Journal of Computer and Communications, 2016, 4, 73-92
Published Online April 2016 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2016.44007

Ensuring Quality of Random Numbers from TRNG: Design and Evaluation of Post-Processing Using Genetic Algorithm

Jose J. Mijares Chan[1], Parimala Thulasiraman[2], Gabriel Thomas[1], Ruppa Thulasiram[2]

[1]Electrical and Computer Engineering Department-Faculty of Engineering, University of Manitoba, Winnipeg, Canada
[2]Computer Science Department, Faculty of Science, University of Manitoba, Winnipeg, Canada
Email: thulasir@cs.umanitoba.ca

Received 19 February 2016; accepted 5 April 2016; published 8 April 2016

Copyright © 2016 by authors and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/
Open Access

Abstract

Random numbers generated by pseudo-random and true random number generators (TRNG) are used in a wide variety of important applications. A TRNG relies on a non-deterministic source to sample random numbers. In this paper, we improve the post-processing stage of TRNGs using a heuristic evolutionary algorithm. Our post-processing algorithm decomposes the problem of improving the quality of random numbers into two phases: (i) *Exact Histogram Equalization*: it modifies the random numbers distribution with a specified output distribution; (ii) *Stationarity Enforcement*: using genetic algorithms, the output of (ii) is permuted until the random numbers meet wide-sense stationarity. We ensure that the quality of the numbers generated from the genetic algorithm is within a specified level of error defined by the user. We parallelize the genetic algorithm for improved performance. The post-processing is based on the power spectral density of the generated numbers used as a metric. We propose guideline parameters for the evolutionary algorithm to ensure fast convergence, within the first 100 generations, with a standard deviation over the specified quality level of less than 0.45. We also include a TestU01 evaluation over the random numbers generated.

Keywords

True Random Number Generators, Genetic Algorithms, Auto-Correlation, Entropy, Power Spectral Density

1. Introduction

As we move towards the Big data and the exascale era, with the Internet and many communication devices

How to cite this paper: Chan, J.J.M., Thulasiraman, P., Thomas, G. and Thulasiram, R. (2016) Ensuring Quality of Random Numbers from TRNG: Design and Evaluation of Post-Processing Using Genetic Algorithm. *Journal of Computer and Communications*, **4**, 73-92. http://dx.doi.org/10.4236/jcc.2016.44007

becoming more prevalent, cyber-security is becoming increasingly important. Many real world computational science applications (such as finance, medicine, and social networks among others) produce tremendous amount of data that not only require a larger storage capacity such as Cloud, but also secure methods for preserving the data from harmful threats.

In recent years, different solution paths had been explored such as new security models [1]-[3], analytical tools [4], prevention strategies [5], keys and random numbers management systems [6] and more relevant to this study, random number generation schemes [7]. Traditionally, the backbone solution for many of the security issues is the use of cryptographic algorithms. A cryptographic algorithm is used for securing private communication over untrusted environments. These algorithms are usually key-based and depend on the strength of random numbers used to generate the keys [8].

Over the past few years, with its increasing popularity and wide coverage, Cloud computing has become an important and frequent solution for storage of Big data. There are many challenges to address, one being the new security challenges. Among these new challenges, the combination of managing and storing the keys that provide access to the client's encrypted data can lead to multiple hazardous situations [9]. A poor key management scheme can expose the system to unauthorized access, data breaches and lack of control caused by having multiple clients sharing the same resource. It can also expose scenarios where network-based cross-client attacks can occurr over shared network infrastructure components. In the same way, other scenarios include losing control over the network infrastructure, either real or virtualized.

The poor key management problem is exacerbated by two important factors: the accidental key replication and/or the weak random number generation [9]. The first factor is due to a weak security policy. It occurs during the cloning of a virtual machine as part of an on-demand service, and in consequence, the cloned images could contain information that is not intended to be public. The second factor is introduced when the entropy level found in the host is insufficient, which compromises the statistical properties of the random numbers. These situations are created by either the virtualization of hardware that delivers a lower entropy level than is expected, or by having too many clients on the same host exhausting the entropy that the host can deliver.

In general, random numbers are generated using a pseudo-random number generator (PRNG), a true-random number generator (TRNG) or a combination of both [10]. A PRNG is an algorithm that in combination with a given *seed* creates a deterministic sequence with the characteristic of being forward and backward unpredictable. The sequence is forward unpredictable, when the history of output numbers do not lead to predict the next output value; if the seed used to create the sequence is unknown, consecutive output sequence numbers cannot backtrack to the unknown seed, creating a sequence that is also backward unpredictable. The PRNGs main drawback is being predictable if the seed and the algorithm are known. This is dangerous and vulnerable to attacks. Despite this, the National Institute of Standards and Technology (NIST) [11] has tried to standardize the selection method for the generator, as well as, suggesting a list of approved PRNGs for cryptographic algorithms [12].

On the contrary, true random generators provide stronger solution. These types of generators use non-deterministic or unpredictable sources to generate random numbers, like thermal, avalanche or atmospheric noise. The random number generation process consists of two phases: *extraction* and *post-processing*. In the extraction phase, a non-deterministic source is sampled to extract the random numbers. This sampled values may not be completely unbiased or consistent. Therefore, in the post-processing phase, the extracted random numbers undergo a strenuous process that modifies the numbers to satisfy the designed statistical properties - in other words, ensure that the numbers are close to be independent and identically distributed (i.i.d.) with uniform distribution [13]. Although the research on PRNGs has been exhaustively covered, the generation of seeds based on a TRNG has been overlooked [11]. Given its unpredictability of the random numbers, gaming machines and strong cryptographic applications apply a scheme where a TRNGs seeds a PRNG. In this paper, we focus on the TRNG post-processing.

In the literature, there are multiple types of sources, design and extraction techniques [14] [15] for TRNGs, making it a complicated case for standardization [11]. In some cases, the design uses a thermal noise circuit that is amplified and sampled [14]. In [16] the design focuses on sampling the jitters of a phase-locked loop circuit found on ASIC based technologies. In [17] the authors exploit the features found in digital circuits, like the meta-stable state of flip-flop circuits. As well, researchers have also explored the use of random sources from audio and video signals [18].

Intel introduced the Intel random number generator [19], relying on the unpredictability of the thermal noise

to modulate the frequency of a clock signal as a source of random binary digits. The design also included a post-processing stage, involving a von Neumann corrector to statistically balance the outputs from data with a bias. Intel re-engineered [20] its own true random number generator in 2011, providing a solution to security threats on the Ivy Bridge processor. In this new design, the noise circuit relied on the thermal noise and the meta-stable state of a R/S Latch, where the source of random binary digits is located at the output of the R/S Latch. One of the main features of this new design is the on-line health tests or On-line Self Test Entropy (OSTE), that empirically test the frequency of binary patterns within a certain range of test against intentional attacks. Using these tests, the random numbers with only the healthy sequences are appended to an entropy pool, from which random numbers are later picked up. Another source for TRNGs are the ones based on race conditions between CPU threads while updating shared variables. This creates a non-deterministic behavior on the value of the updated variable. Colesa *et al*. [15] took advantage of this phenomena. Their study suggests that the execution environment's irregularities, the number of cache misses, the number of instructions executed in the pipeline and the imprecision of the hardware clock used for the timer interrupts are the main contributors to the random behavior observed in the shared variable. They claim that their design satisfies 90% of the standard NIST tests [11].

In [21], we developed a TRNG that generates high quality random numbers by exploiting the natural sources of randomness on the NVIDIA GPU video card such as race conditions during concurrent memory access. The digitized noise signal generated from non-deterministic sources passes through a post-processing algorithm, to ensure the random numbers follow an uniform distribution. Unlike Intel, which invokes the post-processing stage in hardware, our algorithm implements the post-processing step in software, providing flexibility and the capacity to scale the complexity of the algorithm design. The modularity of the design can be adapted to any noise source.

In this paper, the post-processing algorithm proposed on [22] for TRNGs is further explored focusing on a lighter version optimized for quality and performance, ensuring the generation of random numbers meet wide sense stationarity (w.s.s.). In the post-processing step, we apply an evolutionary based genetic algorithm (GA) heuristic that exploits the inherent parallelism avoiding all data dependencies to produce uniform random numbers that satisfies the same statistical properties over time, thereby enforcing stationarity. We ensure that the quality of the numbers generated from the genetic algorithm is within a specified level of error. As well a detailed evaluation using the statistical battery test from L'Ecuyers TestU01 [23] is provided.

The rest of this paper is organized as follows. In Section 2, we give a brief theoretical introduction about the properties of random numbers, its metrics and algorithms. In Section 3, we present our TRNG framework. Section 4 presents the post-processing phase which includes the genetic algorithm. Section 6 presents the results of the evaluations stated in Section 5. Finally in Section 7, we conclude our work and suggest future work.

2. Properties of Random Numbers

Random numbers are a type of numbers that when chosen, its selection process reproduces the characteristics of an underlying distribution. Conceptually, a random number is generated from a random process with the intention to be nearly indistinguishable from a random behavior [13]. Nevertheless, the study of computer-generated random numbers is a fascinating area of research that has been addressed by multiple researchers [13] [24]-[26]. Formally, we can define a random number as an outcome of a random variable X, that takes on different values, $x = \{x_1, x_2, \cdots, x_m\}$ with a probability distribution P_X, usually assumed to be an uniform distribution $U[0,1]$. Once S is defined as a sample space or domain, S can be used to map X to its range x. Then, the i-th random number, x_i, can be expressed by $X(s_i) = x_i$ where $s_i \in S$. The i^{th} index can represent the sample index on a sequence, and it can also be associated to a time index in the generation of this random numbers. For example, as in **Figure 1(a)**, we can observe multiple realizations of X, where the i-th realization of X, $x_i \in [0,1]$ is mapped by $s_i \in [0, 2^{24} - 1]$ representing a sample between 0 and 1 at the maximum resolution using a IEEE 754 floating point format. In general, a random number should meet two basic conditions to be considered random: have an uniform distribution and be unpredictable.

