UNIVERSITÀ DI PADOVA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

PARIRANDOM

Entropy distribution in a p2p network

(Distribuzione dell'entropia in una rete p2p)

RELATORE
Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE
Ing. Michele Bonazza

TESI DI LAUREA DI
Nicola Moretti
Matr. N. 622066

Anno Accademico 2011/2012

To family,
Francesca,
friends,
everybody who supported me and my work
during this fundamental period of my life.

Welcome the caos, as order did not work.
(Karl Kraus)

An updated version of this document is available at https://github.com/hanicker/Tesi.

Contents

Abstract

Entropy generation in a machine has always been an area of great interest, especially given its practical applications. A good source of entropy is a fundamental ingredient in many simulations of large physical systems, robust cryptographic operations and non-parametric statistical methods.

Current software entropy generators bring entropy to a level which is not always suitable for its final use. There is a particular trade-off between the quality of entropy (see chapter 1.2.2) and the speed needed to generate it.

We propose a novel method to improve the quality of entropy generators that leverages the ever growing phenomenon of Peer-to-peer networks. PariRandom is a pseudo-random number generation system that may be used to extend all other existing Pseudo Random Number Generation (PRNG) algorithms, ensuring they have an equal or greater level of entropy. An important aspect of our system is that it does not noticeably increase the underlying traffic, "piggybacking" instead on packets that would be transmitted anyway. The theoretical and experimental results demonstrate an increase in performance, which on a wide-scale, comes close to the performance achieved by hardware entropy generators. Moreover, they guarantee resistance to every kind of attack by malicious nodes in the network.

Chapter 1

Introduction and related work

This report describes the design of PariRandom, a random number generating system that operates on a P2P network to improve the quality of the entropy available to its nodes. The system was developed as part of the PariPari project, a new P2P application designed to offer in a single, modular solution many mainstream internet (e-mail, DNS, web hosting, file hosting, IRC chat, etc.) and peer-to-peer services (file sharing, distributed storage, VoIP, distributed calculation, etc.) that will be described in section 1.1 of this chapter. The individual modules (plugins) of PariPari often have to generate random values at good quality and speed. Although individual nodes can generate entropy "on their own", PariRandom can improve the quality of that entropy.

To understand the innovative features of what has been developed, section 1.2 explains the details of the problem of Random Number Generators and the strengths and weaknesses of current solutions.

Chapter 2 presents our solution, PariRandom, followed by a theoretical analysis.

Chapter 3 describes its implementation as a PariPari plugin followed by an introduction about statistical tests for randomness, the description of a PariRandom simulator and, finally, the results obtained by running three different random testing suites on the data generated by this simulator.

Finally, chapter 4 summarizes our results, analyses their significance and looks at directions of future work.

1.1 PariPari

PariPari is a P2P multi-functional application that offers all traditional P2P services (file-sharing, distributed storage, VoIP,...) along with a number of traditionally server-based ones (email hosting, IRC chat, web hosting, NTP,...). It is currently being developed by 60+ students from the University of Padua, but aims at a larger community of developers as soon as it is officially released. It is written in Java, so that it runs on all Operating Systems that have a Java Virtual Machine installed without the need to recompile the code. As a drawback, this comes at the cost of the impossibility to interact with the machine at low levels (e.g. for direct memory management). Moreover, students at the University of Padova are very familiar with Java, and this choice has been done taking that into consideration, to quickly build a larger initial developers base.

1.1.1 Network layout

PariPari is based on a Distributed Hash Table (**DHT**) named *PariDHT*. After a surge of interest in the academic community (e.g. Chord, CAN, Pastry, Kademlia) recent years have seen DHTs adopted in several applications with a vast public (e.g. the popular eMule and Azureus filesharing clients and the JXTA system) While DHTs can be implemented by many different data structures, virtually all the mainstream ones (including all the ones we cited above) function according to a basic scheme that we summarize below.

