocks that can mostly affect time performances. The brute throughput and the rate throughput over the latency are calculated as in Equation (5). This second scalar value is a complementary indicator that provides an accurate speed information about the generators.

Regarding CIs based post-processing, we tested more than 275 versions of CIPRNG-XOR on our Mésocentre supercomputer facilities (170 were able to pass TestU01) and 169 of CIPRNG-MC/MCMD (93 pass the TestU01). Only are recalled hereafter those who pass the recommended statistical TestU01 battery. To reach a fair comparison, we disabled the use of DSP blocs for linear PRNGs. Additionally, having the ASIC implementations in mind, we excluded each CIPRNG combination that deploys BRAM or DSP macros (MT, TT800), to be independent from the technology.

Results concerning CIPRNG-XOR and CIPRNG-MC (respectively CIPRNG-MCMD) are summarized in Table III and Table IV (resp. in Table V). In the former table, we specify which combination has been studied. In examples contained in these tables, *A* is for XOR64, *B* means XOR128+, and *C* is LFSR258. Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32 generators respectively.

CIPRNG Multi-Cycle: As recalled previously, this particular version of chaotic iterations post-treatment is based on two inputted PRNGs. For FPGA implementation, 7 CIPRNG combinations have been selected for their hardware performance. According to results presented in Table IV, throughput of CIPRNG Multi-Cycle is larger than those of almost all linear PRNGs that pass TestU01 (PCG, MRG32, and XOR64*). Additionally, the consumed area is globally small, even if 2 PRNGs are embedded and without inferring any blocks (DSPs and BRAM). Regarding statistical evaluation, all the selected combinations succeeded the TestU01, contrary to all other chaotic PRNGs based on Hénon [38], Lörenz & Chen [34], and Tent [43] maps.

CIPRNG-XOR: In this last version, 7 other combinations of CIPRNG-XOR generators have been selected for their hardware performance, when compared with linear PRNGs (see Table III). The results illustrate a throughput to generate 32 bits 2.5 times larger for CIPRNG-XOR than for almost all linear PRNGs that can pass TestU01. Furthermore, if we consider the Thoughput/Latency ratio, CIPRNG is respectively 12 times, 30 times, and finally 7 times faster than XOR64*, PCG32, and combined PNRGs (MRG32 and KISS124). Additionally, when DSPs blocks are disabled on use, CIPRNG-XOR is 25 times, 44 times, and finally 35 times faster than XOR64* (0.34Gbps), PCG32 (0.2Gbps), and combined MRG32 (0.25Gbps). The same statement holds for area: CIPRNG-XOR deploys 3 PRNGs, but it is 5 times more efficient than any other linear PRNGs. Compared to all other aforementioned chaotic PRNGs, all configurations of CIPRNG-XOR are more efficient in throughput, area, and ability to face statistical tests. Therefore, for FPGA application, all combinations can contribute in hardware performance and statistical tests compared to linear PRNGs. Finally, compared to CIPRNG-MC, the CIPRNG-XOR is less compact in area resources, but largely more efficient in terms of throughput.

CIPRNG Multi-Cycle Multi-Dimension: This is a final extended version of CIPRNG-MC, in which we apply our

TABLE III: FPGA Implementation of CIPRNG-XOR post-processing using different linear PRNG as strategy

CIPRNG (32Bits)		CIPRNG-XOR [PRNG-64bits, PRNG-64Bits, Strategy-32Bits]							Best Chaotic PRNG				Best Linear PRNG		
PRNG		[A,B,2]	[A,B,3]	[B,B,1]	[B,B,2]	[B,B,3]	[B,C,2]	[B,A,2]	FNDR[d][38]	LG[e][42]	TR[f][45]	B[e][43]	XOR64*	MRG	PCG32
AREA	LUT	364	357	237	222	226	502	345	208	313	***	***	298	1055	345
	FF	582	586	454	424	458	838	582	96	842	***	***	390	1359	418
	DSP	0	0	0	0	0	0	0	***	16	***	0	10	0	10
	RAM	0	0	0	0	0	0	0	***	***	***	***	4	8	0
	Total Area (LUT+FF)*8	7568	7544	5528	5168	5472	10720	7416	4568	9240	11903	***	5504	19312	6104
SPEED	Frequency (Mhz)	257.7	250	250.9	251.8	250	266	257.5	183	233	200	265.9	231	175	179
	Design Latency	3	3	3	3	3	3	3	***	8 to 16	***	***	21	14	20
	Output Latency	1	1	1	1	1	1	1	1	1	1	1	21	14	20
	Throughput/Latency (Gbps)	8.246	8.0	8.028	8.057	8.0	8.512	8.24	5.86	7.5	6.4	8.5	0.7	0.8	0.286
TEST	NIST	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
	TestU01	PASS	PASS	PASS	PASS	PASS	PASS	PASS	NO	NO	NO	NO	PASS	PASS	PASS