2.1. The Uniform Distribution of Random Numbers

In order to observe the distribution of x, the probability mass function (p.m.f.), or f_x, is a useful tool that relies

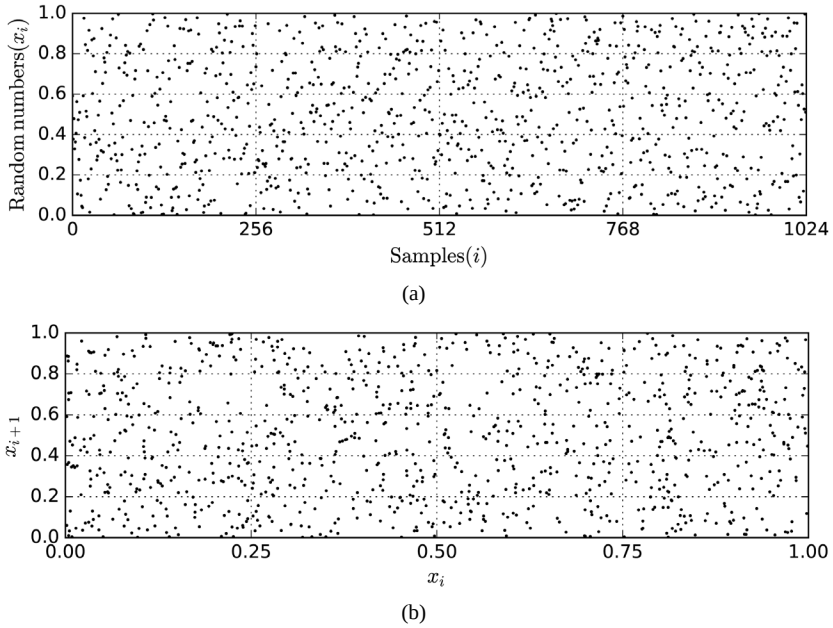

Figure 1. A sample of 1024 random numbers (a); and the patterns between contiguous random numbers (b).

on three properties. The p.m.f. is always positive, its summation of all the p.m.f. is 1.0, and the p.m.f. can be used to calculate the probability of an event by adding all the elements that conform that event. Under ideal conditions, *x* should follow an uniform distribution, as in p.m.f. in **Figure 2(a)**. But in reality, for some random number generators this is difficult to achieve, a typical response is shown in **Figure 2(d)**.

2.2. The Correlation of Random Numbers

As well, in ideal conditions all elements of *x* should be uncorrelated. This can be observed as non-geometric patterns between pairs of contiguous realizations, like between $(x_0, x_1), (x_1, x_2), \cdots, (x_{N-1}, x_N)$ as in **Figure 1(b)**. Similarly, the uncorrelation of *x* can be proven by observing the shape of the autocorrelation function.

The autocorrelation is a mathematical tool that calculates the correlation between the elements on a sequence of numbers, like *x*. It is useful at identifying non-randomness in a sequence and its adequate time series model [27]. The autocorrelation function, $R_{xx}[k]$, is defined by the expectation between two elements in *x* separated by a *k* lag under the assumption that all elements of *x* are equi-distant [28].

$$R_{xx}[k] = \frac{E[x_i - \mu, x_{i+k} - \mu]}{\sigma^2} \tag{1}$$

where $k \in [1-N, N-1]$ is the lagged interval, *N* is the cardinality of *x*, μ and σ is the mean and standard deviation of *x*. As well, R_{xx} can be calculated by replacing μ and σ^2 for the sample mean $\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i$ and the sample variance,

$$R_{xx}[k] = \frac{\sum_{i=0}^{N-k-1} (x_i - \overline{x})(x_{i+k} - \overline{x})}{\sum_{i=0}^{N-1} (x_i - \overline{x})^2} \tag{2}$$

When there is no lag or $k = 0$, the autocorrelation reaches it's maximum value, $R_{xx}[0] = 1$. Differently, as $k \neq 0$ the autocorrelation is always $|R_{xx}[k \neq 0]| < 1$. Since the denominator in (2) normalizes the output, the

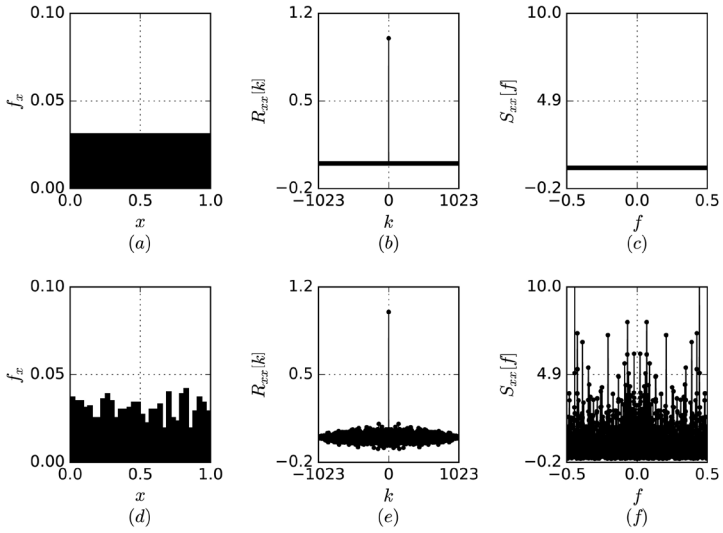

Figure 2. Probability mass function (a), autocorrelation (b) and power spectral density (c) of *x* random numbers in an ideal case scenario. Probability mass function (d); autocorrelation (e) and power spectral density (f) of *x* random numbers from a computer generated algorithm.

autocorrelation function is bounded to $[-1,1]$. In an ideal case, the autocorrelation of a random sequence will result in an impulse shape, as shown in **Figure 2(b)**, showing that random numbers are only correlated at $k=0$, and uncorrelated at any other lag, $R_{xx}[k\neq 0]=0$. In reality, computer-generated random numbers hold small traces of correlation, having a close to zero response when $R_{xx}[k\neq 0]=0$ as shown in **Figure 2(e)**.

The execution of the autocorrelation function from (2) is a computational expensive task that requires a total of $4N^2+N$ floating point operations and $4N^2+9N+4$ memory accesses. There are many other efficient algorithms, among them, the method based on the Wiener-Khinchin theorem. This method uses two Fast Fourier Transforms (FFT) to calculate the autocorrelation and in combination with a single pass variance calculation, like the Welford method [29] [30]; it takes $12N\log_2(N)+9N-2$ floating point operations and $10N\log_2(2N)+12N-4$ memory accesses to calculate it. The algorithm includes the following operations,

$$S_{xx}[f]=\frac{FFT[x-\bar{x}]FFT^*[x-\bar{x}]}{(2N-1)s^2} \tag{3}$$

$$R_{xx}[k]=IFFT[S_{xx}] \tag{4}$$

where in (3), $FFT^*[\cdot]$ represent the complex conjugate operation of the $FFT[\cdot]$, the $FFT[\cdot]$ operation involves calculating the Fast Fourier Transform of *x* after removing $\bar{x}$ and zero padded to $2N$ in size. Then, $S_{xx}[f]$ is the power spectral density of *x* in terms of frequency *f*, while in (4), $IFFT[\cdot]$ is the inverse FFT operation. As well in Equation (3), the variance, s^2, is the Welford method implementation based on the following equations,

$$M_k=\begin{cases}x_k & k=1,\\ M_{k-1}+\dfrac{x_k-M_{k-1}}{k} & \text{else}\end{cases} \tag{5}$$

$$s_k=\begin{cases}0 & k=1,\\ s_{k-1}+(x_k-M_{k-1})(x_k-M_k) & \text{else}\end{cases} \tag{6}$$

$$s^2=\frac{s_N}{N-1} \tag{7}$$

This implementation works well against losing precision due to having catastrophic cancellation, even though it is slightly slower than the naive method,

$$s^2 = \frac{\sum_{k=1}^{N} x_k^2 - \frac{\left(\sum_{k=1}^{N} x_k\right)^2}{N}}{N} \tag{8}$$

2.3. The Spectral Density of Random Numbers

Another useful tool to review the uncorrelation of random numbers is the power spectral density. It relies on the fact that through Fourier analysis, x can be decomposed into frequency components where it describes the energy distribution per unit of frequency, a distinctive signature between sequences of numbers. The relation between the autocorrelation and the power spectral density can be established with the Wiener-Khinchin theorem, which works under the assumption of x is the output of a stationary process. The stationarity of random numbers will be addressed in the following subsection. For the moment, consider that the power spectral density function is formally be defined as

$$S_{xx}[f] = \mathcal{F}\{R_{xx}\} = \sum_{k=1-N}^{N-1} R_{xx}[k] e^{-2\pi f k i} \tag{9}$$

where f is the frequencies at which the power spectral density is evaluated, and i is the imaginary unit. Its implementation can follow the proposed algorithm in (3), which only involves $6N\log_2(N)+9N-2$ floating point operations and $5N\log_2(2N)+12N-4$ memory accesses. Under ideal conditions, the power spectral density of random numbers has a step function response at 1, as shown in **Figure 2(c)**. But in reality, due to the small peaks found at $k \neq 0$ in the autocorrelation of **Figure 2(e)**, the power spectral density spreads around 1 as in **Figure 2(f)**.