Roughly speaking, each node in a DHT is assigned a random address in a b-bit ID space (b is chosen sufficiently large, typically 160 or more, to avoid collisions). Some form of distance (pseudo-)metric is defined on this address space (e.g. the XOR metric for Kademlia), so that one can partition the space, for any given node, in the 2^{b-1} addresses in the "other half" of the network, the 2^{b-2} addresses in the same half but in the other quarter, the 2^{b-3} addresses in the same quarter and the other eighth, and so on . Each node then keeps contacts with a small number k of nodes in the other half of the network, k nodes in the other quarter, k in the other eighth and so on (see Figure 1.1.1). Theoretically $k = 1$ would suffice, but in practice some redundancy is introduced to provide robustness and typically $5 \leq k \leq 20$ is used. Of course less than k nodes might be present in some of the smallest regions, in which case all the nodes in any such region are kept as contacts.

Each resource r (e.g. a file) is also mapped into the same address space using a pseudorandom hash of the keyword(s) that will be used to locate it; information about how to reach it (e.g. the IP of the machine from which it can be accessed) is stored in the node $v(r)$ closest to it in the address space. To locate $v(r)$ — whether to retrieve the information on how to access r, or to store it in the first place — a

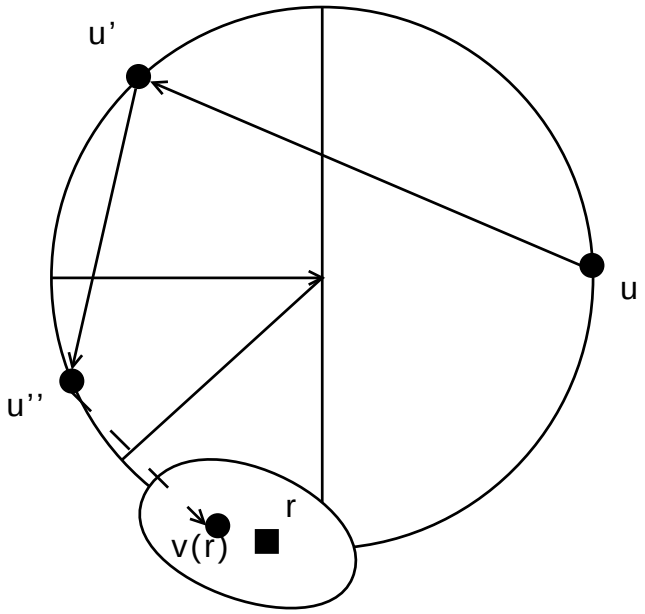

Figure 1.1.1: The search structure in a typical DHT.

node u will forward the query to the node u', among its contacts, closest to r in the address space. In the worst case, u' will be in the other half of the network — but it will certainly be in the same half as r. It can then forward the query to another node u'' that will certainly be in the same quarter as r — and so on, until $v(r)$ is reached in a number of steps with high probability logarithmic in the size of the network.

1.1.2 Host structure

PariPari hosts are based on a plug-in architecture. Users can decide which plugins to load, and can load them at runtime. Plugins cannot communicate directly with each other, or with the outer world. Every request must pass through the Core. For security reasons and for a better overall performance, disk and network management is ruled by two *inner circle* plugins: Storage and Connectivity. From a user's point of view, this means that malicious plugins could never write on disk (or on Sockets) without having the user explicitly allowing it to do so. At the same time, a fair distribution of resources among plugins is guaranteed, improving user experience.

Figure 1.1.2, from [32], shows how communications between plugins residing in different hosts take place. Not being able to communicate directly with their counterparts, plugins ask to Connectivity modules residing in the same hosts as they do for Sockets (or Server Sockets, when they need to listen to incoming requests). Sockets are *bandwidth-limited*, meaning that Connectivity allocates a different amount of bandwidth to each plugin, allowing cohabitation of bandwidth-eager modules.