[a] *** No Information, [b] *A* is for XOR64, [c] *B* means XOR128+, [d] *C* is LFSR258, [e] Values 1,2, and 3 correspond to Taus88, LFSR113, XOR128.
[f] B: Bernoulli, [g] FNDR: Frequency Dependent Negative Resistors. [h] LG: Logistic Map. [i] TR: Timing Reseeding.

TABLE IV: FPGA Implementation of CIPRNG Mult-Cycle post-processing using different linear PRNG as strategy

CIPRNG (32Bits)		CIPRNG Mult-Cycle [PRNG-32bits, Strategy-32Bits]									Best Chaotic PRNG				Best Linear PRNG		
PRNG		[1-2]	[1,3]	[2,1]	[2-3]	[3-1]	[4-1]	[1,1]	[2,2]	[3,3]	FNDR[d][38]	LG[e][42]	TR[f][45]	B[e][43]	XOR64*	MRG	PCG32
AREA	LUT	194	201	187	194	175	119	197	211	175	208	313	***	***	298	1055	345
	FF	386	386	388	418	373	252	356	418	403	96	842	***	***	390	1359	418
	DSP	0	0	0	0	0	0	0	0	0	***	16	***	0	10	0	10
	RAM	0	0	0	0	0	0	0	0	0	***	***	***	***	4	8	0
	Total Area (LUT+FF)*8	4640	4696	4600	4896	4384	2968	4424	5032	4744	4568	9240	11903	***	5504	19312	6104
SPEED	Frequency (Mhz)	304	322	327	307	312	326	300	330	288	183	233	200	265.9	231	175	179
	Design Latency	3/330	3/330	3/330	3/330	3/330	3/330	3/330	3/330	3/330	***	8 to 16	***	***	21	14	20
	Output Latency	3/330	3/330	3/330	3/330	3/330	3/330	3/330	3/330	3/330	1	1	1	1	21	14	20
	Throughput/Latency (Gbps)	3.2/0.03	3.4/0.03	3.5/0.03	3.3/0.03	3.3/0.03	3.478/0.03	3.196/0.03	3.523/0.03	3.069/0.03	5.86	7.5	6.4	8.5	0.7	0.8	0.286
TEST	NIST	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
	TestU01	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	NO	NO	NO	NO	PASS	PASS	PASS

[a] *** No Information, [b] Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32. [c] B: Bernoulli, [d] FNDR: Frequency Dependent Negative Resistors. [e] LG: Logistic Map. [f] TR: Timing Reseeding.

post-processing (Algorithms 1) on TGFSR family (Mersenne twister and TT800) and when tempering function is disabled. This latter offers to us a well-uniform and multi-dimensional distribution. As can be seen in Table V, this new post-processing provides the same hardware performance as the original TGFSR PRNGs. Additionally, this new post-processing improves generators, in such a way that they are able to pass the statistical TestU01 battery, while providing improved performances with almost all chaotic PRNGs, as the ones of [34], [38], [43]. Due to such qualities, these new types of CIPRNGs can thus contribute to parallel processing and computation applications, like in Monte-Carlo simulation.

TABLE V: FPGA implementation of Multi-Cycle Multi-Dimension chaotic iteration post-processing based for MT and TT800

	Mersenne Twister		TT800	
Strategy	Taus88	XOR128	Taus88	LFRS113
LUT	434	447	358	357
FF	676	672	830	853
DSP	3	3	6	6
RAM	2	2	2	2
Area (LUT+FF)*8	8880	8952	9504	9680
Design_Latency	3	3	3	3
Output_Latency	1	1	1	1
Throughput/Latency	4.8	4.8	5.2	5.3

VII. ASIC Implementation

A. General Presentation

Compared to FPGA flow, the ASIC one consists of implementing our design in a specific process technology at transistor level. In our case, UMC-65*nm* LL represents the process technology node, where the Cadence tools v14 are the main software for the implementation purpose.