Another property between the autocorrelation and the power spectral density is the distribution of the peaks found at $k \neq 0$ in the autocorrelation. For simplicity, these peaks will be referred to as $r_x = R_{xx}[k>0]$. Then, the distribution of r_x can be fitted to a normal distribution, for example in **Figure 2(e)**, $r_x \sim \mathcal{N}(\mu_{r_x} = -0.00048, \sigma_{r_x} = 0.02204)$. In other words, r_x follows a normal distribution f_{r_x}. This is

$$f_{r_x}(r) = \frac{1}{\sqrt{2\pi\sigma_{r_x}^2}} \exp\left(-\frac{(r-\mu_{r_x})^2}{2\sigma_{r_x}^2}\right) \tag{10}$$

where r is the magnitude of elements in r_x. Then, when mapping the f_{r_x} from the autocorrelation towards the power spectral density, we can analytically find the shape of the points distribution around the ideal step function in the power spectral density, **Figure 2(c)**, which happens to also follow a normal distribution [22]. This is done by using the characteristic function. The characteristic function of r_x is denoted by ϕ_{r_x}, and it is the Fourier transform of f_{r_x}. This is,

$$\phi_{r_x}(t) = \mathcal{F}\{f_{r_x}\}(t) = \exp\left(i\mu_{r_x}t - \frac{1}{2}\sigma_{r_x}^2 t^2\right) \tag{11}$$

where t is the frequency at which ϕ_{r_x} is evaluated. In most cases, the characteristic function is used to describe a probabilistic distribution behavior over time. In our case, it is a helpful tool to describe that standard deviation of f_{r_x} will always be smaller than the standard deviation from ϕ_{r_x}. In f_{r_x}, the variance or $\sigma_{r_x}^2$ is used as a denominator limiting the growth of the exponential to a small range; while in ϕ_{r_x}, the variance is used to magnify the growth of the exponential, making it more sensible to changes. This means that S_{xx} from (3) is more sensitive to variations from 1, than R_{xx} from (4) or (2) to the changes against an impulse shape. Besides the sensitivity of the metric, the computational intensity, $I = \frac{fpo}{mao}$, of each algorithm can express the relation between the number of floating point operations(fpo) and the memory access operations (mao) as in [31].

Also, the computational intensity can be evaluated in terms of the data size, as shown in **Figure 3(a)**. The autocorrelation from (2) shows to be data intensive in its majority, and only after the data size passes 2^{10}, the number of floating points operations grow closely to the number of memory access operations, **Figure 3(b)** and **Figure 3(c)**, where both keep a ratio of 1. However, the other two algorithms are compute intensive and follow a similar trend, since the power spectral density is necessary to compute the autocorrelation. As shown in **Figure 3(a)**, the computational intensity of the FFT-based autocorrelation from (4) and the power spectral density from (3) is higher, but the number of floating point operations is smaller than the autocorrelation, **Figure 3(b)**. Therefore the power spectral density is suited for long data segments where the computing architecture is ideal for massive floating point operations.

2.4. The Stationarity of Random Numbers

In order to keep the consistency of the random numbers definition among realizations, the statistical characteristics also need to be constant. As discussed previously, the stationarity of random number generation is key concept in this. Therefore, a stationary process is a process characterized by keeping its p.m.f. constant for any displacement over the sample indexes, this is $\forall k \in \mathbb{Z}, f_{x_i}(\cdot) = f_{x_{i+k}}(\cdot)$. This condition is also known as strict or strong-sense stationarity (s.s.s.). As well, process which are s.s.s. are also i.i.d. Unfortunately, in most cases s.s.s. is not achievable, therefore a less restrictive condition is used, wide-sense stationarity (w.s.s.) or covariance stationarity. A w.s.s. process requires only the mean and the autocorrelation to be constant over all sample indexes, presenting a more relaxed constraint.

According to Basu *et al*. [32], one way to confirm the w.s.s. of a process is to compare the power spectral density among all segments of the output. Considering that x is the output, we will extend the notation of x, where $\hat{x}_j$ is the j-th segment, so $\hat{x}_j \subset x$. Then,

$$\forall m, l \quad S_{\hat{x}_m \hat{x}_m}[f] = S_{\hat{x}_l \hat{x}_l}[f] \tag{12}$$

where, $\hat{x}_m$ and $\hat{x}_l$ represent two different segments of x. Given that x has M segments, comparing the m segment to every other $M-1$ segments is computationally expensive. Therefore, we propose another strategy that relaxes this constraint, and reduces the computational burden from [22]. Taking advantage that $S_{\hat{x}_j \hat{x}_j}$ has a

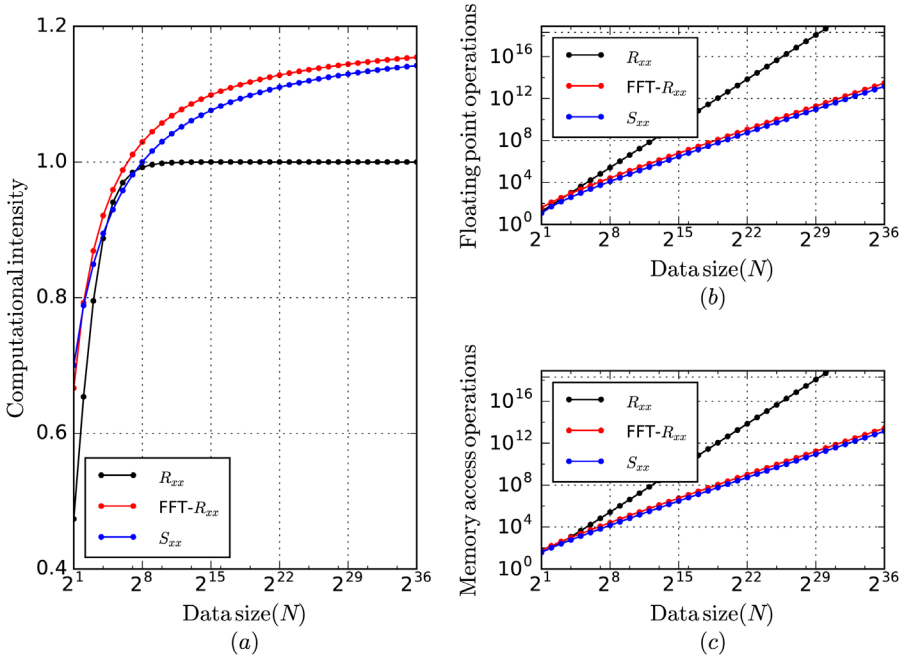

Figure 3. Comparison of the autocorrelation, FFT-based autocorrelation and power spectral density algorithms analyzing (a) the computational intensity; (b) floating point operations and (c) memory access operation in terms of the data size N.

normal distribution, we can introduce a user-specified standard deviation error, σ_0. Then, the segment will be ensured to be close to σ_0 from being w.s.s., such that

$$\forall m \quad \sigma_0 \geq \sigma_{S_{\hat{x}_m}} \tag{13}$$

The details on the algorithm and how (13) becomes an essential metric will be discussed on Section 4.

3. The Architecture of a TRNG

In general, there are three phases [11] [33] that are fundamental in the TRNG architecture, **Figure 4**:

3.1. *The digitized noise source/extraction*: This phase is composed of a noise source and a digitizer that periodically samples and outputs the digitized analog signal (DAS) random numbers. In this work we exploit the natural sources of randomness of the Intel DRNG available on Ivy Bridge Processors [34], the values are being sampled from the /*dev*/*random* over a linux kernel.

3.2. *The post-processing phase*: This phase transforms the DAS random numbers from the previous phase, into internal random numbers by satisfying the given statistical properties of the distribution. In our case, we propose the post processing phase, **Figure 4**, to be divided into two blocks, *histogram specification* and *stationarity enforcement*, as previously suggested on [21] [22]. The histogram specification block maps the output of step 3.1, or DAS numbers, to the specified output distribution, in our case an uniform distribution. This block was studied in detail in [21] and will not be discussed further in this paper.

3.3. *The buffer phase*: This is usually optional and it represents the final phase. The buffered internal random numbers are outputted as the external random numbers.

In **Figure 4**, the output of the histogram specification block is referred to as semi-internal (s.i.) random numbers. The s.i. random numbers are then inputted to the stationarity enforcement block which further improves the generated random numbers. The main purpose of this block is to eliminate any periodicity traces found in s.i random numbers.

The stationarity enforcement block uses a genetic algorithm (GA) to permute the output (s.i. random numbers) from the histogram equalization block. It is within this block where we ensure the random numbers to meet w.s.s. by iterating with GA until the random numbers meet w.s.s with a user-defined error level or standard deviation, σ_0. Further details on the stationarity enforcement block will be discussed in Section 4.