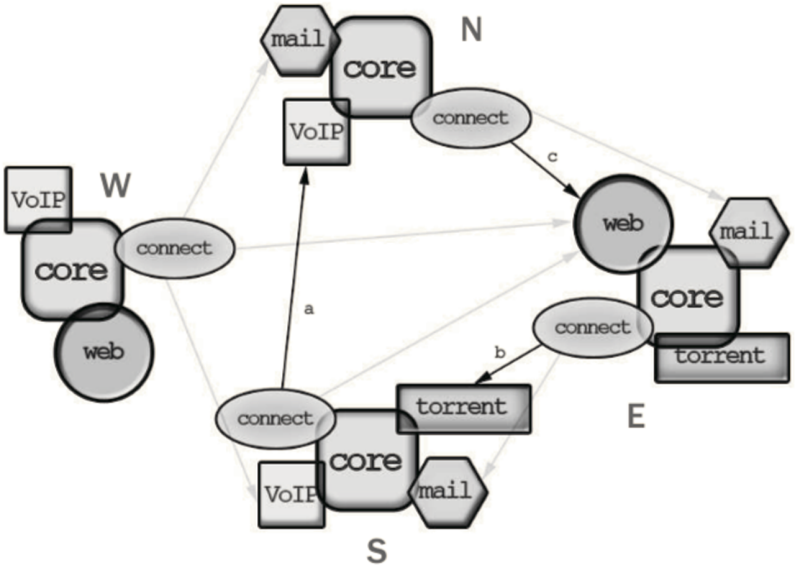

Figure 1.1.2: Four hypothetical hosts in PariPari. Three communications are portrayed as examples. Connection `a` represents the beginning of a VoIP session between hosts `S` and `N`: the VoIP module of host `S` asks a Socket to the Connectivity module running in its same host. Connectivity then opens a Socket with host `N`, whose Connectivity module has previously opened a Server Socket for `N`'s VoIP module. Connection `b` represents Torrent plugins of hosts `E` and `S` communicating with each other. Connection `c` represents host `N` requesting a web content to host `E`.

1.2 Random Number Generation

This section explains the theoretical basis of random values and the existing types of random number generators.

Random numbers are very important in many computational procedures. In Monte Carlo methods, random numbers permit sampling of the relevant part of an extremely large space (multi-dimensional space) to determine e.g. the properties of some phenomenon, like molecular dynamics[1]. In games, random numbers ensure a unique and fair (unbiased) experience[2]. An efficient randomization is primarily useful in cryptographic security, e.g. in symmetric and asymmetric keys generation. Based on the entropy level, the security of every connected application may vary considerably[3]. It is then not surprising that Random and Pseudo-Random Number Generation has attracted a vast interest from mathematicians, statisticians and physicists.

Within the *PariPari* project, it becomes important also for other applications, like the security systems related to the support of the *DHT* network or, at a lower level, the extraction of random values from a database.

Subsections 1.2.1 and 1.2.2 respectively investigate the properties of random bits and of sequences of random bits. Subsection 1.2.3 introduces Pseudo-Random Numbers Generators (PRNG), used to speed-up the generation of random values when a fast RNG is not available. Subsection 1.2.4 describes some ways to improve the entropy quality of a string by reducing its length. Finally, subsections 1.2.5 and 1.2.6 explains the two main problems of Random Number Generators.

1.2.1 What is a random bit?

Mathematically, a single toss of a fair coin has an entropy of one bit. If the coin is not fair the entropy is lower (if asked to bet on the next outcome, we would bet preferentially on the most frequent result). In both cases, it's an example of a Bernulli trial.

A random bit is *the outcome of a process whose possible outcomes are 0 and 1 and that has an expected value of* $\frac{1}{2}$. This introduces the notion of unbiased bit, meaning that a bit has the same probability of being 0 or 1.

[1] An extensive list of Monte Carlo methods application is available in [26].

[2] A curious attack is the one being carried out from an american group named Cigital (a security company) against a Texas Hold'em Poker software, after discovering that the deck shuffling algorithm relied on a seed based on the number of seconds passed after midnight.

[3] An example of all is the problem that has plagued OpenSSL packet for Debian systems until 2008, which relied on a key generator with an initial seed limited to a space counting only 32.768 values, that imposed the regeneration of every keys created so far and the revocation of existing ones.

1.2.2 Multiple random bits

The property of being unbiased (described in previous subsection) becomes insufficient to characterize a sequence of random bits stated to be random. In fact, we can produce an unbiased sequence like e.g. 0101010101 by repeating the block 01, but this sequence is obviously not radom.

We need to introduce the second important notion of independent bits.

A "true random value" (here, value can be intended as a sequence of bits) of length m is

> a value chosen from all possible values with length m (2^m) where every value has the same probability to be chosen[4].

Thus, a True RNG has to

> produce a sequence of 0 and 1's that can be combined in subsequences or blocks representing *true random values* [5].

This definition includes both properties, and represents the goal of every Random Number Generator.

From a theoretical point of view, there are no natural or artificial processes able to generate a true random sequence (there isn't any way to achieve true randomness).