Table VI summarizes the ASIC implementation, which uses two global flows: the synthesis flow using *Cadence RTL Compiler*, and physical place and route (P&R) flow in a second step, with *Cadence Encounter Digital Implementation*. Both flows include *Switching Activity Interchange* information generated from simulation process for timing and dynamic power estimation (1 million samples). In addition, signoff verification flow is used to close timing and power requirements. The condition operation mode for the technologies deployed in each flow is as follows: the synthesis is based on one mode using the *Worst Case* library (WC=108°C and 1.08 Volt), while *Multi Mode Multi Corner* is applied for P&R flow including both worst and best case library (BC=−40°C and 1.32 Volt).

TABLE VI: 65nm ASIC Implementation of two chaotic iteration post-processing using different linear PRNG as strategy

CIPRNG (32Bits)		CIPRNG Multi Cycle [PRNG-32bits, Strategy-32Bits]									CIPRNG-XOR [PRNG-64bits, PRNG-64Bits, Strategy-32Bits]						
PRNG		[1,2]	[1,3]	[2,1]	[2,3]	[3,1]	[4,1]	[1,1]	[2,2]	[3,3]	[A,B,2]	[A,B,3]	[B,B,1]	[B,C,2]	[B,A,2]	[B,B,3]	[B,B,2]
Area	Standard Cells Area μm^2	4718	4739	4791	5267	4437	3229	4372	5136	4831	9070	9165	11240	12579	9104	11867	11168
	Gate Elements (GE μm^2)	3276	3291	3327	3658	3081	2242	3036	3567	3355	6299	6465	7806	8735	6322	8240	7756
	Transistor Area (TE μm^2)	13104	13164	13308	14632	12324	8968	12144	14268	13420	25196	25860	31224	34940	25288	32960	31024
Speed	Frequency (Mhz)	492	427	478	594	473	380	489	427	454	276	340	273	275	281	248	270
	Output Latency	3/330	3/330	3/330	3/330	3/330	3/330	3/330	3/330	3/330	1	1	1	1	1	1	1
	Throughput /Latency (Gbps)	5.2/0.03	4.6/0.04	5/0.05	6/0.6	5/0.05	4/0.04	5.2/0.05	4,5/0.04	4.8/0.04	8.8	10.9	8.7	8.8	9	7.9	8.6
Power	Internal Power (mW)	1.08	1.03	1.05	1.16	0.98	0.72	1	1.11	1.07	1.72	1.9	2.023	2.45	1.74	2.27	1.83
	Switching Power (mW)	0.37	0.37	0.39	0.5	0.34	0.25	0.36	0.41	0.38	0.83	0.94	0.98	1.2	0.86	1.15	0.93
	Total Power (mW)	1.46	1.4	1.45	1.66	1.32	0.97	1.36	1.52	1.45	2.56	2.73	3.02	3.67	2.61	3.44	2.77

[a] A is for XOR64, [b] B means XOR128+, [c] C is LFSR258, [d] Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32.

B. ASIC Comparison

The result analyzes of the various ASIC implementation of CIPRNG can be summarized as follow.

Area Analysis: When dealing with ASIC implementations, two measures can be considered to evaluate area consumption: either the *Gate Equivalent* (GE = Area / (AND gate area for $65nm = 1.44\mu m^2$)) or the number of transistors (TE = GE$\times$4, where AND has 4 transistors). This latter is independent from the technology estimation of the area of the circuit. It is obvious that CIPRNG-XOR needs twice the area of CIPRNG-MC, due to the use of three generators. For CIPRNG-MC, $[1,3]$, $[2,1]$, and $[4,1]$ have the lowest area, which uses Taus88 (1) as a strategy. In the case of CIPRNG-XOR, $[A,B,2]$, $[A,B,3]$, and $[B,A,2]$ are selected as best candidates for the lowest area consumption in chaotic iterations based PRNGs.

Static Timing Analysis: Physical implementation flow introduces a large amount of changes when compared with RTL design (*i.e.*, datapath transformation). Following Table VI, the CIPRNG-MC throughput is twice better than CIPRNG-XOR, and similarly for the area. However, due to the latency problem, this latter drops the throughput and balance CIPRNG-XOR up to 200 times the ones of CIPRNG-MC and other linear PRNGs who pass TestU01. Finally, combinations $[A,B,3]$, $[B,A,2]$, and $[A,B,2]$ are candidates of chaotic iterations PRNGs who pass TestU01 and with good time performances.