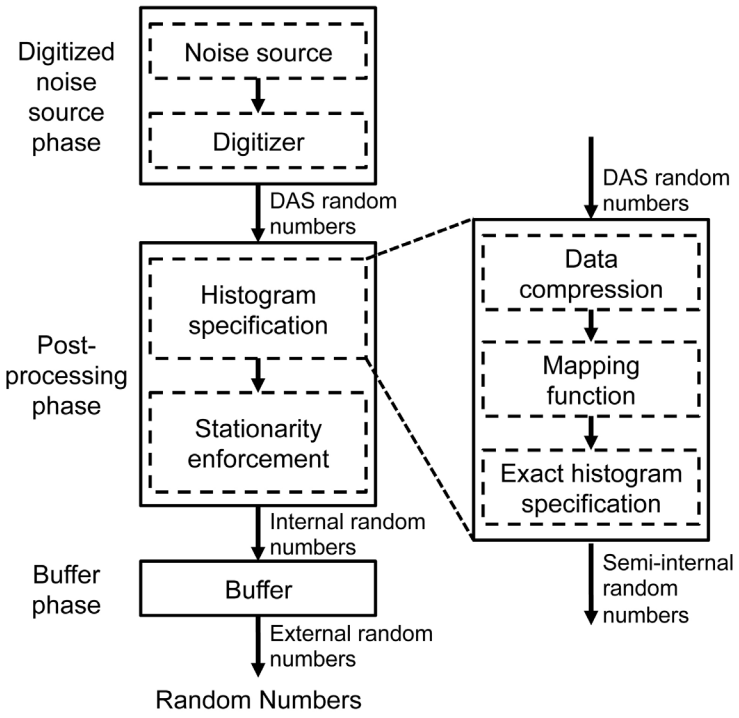

Figure 4. The architecture of a TRNG.

4. Proposed Algorithm: The Stationary Enforcement Block

Figure 5 illustrates the stationarity enforcement block. In this block, the s.i. random numbers, *x*, obtained from the histogram specification block are the input to the stationarity enforcement block in **Figure 4**. The s.i. random numbers may have small traces of periodicity not corrected by the histogram specification block, however, the histogram specification block guarantees *x* to follow an uniform distribution. **Figures 6(a)-(f)** show an example of DAS random numbers and semi-random numbers, including its p.m.f. and autocorrelation.

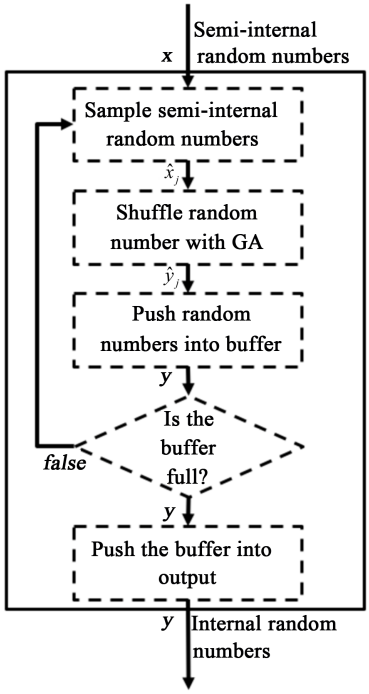

Figure 5. The architecture of the stationarity enforcement block.

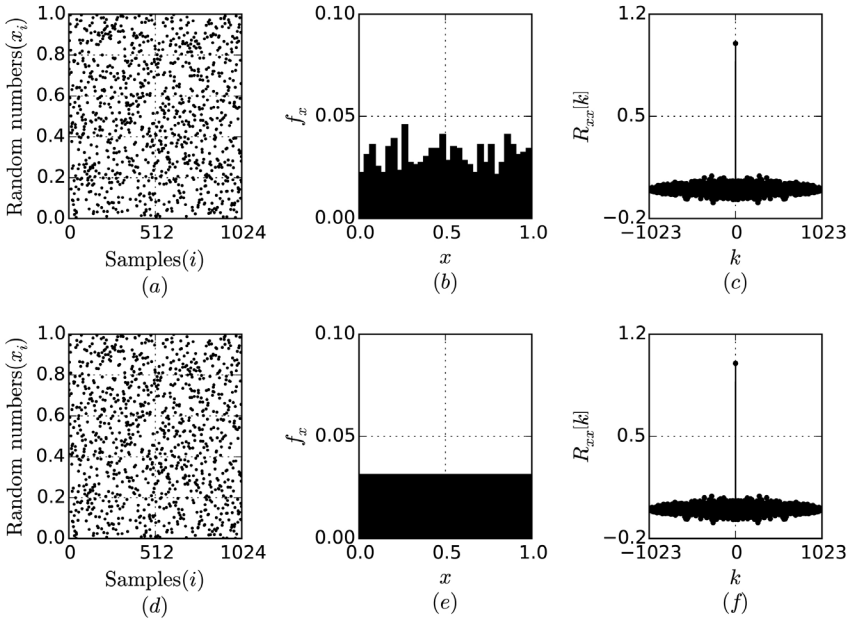

Figure 6. DAS Random numbers (a); its probabilistic mass function (b) and autocorrelation (c); Semi-internal random numbers (d); its probabilistic mass function (e) and autocorrelation (f).

In **Figure 5**, each block is considered as one phase (or step) in the proposed iterative algorithm.

4.1. Sampling semi-internal random numbers: As previously defined, x exhibits $\mathcal{U}(0,1)$, an uniform distribution. x will be consider to have a size of N. And $\hat{x}_j$ will be the j-th segment of size M, that is sampled from x, such that the concatenation of all $\hat{x}_j$ form x.

4.2. Shuffling random numbers with GA: In this stage, x_m is passed through a genetic algorithm (GA) to alter its sequence order, which in consequence will have a major impact on the power spectral density. We explain our parallel GA implementation in **Algorithm 1**, which relies on the data non-dependencies to avoid synchronization as much as possible. Given the nature of GA, parallelism can be exploited at a coarse grain level. In this algorithm, we focus on removing the data dependencies, so a large number of threads can be launch without a synchronization constraint, allowing a finer-grain parallelism.

A traditional implementation of permutations-based GA involves generating the parents population of random numbers, followed by calculating the permutation indexes. For simplicity, the parents populations are sorted so the crossover operations are consistent among parents. A drawback of this is the need of unsorting the random numbers before evaluating them. Sorting, calculating the permutation indexes and unsorting them can take from $M\log_2(M)+M$ operations at its best, and in its worst case $M\log_2^2(M)+M$ operations. On the other hand, our implementation involves generating the populations of permutation indexes representing the parents. This approach avoids sharing values between parents, instead, it only shares the indexes transferring the order pattern to the children. This approach only involves M operations.

The idea behind this algorithm is presented in **Algorithm 1**. The sequence order or indexes will be considered as a chromosome solution. As well, we introduce the idea of having an external population that migrates part of its genetic content to the pool, avoiding stagnation. Since, its parallel implementation takes advantage of the data independence, every thread holds a unique id, *myid* , and keeps updating a shared array, Th_{done}, in which only one memory position is accessed by a thread. In the following, we will explain all the processes that are used in **Algorithm 1**.

- **Selection**: This procedure selects the pool of permutation indexes. At this stage, the intention is to sample and select a group of permutation indexes that will be later used on the mating process inside the GA. The selec-

Algorithm 1 Parallel Permutation-based Genetic Algorithm

P_{pool},	Ch_{pool},	Ext_{pool}	are the parents, children and external populations.
n_P,	n_{Ch},	n_{Ext}	are the parents, children and external population size. This are inputs
P_{val}			is the parents pool fitness score vectors.
x_{Best}			is the best solution.
x_{Ext}			is the external random numbers.
σ_{Best}			is the best solution fitness score.
σ_0			is the target fitness score, also an input.
Th_{done}			is a Shared vector among threads.
$myid$			is the thread id, unique to each thread.

Require:

In parallel do:

1: $\mathrm{Th}_{done}[myid] = false$
2: $P_{pool} = \textbf{Selection}(n_P, M)$
3: $P_{val} = \textbf{Evaluation}(P_{pool}, x_m)$
4: $\sigma_{Best} = \mathrm{Min}(P_{val})$
5: $x_{Best} = x_m[P_{pool}[\mathrm{ArgMin}(P_{val})]]$
6: **while** $\sigma_{Best} > \sigma_0 \& \textbf{Any}(\mathrm{Th}_{done}) = false$ **do**
7: $\quad Ch_{pool} = \textbf{CrossOver}(P_{pool}, n_{Ch})$
8: $\quad Ch_{pool} = \textbf{Mutation}(Ch_{pool})$
9: $\quad x_{Ext} \leftarrow M$ new random numbers
10: $\quad Ext_{pool} = \textbf{Selection}(n_{Ext}, M)$
11: $\quad \sigma_{Best}, x_{Best}, P_{pool}, P_{val} = \textbf{Replacement}(P_{pool}, Ch_{pool}, Ext_{pool}, x_{Ext}, x_{Best})$
12: **end while**
13: $\mathrm{Th}_{done}[myid] = true$
14: **if** $\sigma_{Best} \leq \sigma_0$ **then return** x_{Best}
15: **end if**

tion procedure is illustrated in **Algorithm 2**. We implement the Fisher-Yates shuffle algorithm due to its uniformity among picking a random permutation and its generation speed [30]. So given that *n* is the number of vectors and *M* is the length of a vector, this algorithm will return a pool of permutation indexes, *a*.

- **Evaluation**: This process is focused on evaluating a pool of permutation indexes based on calculating the standard deviation of the power spectral density from Equation (3), as is detailed in **Algorithm 3**.

- **Cross Over**: This process is focused on the cross over operation. The Order 1 cross over is a simple permutation based crossover that connects one group of indexes from one parent with the remaining indexes of the other parent. The process is showed in detail in **Algorithm 4**.

- **Mutation**: This process is focused on the mutation operation. The exchange mutation, as in **Algorithm 5** is a simple technique that only 5% of the times it selects randomly two indexes and swaps their positions.

- **Replacement**: This process is dedicated to applying elitism by selecting the best sets of indexes based on their fitness values [35] [36]. It also updates the pool of parents with the best sets possible. This process takes on consideration the pool of children and the pool of externals, which have a key purpose for escaping local minimas or stagnation over the fitness function. The replacement process is described further detail in **Algorithm 6**.