Instead, from a practical point of view, some natural processes like the atmospheric noise, the brownian motion, the thermal noise, the photoelectric effect, etc. can be considered as sources of true randomness.

A Random Number Generator (RNG) uses one of these non-deterministic sources [6], along with some processing function (i.e., the entropy distillation process), to produce randomness. Processing functions are explained in subsection 1.2.4.

1.2.3 Pseudo-Random Number Generators

This subsection introduces another important type of Random Number Generator, called Pseudo-Random Number Generator (PRNG).

[4] Part of the definition contained in [27].

[5] Definition taken from *Federal Information Processing Standards 140*[30].

[6] We can note that even RNGs are susceptible to being "pseudo" and must be subjected to randomness tests as software generators. Ultimately, it seems to be difficult to differentiate true randomness in physical processes from mere entropy, a lack of knowledge of some aspect or another of the system. Only by systematically analysing a series of experimental results for "randomness" one can make a judgement on whether or not the underlying process is truly random, or merely unpredictable. Note well that unpredictable and random are often used as synonyms, but they are not really the same thing. A thing may be unpredictable due to entropy – our lack of the data required to make it predictable. Examples of this sort of randomness abound in classical statistical mechanics or the theory of deterministic chaos.

Computers are deterministic, meaning that given an input, they always produce the same output.

Pseudo-Random Number Generators are used to generate randomness in a deterministic environment. A PRNG takes a small input called seed and produces a long sequence of pseudo-random values that appears to be random to an observer with a sufficiently limited computational power. In contexts in which unpredictability is needed, the seed itself must be random and unpredictable. Hence, by default, a PRNG should obtain its seeds from the outputs of an RNG; i.e., a PRNG requires a RNG as a companion.

As we said, outputs of a PRNG are typically deterministic functions of the seed; i.e., all true randomness is confined to seed generation. The deterministic nature of the process leads to the term "pseudorandom." Since each element of a pseudorandom sequence is reproducible from its seed, only the seed needs to be saved if reproduction or validation of the pseudorandom sequence is required.

Further we can find a subclass of PRNG called CSPRNG: Cryptographically secure PRNG. CSPRNG is a PRNG with properties that make it suitable for an application in cryptography.

1.2.4 Processing functions

The entropy of a sequence of bits can be improved by using a group of functions called randomness extractors. A randomness extractor is a function which, when applied to an entropy source, generates a random output that is shorter, but uniformely distributed. The goal of this process is to generate a truly random output stream, which could be considered as being a true random number generator. Anyway, no single randomness extractor currently exists that has been proven to work when applied to any type of high-entropy source.

There is another important class of functions called randomness expanders. These functions are keyed by secret, uniformly random strings, with each function taking as input any publicly known value and outputting a value indistinguishable from one distributed uniformly at random. The key can be so short (e.g. of logarithmic length), that one can often eliminate the need for any truly random bits by enumerating all choices for the seed.

1.2.5 A common problem: Unpredictability

A first problem of Random Number Generators is the difficulty to generate uncorrelated values.

Algorithms like Blum Blum Shub ([21]) reduce their security to the known computational difficulty of some computations (e.g. integer factorization). Anyway, as

stated in [2], even having a perfect (cryptographically secure) PRNG, it is necessary to consider physical sources of randomness. Blum ([5]) shows that criptographycally-secure theory leaves unsolved a fundamental problem: the source for the random seed. Using a fair source to generate this seed may be crucial because of the danger that the pseudo-random number generator might amplify any dependence or bias in the bits of the seed. In fact, Random and pseudorandom numbers generated for cryptographic applications should be unpredictable, respecting two important properties:

1. *forward unpredictability*: if the seed is unknown, the next output number in the sequence should be unpredictable in spite of any knowledge of previous random numbers in the sequence;[7]

2. *backward unpredictability*: it should not be feasible to determine the seed from knowledge of any generated values. Note that this is a stronger property than 1: if you can predict the seed you can also predict the sequence of generated values.

No correlation between a seed and any value generated from that seed should be evident; each element of the sequence should appear to be the outcome of an independent random event whose probability is as close as possible to $\frac{1}{2}$.

A common way to deal with this problem is to change the local seed with a certain frequency, so that it becomes impossible to gain an amount of values sufficient to infer useful information about the generator before seed is changed again.