Power Analysis: Concerning power analysis, we estimated both static and dynamic power, which compute *leakage* and *switching&internal* power of the design. The leakage power measures each cell (logic) in various states, while dynamic power depends on the initial state of cells, the toggling input, the transition rate, and the output capacitive load. In Table VI, various dynamic power analyzes illustrate a low power consumption of both CIPRNG-MC and CIPRNG-XOR. It is clear from Table VI that, when we propagate the clock (switching), the switching power of the CIPRNGs is lower than the internal power consumed by the internal cell of CIPRNGs. This is confirmed by the area of both CIPRNG family. Despite such results, CIPRNG-XOR consumes twice the power of CIPRNG-MC, which is balanced by frequency and throughput. We can finally select the combinations $[B,A,2]$, $[A,B,2]$, and $[B,B,2]$ as candidates of chaotic iterations based PRNGs who can pass TestU01 and with power performances.

VIII. Statistical tests

During experiments, the test batteries are run in Z-book Intel Core $i7-4800MQCPU@2.70GHz\times 8$, working with Ubuntu 16.4 (64bits) and GCC 5.4.0. For NIST, 100 sequences of 10^6 bits are generated and tested. The results confirm that all the chaotic iterations post-processings for linear PRNGs can pass the NIST, where the minimum passing rate for each statistical test is approximately 96 for a sample size of 100 binary sequences. In the TestU01 case, all CIPRNG configurations for both proposals (MC and XOR) can successfully pass this battery, which is failed when considering the other chaotic PRNGs evoked in this article.

IX. Conclusion

In this paper, which is an extension of [16], we have presented a new family of post-processing PRNGs based on chaotic iterations for FPGA and ASIC. This work has studied the performance of various linear PRNGs implementations in FPGA regarding the linear complexity, seed size, arithmetic operations, and throughput/latency. In order to investigate these parameters, a SoC based on Zynq EPP platform (hardware and firmware) has been developed to accelerate the implementation and tests of various PRNGs on FPGA. The results are used as sources of information in the design of an hardware post-processing treatment based on chaotic iterations. This latter has been considered to improve the statistical profile of flawed generators. The conclusion that can be outlined is that chaotic iterations post-processing provides an alternative implementation of combined PRNGs without any supplemental cost, which is 2.5 times faster and 5 times more efficient than almost all the linear PRNGs that can pass TestU01.

References

[1] A. Vassilev and T. A. Hall, "The importance of entropy to information security," *Computer*, vol. 47, no. 2, pp. 78–81, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1109/MC.2014.47

[2] A. Vassilev and R. Staples, "Entropy as a service: Unlocking cryptography's full potential," *Computer*, vol. 49, no. 9, pp. 98–102, Sept 2016.

[3] P. L'Ecuyer, "Uniform random number generation," *Annals of Operations Research*, vol. 53, no. 1, pp. 77–120, 1994.

[4] L. Kocarev, J. Szczepanski, J. M. Amigo, and I. Tomovski, "Discrete chaos-i: Theory," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 6, pp. 1300–1309, June 2006.

[5] X. Fang, B. Wetzel, J. M. Merolla, J. M. Dudley, L. Larger, C. Guyeux, and J. M. Bahi, "Noise and chaos contributions in fast random bit sequence generated from broadband optoelectronic entropy sources," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 3, pp. 888–901, March 2014.

[6] T. Addabbo, A. Fort, L. Kocarev, S. Rocchi, and V. Vignoli, "Pseudo-chaotic lossy compressors for true random number generation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 8, pp. 1897–1909, 2011.

[7] P. Z. Wieczorek and K. Goofit, "Dual-metastability time-competitive true random number generator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 1, pp. 134–145, Jan 2014.

[8] R. L. Devaney, *An Introduction to Chaotic Dynamical Systems, 2nd Edition*. Westview Pr., March 2003.

[9] T. Y. Li and J. A. Yorke, "Period three implies chaos," *Amer. Math. Monthly*, vol. 82, no. 10, pp. 985–992, 1975.

[10] T. Stojanovski and L. Kocarev, "Chaos-based random number generators-part i: analysis [cryptography]," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 48, no. 3, pp. 281–288, Mar 2001.

[11] F. Pareschi, G. Setti, and R. Rovatti, "Implementation and testing of high-speed cmos true random number generators based on chaotic systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 12, pp. 3124–3137, Dec 2010.