Algorithm 2 Selection method

a is a pool of size n vectors of length M
a_k is the k-th indexes vector, which is initialized with ascending integers from 0 to $M-1$
n, M are inputs

Require:

for $k = 0$ **to** $n-1$ **do**
  for $i = 0$ **to** $M-2$ **do** ▷ Fisher-Yates Shuffle Algorithm
    $j \leftarrow$ random integer number from $\{0, 1, \cdots, M-1\}$
    swap $a_k[i]$ and $a_k[i+j]$
  end for
end for
return a

Algorithm 3 Evaluation method

a is a vector of the pool fitness score
a_k is the k-th element
x_m is the segment random numbers
b is the pool of indexes of size n by M
x_m, b are inputs

Require:

for $k = 0$ **to** $n-1$ **do**
  $x_a \leftarrow x_m[b_k]$
  $a_k \leftarrow$ st.dev. of $S_{x_a x_a}$ ▷ From Eq.(3)
end for
return a

Algorithm 4 Order 1 Cross Over method

a is the pool of vectors, the pool has a size of n, and vector a size of M
a_k is the k-th vector
a, n are inputs
P_A, P_B will be referred as the parent A, and parent B
Pos_0, Pos_1 will be referred as partition point 0, and point 1
$A = \{A_0, A_1, A_2\}$ the child vector product of the cross over operation

Require:

for $k = 0$ **to** $2 * n - 1$ **in steps of** 2 **do**
  $P_A \leftarrow a_k$
  $P_B \leftarrow a_{k+1}$
  $Pos_0 \leftarrow$ random number from 0 to $M-2$
  $Pos_1 \leftarrow$ random number from Pos_0 to $M-1$
  $A_1 \leftarrow P_A[Pos_0 : Pos_1]$
  $A_0 \leftarrow$ only the first Pos_0 elements from $P_B \setminus A_1$
  $A_2 \leftarrow$ elements from $P_B \setminus \{A_0 \parallel A_1\}$
  $a_k \leftarrow A_0 \parallel A_1 \parallel A_2$
end for
return a

Algorithm 5 Exchange Mutation method

a	is the pool of vectors, the pool has a size of n, and vector a size of M
a_k	is the k-th vector
a	is an input
Pos_0, Pos_1	will be referred as partition point 0, and point 1

Require:

1: **for** $k = 0$ **to** $n - 1$ **do**
2: $Pos_0 \leftarrow$ random number from 0 to $M - 1$
3: $Pos_1 \leftarrow$ random number from 0 to $M - 1$
4: $Pr \leftarrow$ random number from 0 to 1
5: **if** $Pr < 0.05$ **then**
6: swap $a_k[Pos_0]$ and $a_k[Pos_1]$
7: **end if**
8: **end for**
9: **return** a

Algorithm 6 Replacement method

$P_{pool}, Ch_{pool}, Ext_{pool}$	are the parents, children and external pool of indexes.
$P_{val}, Ch_{val}, Ext_{val}$	are the parents, children and external pool score vectors.
n_P	is the number of parents in P_{pool}

Require:

1: $P_{val} = \text{Evaluate}(P_{pool}, x_{Best})$
2: $Ch_{val} = \text{Evaluate}(Ch_{val}, x_{Best})$
3: $Ext_{val} = \text{Evaluate}(Ext_{val}, x_{Ext})$
4: $\sigma_P = \text{Min}(P_{val})$
5: $\sigma_{Ch} = \text{Min}(Ch_{val})$
6: $\sigma_{Ext} = \text{Min}(Ext_{val})$
7: $P_{pool} \leftarrow$ select only the best n_P indexes from $\{P_{pool}, Ch_{pool}, Ext_{pool}\}$ using $\{P_{val}, Ch_{val}, Ext_{val}\}$
8: $P_{val} \leftarrow$ select only the best n_P scores from $\{P_{val}, Ch_{val}, Ext_{val}\}$
9: $x_{Best} \leftarrow$ select the best indexes from $\{P_{pool}, Ch_{pool}, Ext_{pool}\}$ using either x_{Best} or x_{Ext}
10: $\sigma_{Best} \leftarrow$ the best value from $\{\sigma_P, \sigma_{Ch}, \sigma_{Ext}\}$
11: **return** $\sigma_{Best}, x_{Best}, P_{pool}, P_{val}$

- **Termination condition**: The GA exits when the fitness is $\sigma_{Best} \leq \sigma_0$. If the condition is successful, the algorithm proceeds to update $Th_{done}[myid] = true$, and in consequence any other thread running will stop by the end of its iteration as shown in 1 at line 6; then the algorithm returns the best sequence of random numbers found, x_{Best} or the answer for the *j*-th segment, $\hat{y}_j$. Otherwise, if the condition is unsuccessful the algorithm loops back to the cross over stage on the GA, line 7.

4.3-4.5. **Handling the output buffer**: In **Figure 5**, the output of the GA in 4.2, $\hat{y}_j$, is pushed into the buffer *y*. Then if the buffer is not full yet, it will loop back the algorithm to sample the semi-internal random numbers at step 4.1. Otherwise *y* is pushed out as the internal random numbers.

5. Evaluation Methodology

In this section, we introduce a group of tests to show two aspects: first, the characteristics of the random number generation aided by the GA post-processing; and second, the strengths and weaknesses of the algorithm against the SmallCrush battery test [23] when changing the segment size . The results of these tests are presented in Section 6.

5.1. **Characteristics of a GA post-processing stage**: In this evaluation, we follow the *m* segment of random numbers analyzing step by step the change on x_m, its autocorrelation or $R_{x_m x_m}$ from Equation (4), its power spectral density or $S_{x_m x_m}$ from (3) and its distribution described by its probability mass function or f_{x_m}. The evaluation is iterated over a loop from step 6 - 12 in **Algorithm 1** or generation, until the standard deviation of the power spectral density, or σ_{Best}, is smaller than the user specified quality level, σ_0.

Having x_m with $N = 1024$ samples, the algorithm is evaluated on its convergence capability given an error level (target) of $\sigma_0 = 0.415$. The parents, children and external populations size are set to $M = 32$ each. With the intention of explaining the behavior of the algorithm, we present a snapshot at three different generations; at

the first generation or $g_{ini} = 0$, in the middle or $g_{mid} = 300$ and then in the last generation or g_{end}, which will correspond with the output.

For the GA parameters, the mutation ratio was set to 0.05% or 5% and the elitism was applied at the replacement stage to all individuals, letting survive only the best fitted individuals, which at the next generation will be considered as the parents pool.

5.2. **SmallCrush**: This evaluation is intended to highlight the weakness and strengths of the post-processing. It evaluates a vast group of configurations as shown in **Table 1**. The evaluations are done considering x_m with $N = 1024$ samples, an error level (target) of $\sigma_0 = 0.45$. The parents, children and external populations size are set to $M = 32$ each. SmallCrush is a battery test that forms part of the statistical package TestU01 [23]. This a modern test package that includes most available test packages, surpassing by flexibility and extension popular ones like DIEHARD [37] and NIST [38]. As a standard, TestU01 uses statistical testing where the test outcome or *p-value*, fails if it is outside of the interval $[0.001, 0.999]$. In essence SmallCrush consist of 10 tests: the birthday spacing test [39], the collision test, the gap test, the simplified poker test, the coupon collector test, the maximum-of-t test [30], a weight distribution test proposed by Matsumoto and Kurita [40], the rank of a random binary matrix test [39], two tests of independence between the Hamming weights of successive blocks of bits [41], and various tests based on random walk over the set of integers [23].

- *The Birthday Spacing Test*. The birthday spacing test checks the separation between numbers by mapping them to points in a cube. It checks that the number of collisions between their the spacing follows a Poisson distribution.

- *The Collision Test*. This test checks for n-dimensional uniformity, by checking the number of collisions per subdivision of the interval $[0,1]$ is uniformly distributed.

- *The Gap Test*. This test checks for the number of times that random number within $[0,1)$ fall outside of a expected interval. It applies chi-square testing between the observed and expected number of observations.

- *The Simplified Poker Test*. This test checks for the number of different integers that are generated, it applies chi-square testing between the observed and expected number of observations.

- *The Coupon Collector Test*. This test checks over a group of random number integers for integers without collision, unique numbers, it applies chi-square testing between the observed and expected number of observations among the group of random numbers.

- *The Maximum-of-t Test*. This test checks for the number of times that the maximum random number within $[0,1)$ follows an exponential distribution via chi-square and Anderson-Darling test.

- *The Weight Distribution Test*. This test takes a group of uniform distributions; then it computes a binomial distribution and it gets compared via chi-square against the expected distribution.

- *The Rank of a Random Binary Matrix Test*. This test generate a binary matrix with uniform random numbers and then it calculates the rank of the matrix, or the number of linearly independent rows. Then, the probability distribution is compared to a known distribution via chi-square.

- *The Independence Test between Hamming Weights of successive blocks*. This involves two test that check for independence among Hamming Weights. The first test considers only the most significant bits of random numbers in a block and its corresponding Hamming Weight, then the Hamming Weights are arrange in successive pairs that later are count for the number of possibilities and compared against expected values via a chi-square test. The second test, considers mapping the counting from the first test into a 2-dimensional plane, where it is segmented into 4 blocks that its values are related to an expected value. Therefore, via a chi-square test the observed quantities can be compared to the expected values.

- *The Random Walk Test*. This test performs a random walk over integers, having an equal probability to move to right or left during the walk. So a group of statistics and the distribution that its position at the end of the test, both, are compared against their theoretical positions via a chi-square test.

Table 1. Small crush test configurations.