1.2.6 A second common problem: Subtle patterns

Another problem of Random Number Generators is related to the amount of computation needed to unveil non-randomness.

In the context of modern computer, numerical simulations can consume a lot of e.g. bits and unsigned integers. Running simulations on large compute clusters can consume more than 10^{20} of uniform deviates in a single extended computation over the course of months to years. Even generators that appear to pass many tests sampling millions of random numbers may raise problems in these situations. Many of them are, in fact, state-periodic, repeating a single sequence after a certain number of returns (very short for some older generators). Then, if the simulation asks more random numbers than the period, it will reproduce same sample sequence

[7] An article of June 9, 2011 from Intel ([9]) describes Bull Mountain: a Digital Random Number Generator (DRNG) hardware implementation in new Intel® 64 Architecture. One result of Intel studies is that in MT19937 (commonly used variant of Mersenne Twister, an algorithm that provides fast generation of very high-quality pseudorandom numbers), observing 624 iterates allows one to predict all future iterates.

without generating the independent, identically distributed samples that may be needed.

A related issue is associated with the dimensionality of the correlation. Some generators produce random values subtly patterned in a space of high dimensionality. Some tests can reveal non uniformity by distributing random coordinate (N-tuples) in an N dimensional space. A drawback of this approach is that the space must be filled with a certain density, that is related to the number of dimensions. If this number is elevated, the computation needed is too heavy.

Chapter 2

PariRandom: Theoretical Analysis

This Chapter provides an abstract description of PariRandom, and its theoretical analysis.

Section 2.1 describes PariRandom algorithm: the interactions between nodes of the network and how local seeds are updated.

Section 2.2 analyses the minimum entropy quality that can be reached in the worst case (if the network is completely compromised).

Section 2.3 describes a mathematical approach to evaluate the improvement of local entropy quality for different network models.

Sections 2.4 and 2.5 summarizes the mathematical and statistical instruments used in section 2.3.

Finally, Section 2.6 lists the proofs of theorems contained in section 2.3.

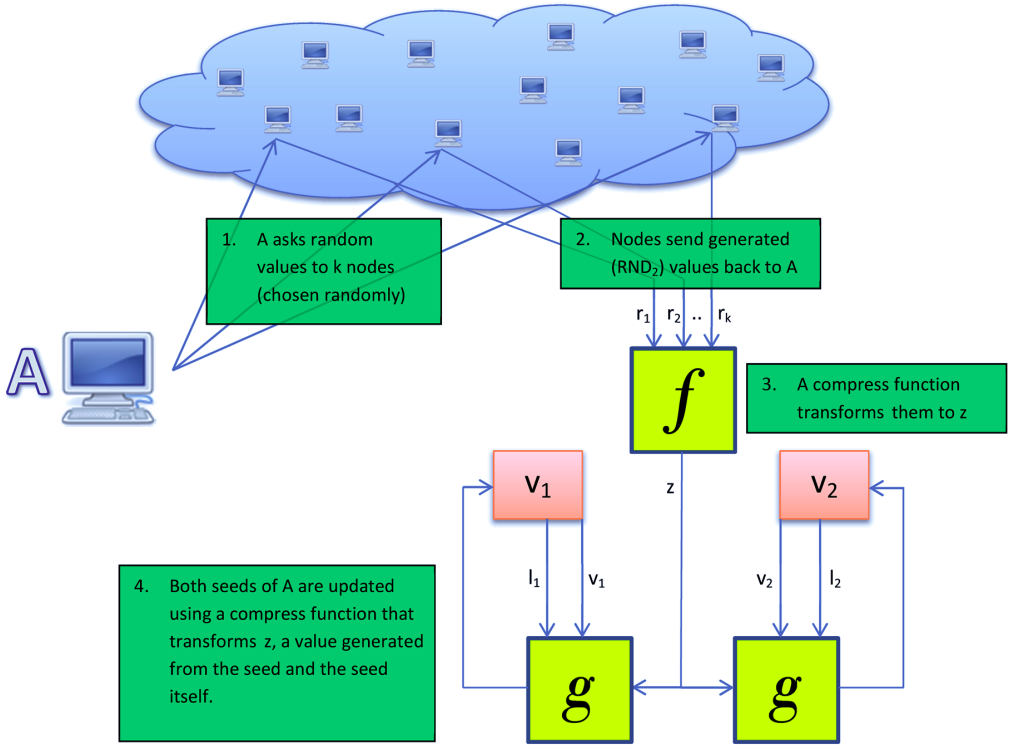

Figure 2.1.1: PariRandom Algorithm: node A starting a *seed-update* process.