[12] Y. Wu, Y. Zhou, and L. Bao, "Discrete wheel-switching chaotic system and applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 12, pp. 3469–3477, Dec 2014.

[13] Y. Liu, R. C. C. Cheung, and H. Wong, "A bias-bounded digital true random number generator architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 1, pp. 133–144, Jan 2017.

[14] J. Choi, J. Jung, and I. C. Park, "Area-efficient approach for generating quantized gaussian noise," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 7, pp. 1005–1013, July 2016.

[15] V. Rajagopalan, V. Boppana, S. Dutta, B. Taylor, and R. Wittig, "Xilinx zynq-7000 epp–an extensible processing platform family," in *23rd Hot Chips Symposium*, 2011, pp. 1352–1357.

[16] M. Bakiri, J.-F. Couchot, and C. Guyeux, "Fpga implementation of f2-linear pseudorandom number generators based on zynq mpsoc: A chaotic iterations post processing case study," in *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications - Volume 4: SECRYPT,*, 2016, pp. 302–309.

[17] A. Canteaut, "Berlekamp–massey algorithm," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 80–80.

[18] R. M. May *et al.*, "Simple mathematical models with very complicated dynamics," *Nature*, vol. 261, no. 5560, pp. 459–467, 1976.

[19] J. Černák, "Digital generators of chaos," *Physics letters A*, vol. 214, no. 3, pp. 151–160, 1996.

[20] O. Rssler, "An equation for continuous chaos," *Physics Letters A*, vol. 57, no. 5, pp. 397 – 398, 1976.

[21] J. M. Bahi, X. Fang, C. Guyeux, and L. Larger, "Fpga design for pseudo-random number generator based on chaotic iteration used in information hiding application," *Appl. Math*, vol. 7, no. 6, pp. 2175–2188, 2013.

[22] P. LEcuyer, "Tables of maximally equidistributed combined lfsr generators," *Mathematics of Computation of the American Mathematical Society*, vol. 68, no. 225, pp. 261–269, 1999.

[23] P. Lecuyer, "Maximally equidistributed combined tausworthe generators," *Mathematics of Computation of the American Mathematical Society*, vol. 65, no. 213, pp. 203–213, 1996.

[24] D. B. Thomas and W. Luk, "The lut-sr family of uniform random number generators for fpga architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 4, pp. 761–770, April 2013.

[25] M. E. ONeill, "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation," *ACM Trans. Math. Softw.(submitted)*, pp. 1–46, 1988.

[26] P. L'ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.

[27] G. Marsaglia, "Random number generators," *Journal of Modern Applied Statistical Methods*, vol. 2, no. 2, 2003.

[28] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[29] F. Panneton, P. L'ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 1, pp. 1–16, 2006.

[30] M. Matsumoto and Y. Kurita, "Twisted gfsr generators ii," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 4, no. 3, pp. 254–266, 1994.

[31] G. Marsaglia *et al.*, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[32] S. Vigna, "An experimental exploration of marsaglia's xorshift generators, scrambled," *arXiv preprint arXiv:1402.6246*, 2014.

[33] ——, "Further scramblings of marsaglias xorshift generators," *Journal of Computational and Applied Mathematics*, vol. 315, pp. 175–181, 2017.

[34] M. A. Zidan, A. G. Radwan, and K. N. Salama, "The effect of numerical techniques on differential equation based chaotic generators," in *ICM 2011 Proceeding*, Dec 2011, pp. 1–4.

[35] G. Chen and J. Lü, "Dynamics of the lorenz system family: analysis, control and synchronization," *SciencePress, Beijing*, 2003.

[36] A. S. Elwakil and M. P. Kennedy, "Construction of classes of circuit-independent chaotic oscillators using passive-only nonlinear devices," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 48, no. 3, pp. 289–307, Mar 2001.

[37] M. A. Zidan, A. G. Radwan, and K. N. Salama, "Random number generation based on digital differential chaos," in *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*. IEEE, 2011, pp. 1–4.

[38] P. Dabal and R. Pelka, "Fpga implementation of chaotic pseudo-random bit generators," in *Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2012*, May 2012, pp. 260–264.

[39] A. Elwakil and M. Kennedy, "Chaotic oscillator configuration using a frequency dependent negative resistor," *International journal of circuit theory and applications*, vol. 28, no. 1, pp. 69–76, 2000.