Number of Processors	Segment Size						
2							
4	64	16384	17408	32768	49152	65536	1048576
8							

6. Results

In this section we present the results according to the evaluation methodology presented in Section 5. The results are presented in the following.

1. **Results from the Characteristics of a GA post-processing stage**:

As observed on **Figure 7**, the standard deviation of the power spectral density, or σ_{Best}, always tends to decrease as the algorithm goes forward on the generations. This behavior is expected. It is caused by the elitism implementation on **Algorithm 1**, and in the replacement method in **Algorithm 6**. The elitism ensures to either decrease or stagnate σ_{Best}, keeping only the best individuals available for the next generation.

In **Figure 7** in a didactic manner, we selected three points to compare the progress on the distribution of the power spectral density. Its respective shape on the autocorrelation and the shape of the probability mass function and the points distribution pattern. In **Figure 8** at generation $g_{ini} = 0$, the random numbers show some traces of periodicity, that can be seen on the peaks with a lag different than 0 on the autocorrelation at **Figure 8(c)**. This is also shown on the power spectral density, **Figure 8(d)**, as points that overshoot more than 5.0. These random numbers, at generation 0, are already of an outstanding good quality, $\sigma_{Best} = 0.4876$, but as shown in **Figure 8(c)** and **Figure 8(d)** there is still room for improvement. Nevertheless, due to the fact that we only manipulate the order and not the magnitudes of those numbers, the probability mass function remains uniform across all generations.

In **Figure 9(d)** at generation $g_{mid} = 300$, the overshoots on the power spectral density converge to less than 5.0, still with traces of periodicity on the autocorrelation, in **Figure 9(c)**, but closer to the solution. At this

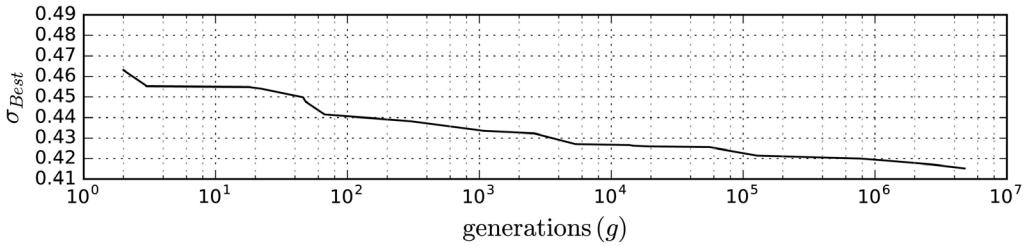

Figure 7. Standard deviation of the Power spectral density plotted against the change on generations.

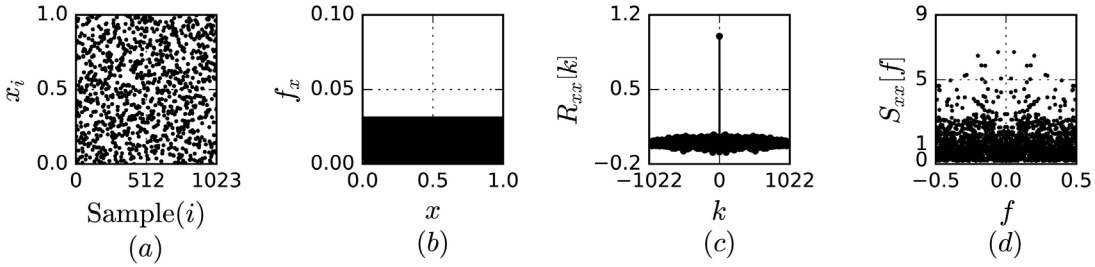

Figure 8. (a) Random numbers at generation 0; (b) its probability mass function; (c) the autocorrelation; (d) the power spectral density.

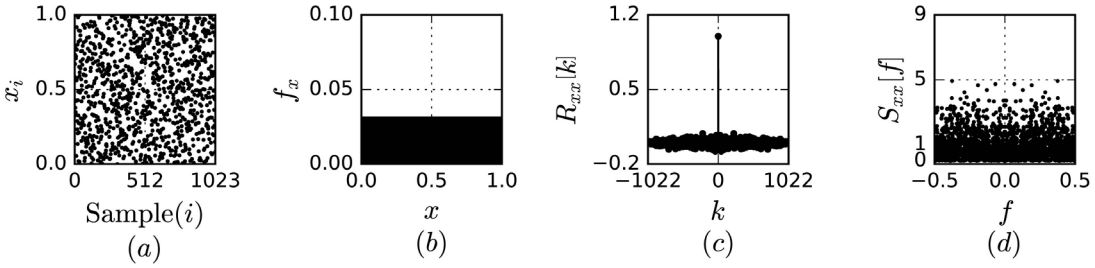

Figure 9. (a) Random numbers at generation 300; (b) its probability mass function; (c) the autocorrelation; (d) the power spectral density.

generation with $\sigma_{Best} = 0.4382$, the distribution pattern of the points in **Figure 9(a)** is sparser than the one in **Figure 8(a)**.

The algorithm converges until generation $g_{end} = 4779992$, where the power spectral density is concentrated in its majority below 3.0 with a $\sigma_{Best} = 0.4150$, this can be reviewed on **Figure 10(d)**. As well, the improvement on the power spectral density is reflected on the autocorrelation as closer appearance of an impulse shape, **Figure 10(c)**. But the overall benefit of the algorithm can be observed on **Figure 10(a)**, where the points are spread covering the majority of the space, with a less visible agglomerates or darker spots as in **Figure 9(a)** and **Figure 8(a)**.

As we have reviewed, the underlining idea of the **Algorithm 1** is to benefit from the interactions of the pools of parents, children, and externals. To review in more detail these interactions, we look at the effects of those distribution in the following two examples.

It is well known, that one of the major drawbacks of elitism is the reduction of genetic content over the population which perpetuates the stagnation of the fitness function. But as well, elitism is characterized for a smooth fitness function. So in order to compensate for the lack of genetic content the external pool is introduced. In **Figure 11** over an example, we can observe the distributions of the children pool scores overlapping the external pool scores in its majority. At each iteration, the minimum among the two distribution is selected as the best, as it is described on **Algorithm 6**. Here, the external pool becomes another set of points over the solution space, uniformly distributed, meanwhile the children pool is distributed nearby the parents distribution, since it keeps traces of the permutation orders, described on **Algorithm 2**, **Algorithm 4**, and **Algorithm 5**.

In contrary to the example on **Figure 11**, we also present in another example, **Figure 12**, the effects of the

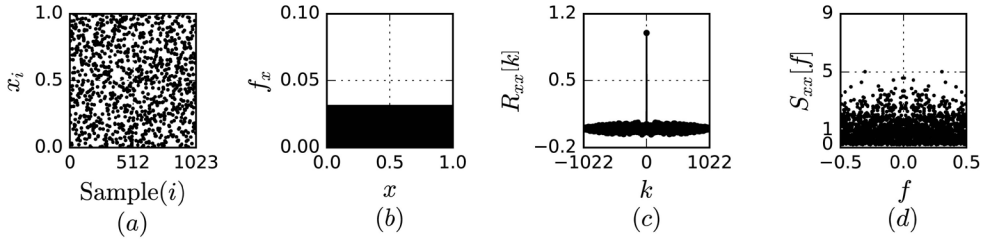

Figure 10. (a) Random numbers at generation 4779992, (b) its probability mass function, (c) the autocorrelation, (d) the power spectral density.

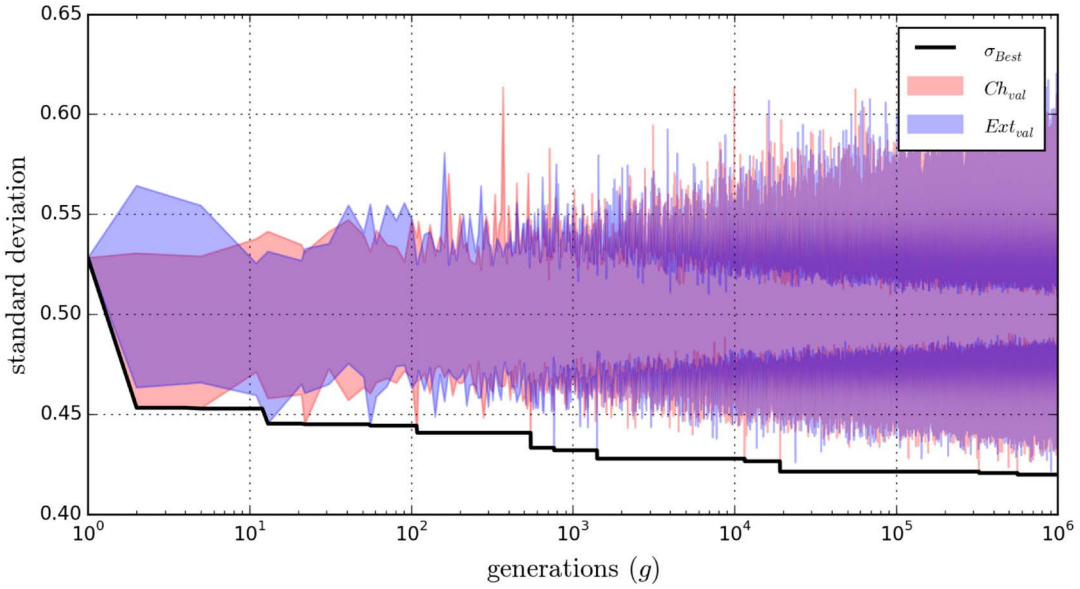

Figure 11. Standard deviation of the Power spectral density from the parents, children and external pool plotted against the change on generations.