2.1 Algorithm

PariRandom improves a generic pseudorandom generation algorithm (RND) relying on a network of N nodes.

Each node stores two variables initialized with two different values generated by a cryptographically secure pseudo-random number generator.

These variables (v_1 and v_2) are seeds for two local randomizers based on RND (RND_1 and RND_2). RND_1 is used to generate random values for normal usage (by user or other applications). Each time a value is generated by RND_1 a local counter is incremented. When this counter reaches a threshold value for a certain node A, the node starts a *seed-update* process, explained in figure 2.1.1. A asks to k other nodes[1] a random value. Those k nodes sends back k random values (r_1, r_2,... r_k) generated using RND_2. When A receives this values it computes

$$z = f(r_1, r_2, \ldots r_k) \tag{2.1.1}$$

[1]These nodes can be the next k nodes that communicates with A, so that we can implement a piggybacking technique (adding random values to the end of transmitted messages). We can also randomly choose them from the neighbourhood of the network. In a DHT network a valuable choice would be to contact a set of k nodes that covers the network space.

The compression function[2] f is an *exclusive disjunction* (*XOR* - see 2.2) of input values with a little variation to make it noncommutative:

$$z = f(r_1, r_2, \dots r_k) = shift(r_1, 1) \oplus shift(r_2, 2) \dots shift(r_k, k) \tag{2.1.2}$$

where *shift* is defined as

$$x = b_0 \circ b_1 \dots b_r; shift(x, t) = b_{r-t-1} \circ b_{r-t}, \dots b_r \circ b_0 \circ b_1 \dots b_{r-t} \tag{2.1.3}$$

and r is the lenght of x in bit and b_i is the bit in the position i of x (e.g.: 00100 $\rightarrow$ 10000 ($t = 2$), 01001 $\rightarrow$ 01010 ($t = 3$)).

Shift is used in some genetic algorithms ([11]) and should be able to catch network entropy (order of values received from other nodes).

Finally A updates its two seeds accordingly to

$$v_1\prime = g(v_1, l_1, z) \tag{2.1.4}$$

$$v_2\prime = g(v_2, l_2, z) \tag{2.1.5}$$

where l_1 and l_2 are two different values respectively generated by RND_1 and RND_2 and g represents a compression function[3] that is the XOR of three values:

$$g(x, y, z) = x \oplus y \oplus z \tag{2.1.6}$$

[2] A compression function is a function that transform fixed length inputs to an output of the same size of one of the inputs ([10]).

[3] Some algorithms updates the local seed by generating a random value with a third party generator (*/etc/rand* or a physical generator). In this case we can choose to generate l_1 and l_2 in the same way.

2.2 Minimum entropy level

This section proves a cornerstone of our algorithm: when applied to an existing PRNG, PariRandom can only improve its security and performance.

There is no possibility for PariRandom to decrease the quality of the PRNG that is applied upon it. In this sense we can think of PariRandom as a framework that can be used to improve existing randomization algorithms in any software that depends on P2P networks without any drawback except for a little increase of traffic that can be lowered through piggybacking technique.

The base of our algorithm is the *seed-update* process. When we apply formulas 2.1.4 and 2.1.5 we update the local seed of both local PRNG. Usually a PRNG periodically updates the local seed (s) according to

$$s\prime = h(s, l) \tag{2.2.1}$$

where l is a random value generated by the PRNG.

As we can see the difference between formulas 2.1.1,2.1.4 and 2.2.1 stands in the value z, that represents the output of compress function f and is XOR'd with the other input values. Exclusive OR has a very important property, that makes it widely used in entropy generation field[4]:

> [..] as long as at least one string is chosen uniformly at random and the other numbers are independent of it, x is distributed uniformly at random. [1]

This property recalls the notion of independency:

> A sequence of random variables X_1, X_2, ... are statistically independent if the following is satisfied for each of variables X_i: the variables apart from X_i do not provide any useful information for predicting the value of X_i.