[40] M. Hénon, "A two-dimensional mapping with a strange attractor," *Communications in Mathematical Physics*, vol. 50, no. 1, pp. 69–77, 1976.

[41] P. Dabal and R. Pelka, "A chaos-based pseudo-random bit generator implemented in fpga device," in *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, April 2011, pp. 151–154.

[42] ——, "A study on fast pipelined pseudo-random number generator based on chaotic logistic map," in *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, April 2014, pp. 195–200.

[43] P. Giard, G. Kaddoum, F. Gagnon, and C. Thibeault, "Fpga implementation and evaluation of discrete-time chaotic generators circuits," in *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, Oct 2012, pp. 3221–3224.

[44] T. Geisel and V. Fairen, "Statistical properties of chaos in chebyshev maps," *Physics Letters A*, vol. 105, no. 6, pp. 263–266, 1984.

[45] C.-Y. Li, J.-S. Chen, and T.-Y. Chang, "A chaos-based pseudo random number generator using timing-based reseeding method," in *2006 IEEE International Symposium on Circuits and Systems*, May 2006, pp. 4 pp.–3280.

[46] C. Y. Li, Y. H. Chen, T. Y. Chang, L. Y. Deng, and K. To, "Period extension and randomness enhancement using high-throughput reseeding-mixing prng," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 2, pp. 385–389, Feb 2012.

[47] E. Barker and A. Roginsky, "Draft NIST special publication 800-131 recommendation for the transitioning of cryptographic algorithms and key sizes," 2010.

[48] P. L'Ecuyer and R. Simard, "Testu01: Ac library for empirical testing of random number generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 22, 2007.

[49] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[50] R. A. Rueppel, "Linear complexity and random sequences," in *Advances in CryptologyEUROCRYPT85*. Springer, 1985, pp. 167–188.

[51] Q. Wang, S. Yu, C. Li, J. L, X. Fang, C. Guyeux, and J. M. Bahi, "Theoretical design and fpga-based implementation of higher-dimensional digital chaotic systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 3, pp. 401–412, March 2016.

[52] J. Bahi, R. Couturier, C. Guyeux, and P.-C. Héam, "Efficient and cryptographically secure generation of chaotic pseudorandom numbers on gpu," *The journal of Supercomputing*, vol. 71, no. 10, pp. 3877–3903, oct 2015.

[53] S. Contassot-Vivier, J.-F. Couchot, C. Guyeux, and P.-C. Heam, "Random walk in a n-cube without hamiltonian cycle to chaotic pseudo-random number generation: Theoretical and practical considerations," *International Journal of Bifurcation and Chaos*, vol. *, p. *, 2016.
[54] Knudsen, "Chaos without nonperiodicity," *Amer. Math. Monthly*, vol. 101, 1994.
[55] C. Guyeux and J. Bahi, "An improved watermarking algorithm for internet applications," in *INTERNET'2010. The 2nd Int. Conf. on Evolving Internet*, Valencia, Spain, sep 2010, pp. 119 – 124.

Mohammed Bakiri received his B.S. degree in electrical engineering from Saad Dahleb University, in 2009, and then Master degree from Amar Thelidji University, in 2011, Algeria. He is currently Ph.D student in Informatics in the Femto-ST Institute, Bourgogne Franche-Comté University, France. He joined the Centre de Développement des Technologie Avancées, CDTA, Algiers, in 2010, as research engineer in Microelectronics. He is presently working on new chaotic algorithms for pseudorandom number generators dedicated to FPGA/ASIC.

Jean-François Couchot is an Associate Professor in the Department of Computer Science (DISC) of the FEMTO-ST institute, Bourgogne Franche-Comté University. He received a Ph.D. in Computer Science in 2006 in the FEMTO-ST institute. He has applied for a postdoctoral position at INRIA Saclay Ile de France in 2006. His research focuses on discrete dynamic systems (data hiding, pseudorandom number generators, hash function) and on bioinformatics, especially in gene evolution prediction. He has written more than 25 scientific articles in these areas.

Christophe guyeux The research interests of Pr. Guyeux are in the areas of interdisciplinary sciences and complex systems. He applies techniques from mathematics and/or computer science to solve scientific questions raised in biology, environment, or computer science fields. Current areas of application in computer science include chaos study of discrete dynamical systems applied to information security, wireless sensor networks, and biology.