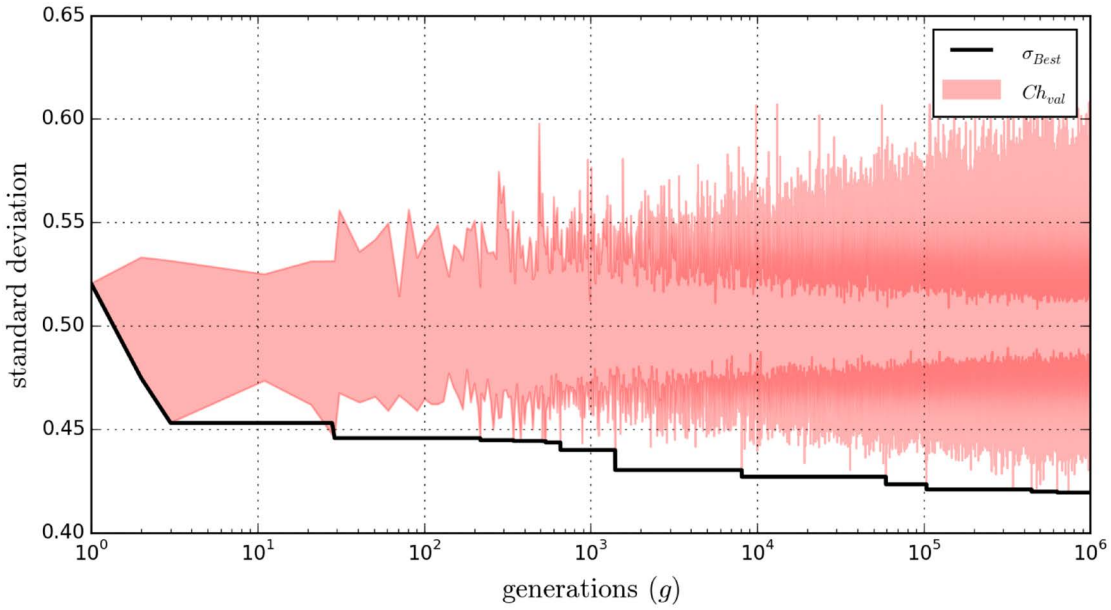

Figure 12. Standard deviation of the Power spectral density from the parents and children pool without external pool plotted against the change on generations.

lack of an external pool. Here, the fitness function tends to stagnate for longer periods and also is noted that children will not impact σ_{Best} as much as the case of **Figure 11**, since the sample of the solution space is smaller. Consequently, this evolves a lack of genetic diversity and the proliferation of dominant traits on the children observed along different generations as stagnation.

Balancing between the two factors is beyond the scope of this paper, but a good inside on which variables contribute to modifying the fitness curve are the pool size and the children to external pool ratio.

2. **Results from SmallCrush Test**

In order to make easier the review of the results, given that there are 10 SmallCrush evaluations with the 3 different configurations on the numbers of processors and 7 different segment sizes, we present a coded color map highlighting the *p-value* in **Figures 13-15**. In a SmallCrush test, the test is passed only if all the 10 evaluations have a *p-value* within the range of $[0.001, 0.999]$.

The first configuration is using 2 processors, where the coded color results are presented at different segment sizes. in **Figure 13**. The evaluation of SmallCrush in this configuration is close to pass the battery test but it fails only in a test, either Maximum-of-t test or the Random Walks. Independently of this, the battery test successfully passes all the tests when the segment size is 17408.

The second configuration is using 4 processors and the results are presented in **Figure 14**. In this configuration, all the test failed either at Maximum-of-t or the Random Walks, in a similar ways as in **Figure 13**. The results tend to fail exclusively on one the test only. But the only evaluation at which the failing test is close to passing is when the segment size is 17,408.

The third configurations uses only 8 processors, the results are shown in **Figure 15**. In a similar manner, the failed evaluations are exclusive of each other, between Maximum-of-t and Random Walks. But the only evaluation that passes has a segment size of 17,408.

Recapitulating the overall SmallCrush tests, the segment size plays a key role on the quality of the tests results. As well, only Maximum-of-t and Random Walks fail among different segment sizes and configurations with different number of processors. The Maximum-of-t test focus on extracting the maximum of a sample set of random values, while the maximums follow an exponential distribution. Therefore, failing Maximum-of-t is due to inherit statistical properties of nearly i.i.d. uniform distributions. This improves at certain segment sizes, cause it aligns with the sample set size of the Maximum-of-t test. The Random Walks rely on the properties found on Markov chains, where the probability of the current position depends on the previous, This means that failing Random Walks is due to a lack of consistency among continuous segments. This evaluation passes when the segment size fits the sampling size used over the random walk so this connecting property is satisfied.

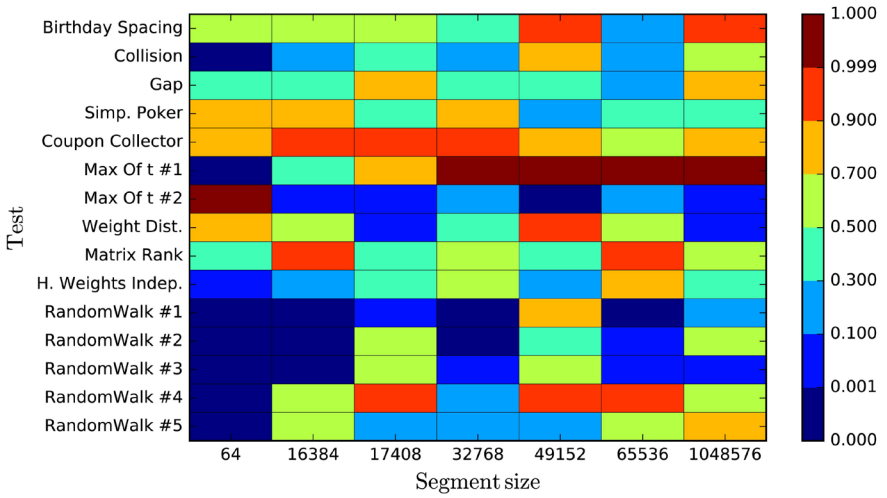

Figure 13. SmallCrush results from TestU01 battery test using 2 processors.

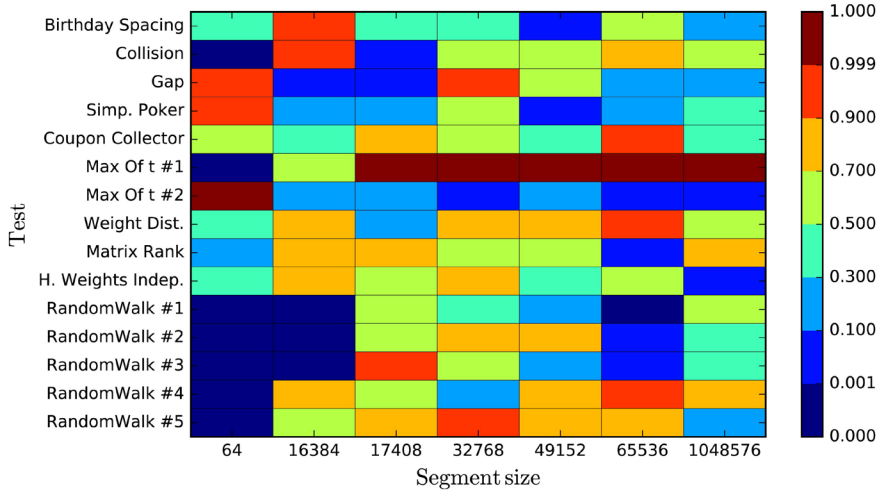

Figure 14. SmallCrush results from TestU01 battery test using 4 processors.

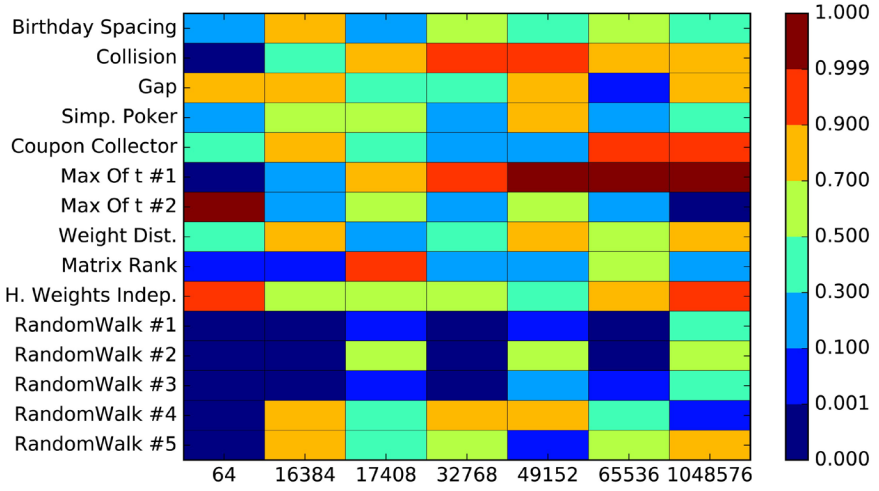

Figure 15. SmallCrush results from TestU01 battery test using 8 processors.

So far, the segment size of 17,408 random numbers is one sub-optimal solution for centering the evaluations, it is almost independent of the number of processors. This condition seems to remain constant as the number of processors is increased, since passing the test is connected to the consistency of σ_{Best} among segments which is aided by increasing the sampling size. This is benefited by the parallelism exposed on GA as in the **Algorithm 1**.

7. Conclusion

The use of a post-procesing stage for TRNGs using GA inteded for seed generation proves to satisfy the specified quality level within a desirable number of generations. We have presented the theoretical foundations that guarantee the algorithm convergence to a nearby point of σ_0, as well as, the analysis of a GA approach aided to avoid stagnation and its respective complexity and performance justification. In addition, guidelines have been presented showing the influence different parameters that connect the statistical properties among the random numbers and the task decomposition from the post-processing algorithm. From this parameters, we observed that the segment size in the post-processing is key for centering the quality of the random numbers. For future work, we will extend the research towards meta-heuristics that target statistical properties connecting segments of random numbers.

References

[1] Sumter, L. (2010) Cloud Computing: Security Risk. *Proceedings of the ACM* 48*th Annual Southeast Regional Conference*, Oxford, MS, 15-17 April 2010, 112:1-112:4.

[2] Mowbray, M. and Pearson, S. (2009) A Client-Based Privacy Manager for Cloud Computing. *Proceedings of the ACM* 4*th International ICST Conference on Communication System Software and Middleware*, Dublin, 15-19 June 2009, 5. http://dx.doi.org/10.1145/1621890.1621897

[3] Lin, D. and Squicciarini, A. (2010) Data Protection Models for Service Provisioning in the Cloud. *Proceedings of the* 15*th ACM Symposium on Access Control Models and Technologies*, Pittsburgh, 9-11 June, 183-192. http://dx.doi.org/10.1145/1809842.1809872

[4] Lombardi, F. and Di Pietro, R. (2010) Transparent Security for Cloud. *Proceedings of the ACM Symposium on Applied Computing*, Sierre, 22-26 March 2010, 414-415. http://dx.doi.org/10.1145/1774088.1774176

[5] Bleikertz, S., Schunter, M., Probst, C.W., Pendarakis, D. and Eriksson, K. (2010) Security Audits of Multi-Tier Virtual Infrastructures in Public Infrastructure Clouds. *Proceedings of the* 2010 *ACM Workshop on Cloud Computing Security Workshop*, Chicago, 4-8 October 2010, 93-102.

[6] Kerrigan, B. and Chen, Y. (2012) A Study of Entropy Sources in Cloud Computers: Random Number Generation on Cloud Hosts. Springer, St. Petersburg.

[7] Vuillemin, T., Goichon, F., Lauradoux, C. and Salagnac, G. (2012) Entropy Transfers in the Linux Random Number Generator. Technical Report, Research Report 8060, INRIA, Montbonnot-Saint-Martin.

[8] Almorsy, M., Grundy, J. and Müller, I. (2010) An Analysis of the Cloud Computing Security Problem. *Proceedings of APSEC* 2010 *Cloud Workshop*, Sydney, 30 November 2010, 6 p.

[9] Grobauer, B., Walloschek, T. and Stocker, E. (2011) Understanding Cloud Computing Vulnerabilities. *IEEE Security & Privacy*, **9**, 50-57.

[10] Rukhin, A., Soto, J., Nechvatal, J., Barker, E., Leigh, S., Levenson, M., Banks, D., Heckert, A., Dray, J., Vo, S., Rukhin, A., Soto, J., Smid, M., Leigh, S., Vangel, M., Heckert, A., Dray, J. and Bassham Iii, L.E. (2010) Statistical Test Suite for Random and Pseudo Random Number Generators for Cryptographic Applications. NIST Special Publication, Gaithersburg.

[11] Barker, E. and Kelsey, J. (2012) NIST DRAFT Special Publication 800-90b Recommendation for the Entropy Sources Used for Random Bit Generation. NIST Special Publications, Gaithersburg. http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf

[12] Easter, R.J., French, C. and Gallagher, P. (2012) Annex c: Approved Random Number Generators for _ps pub 140-2, Security Requirements for Cryptographic Modules. NIST, Gaithersburg, February.

[13] Lecuyer, P. (2012) Random Number Generation. Springer. http://dx.doi.org/10.1007/978-3-642-21551-3_3

[14] Bucci, M., Germani, L., Luzzi, R., Trifiletti, A. and Varanonuovo, M. (2003) A High-Speed Oscillator-Based Truly Random Number Source for Cryptographic Applications on a Smart Card IC. *IEEE Transactions on Computers*, **52**, 403-409.

[15] Colesa, A., Tudoran, R. and Banescu, S. (2008) Software Random Number Generation Based on Race Conditions.

IEEE 10*th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, 26-29 September 2008, 439-444. http://dx.doi.org/10.1109/synasc.2008.36

[16] Fischer, V. and Drutarovsky, M. (2003) True Random Number Generator Embedded in Reconfigurable Hardware. *Cryptographic Hardware and Embedded Systems-CHES* 2002, Springer, 415-430.

[17] Epstein, M., Hars, L., Krasinski, R., Rosner, M. and Zheng, H. (2003) Design and Implementation of a True Random Number Generator Based on Digital Circuit Artifacts. *Cryptographic Hardware and Embedded Systems-CHES* 2003, Springer, 152-165. http://dx.doi.org/10.1007/978-3-540-45238-6_13

[18] Chen, I.-T. (2013) Random Numbers Generated from Audio and Video Sources. Mathematical Problems in Engineering.

[19] Jun, B. and Kocher, P. (1999) The Intel Random Number Generator. Cryptography Research Inc. White Paper, San Francisco.

[20] Hamburg, M., Kocher, P. and Marson, M.E. (2012) Analysis of Intels Ivy Bridge Digital Random Number Generator. http://www.cryptography.com/public/pdf/IntelTRNGReport20120312.pdf

[21] Chan, J.J.M., Sharma, B., Lv, J., Thomas, G., Thulasiram, R. and Thulasiraman, P. (2011) True Random Number Generator Using GPUs and Histogram Equalization Techniques. *IEEE* 13*th International Conference on High Performance Computing and Communications*, Banff, AB, 2-4 September 2011, 161-170.

[22] Chan, J.J.M., Thulasiraman, P., Thomas, G. and Thulasiram, R. (2015) Stationarity Enforcement of Accelerator Based TRNG by Genetic Algorithm. *IEEE Trustcom/BigDataSE/ISPA*, Helsinki, 20-22 August 2015, Volume 1, 1122-1128.

[23] L'Ecuyer, P. and Simard, R. (2007) Testu01: A Software Library in ANSI C for Empirical Testing of Random Number Generators. Software User's Guide. http://www.iro.umontreal.ca/~lecuyer

[24] Gentle, J.E. (2006) Random Number Generation and Monte Carlo Methods. Springer Science & Business Media, New York.

[25] Knuth, D.E. (2011) Art of Computer Programming, Volumes 1-4A Boxed Set. 3rd Edition, Addison-Wesley Professional, Massachusetts.

[26] Niederreiter, H. (1991) Recent Trends in Random Number and Random Vector Generation. *Annals of Operations Research*, **31**, 323-345.

[27] Box, G.E.P., Jenkins, G.M. and Reinsel, G.C. (2011) Time Series Analysis: Forecasting and Control, Volume 734. John Wiley & Sons, Hoboken.

[28] Proakis, J.G. (2007) Digital Signal Processing: Principles, Algorithms, and Applications, 4/E. Pearson Education, Boston.

[29] Welford, B.P. (1962) Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, **4**, 419-420.

[30] Knuth, D. (1997) The Art of Computer Programming. Volume 2: Semi Numerical Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts.

[31] Augustin, W., Heuveline, V. and Weiss, J.-P. (2009) Optimized Stencil Computation Using In-Place Calculation on Modern Multicore Systems. *Euro-Par* 2009 *Parallel Processing*, *Lecture Notes in Computer Science*, Volume 5704 Springer Berlin Heidelberg, 772-784.

[32] Prabahan Basu, Daniel Rudoy, and Patrick J Wolfe. A non-parametric test for stationarity based on local Fourier analysis. *IEEE International Conference on Acoustics, Speech and Signal Processing*, Taipei, 19-24 April 2009, 3005-3008.

[33] Killmann, W. and Schindler, W. (2001) AIS 31: Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators, Version 3.1. Bundesamt fur Sicherheit in der Informationstechnik (BSI), Bonn.

[34] Hofemeier, G. (2012) Intel Digital Random Number Generator (DRNG) Software Implementation Guide.

[35] Ahn, C.W. and Ramakrishna, R.S. (2003) Elitism-Based Compact Genetic Algorithms. *IEEE Transactions on Evolutionary Computation*, **7**, 367-385.

[36] Reed, P.M., Minsker, B.S. and Goldberg, D.E. (2001) The Practitioners Role in Competent Search and Optimization Using Genetic Algorithms. *Bridging the Gap*, **10**, 97.

[37] Marsaglia, G. (1996) Diehard: A Battery of Tests of Randomness. http://stat.fsu.edu/pub/diehard/

[38] Ruhkin, A.L. (2001) Testing Randomness: A Suite of Statistical Procedures. *Theory of Probability & Its Applications*, **45**, 111-132.

[39] Marsaglia, G. (1985) A Current View of Random Number Generators. In: *Computer Science and Statistics, Sixteenth Symposium on the Interface*, Elsevier Science Publishers, North-Holland, Amsterdam, 3-10.

[40] Matsumoto, M. and Kurita, Y. (1994) Twisted GFSR Generators II. *ACM Transactions on Modelling and Computer Simulation* (*TOMACS*), **4**, 254-266.

[41] L'ecuyer, P. and Simard, R. (1999) Beware of Linear Congruential Generators with Multipliers of the Form a = ±2q ± 2r. *ACM Transactions on Mathematical Software* (*TOMS*), **25**, 367-374.