Using this property we can prove that at any time during the execution the entropy of seed s_1 updated with PariRandom algorithm is equal or higher than if the seed (s_2) was updated with a normal local seed-update algorithm (2.2.1).

This proof is based on other two considerations:

1. Each newseed generated is based on last seed:

[4] This quotes comes from [3] and is referenced by [2] that in QRND algorithm computes a XOR of multiple uncorrelated bit strings to output a final string where each i-bit is the XOR'd product of all i-bits of initial strings and that has an entropy that is equal or higher than the maximal entropy of initial strings. An important document that explores the risks of "exclusive or" with correlated values is [6], that will be used as an input for Section 2.3.

$$newseed \approx currentseed \oplus oldseed \tag{2.2.2}$$

In this way if we go back to the beginning, we see that the last *newseed* is the result of an exclusive or from an *oldseed* that has been generated locally and does not derive from the network (*local starting seed*).

2. Other nodes has no direct (or even indirect in RND_1) knowledge of local seeds. This makes it impossible for malicious nodes to communicate values that can break XOR property due to any correlation between them and local seeds.

We can now demonstrate the following theorem:

PariRandom minimum entropy quality 2.2.1. *PRNG P is applied upon PariRandom. After any seed-update process performed by PariRandom, the entropy quality of the newseed is equal or higher than the entropy that can be obtained by P without the use of PariRandom.*

Proof. Proof by induction:

1. When we start PariRandom algorithm, we set s_1 by generating a random seed with a third party generator, in the same way used by most PRNGs. The entropy is obviously equal.

2. Assume that after n updates $s_1{}^n$ has an entropy that is equal or higher than $s_2{}^n$.

3. During update *n+1*, $s_1{}^{n+1}$ is the result of an exclusive or between $s_2{}^{n+1}$ and z (the difference between eq. 2.1.4 and 2.2.1).

$$s_1^{n+1} = s_2^{n+1} \oplus z \tag{2.2.3}$$

z is the output of compression function f. Using the XOR property cited before, we can state that if z is completely uncorrelated with all other inputs of the g compression function, the entropy gained by the newseed can't be lower using PariRandom algorithm.

As described in 2.1, RND_1 generates values for local use only. This means that there is no way for malicious nodes to know the local seed $s_1{}^{n+1}$. Value z will necessarily be uncorrelated with both the local seed and the value generated by the local seed (respectively v_1 and l_1 in 2.1.4).

If the entropy gain is not lower, and the entropy in the previous step was not lower, the sum will not be lower.

□

The unpredictability problem (see 1.2.5) has been partially solved by using two different random generators. This means that the amount of values generated by RND_1 is the same with or without PariRandom.

The worst attack against a PRNG running on PariRandom compared to the simple PRNGs will involve a complete knowledge of v_2 due to the big amount of random values generated by RND_2 and shared across the network. In this scenario, the quality will at least be the same.

One way to decrease the chance of revealing information regarding v_2 is to use a hashing algorithm before distributing values that are locally generated. This is computationally expensive but widely used in the literature[5]. This has not been implemented yet, due to the need to keep algorithm as simple as possible while testing its quality. Another element that improves security is the *shift* function, as it may be difficult for malicious nodes to forecast the order in which their messages will be received.

[5]In particular, Viega in [7] sees in hashing a tool to accumulate entropy and carefully estimates computational weight, while the already cited [1] uses hashing for our same scope: "For the random number generation, we need a bit commitment scheme h, i.e., a scheme where h(x) does not reveal anything about x. In practice, a cryptographic hash function might be sufficient for h so that the protocols below can be easily implemented".

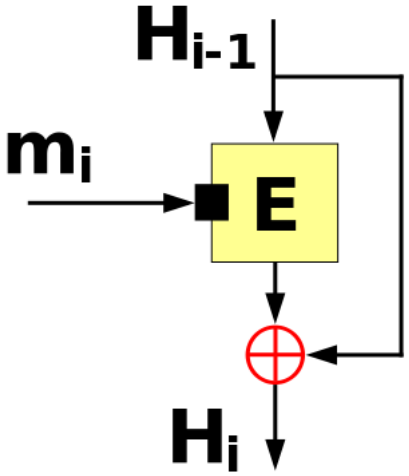

Figure 2.3.1: The Davies–Meyer one-way compression function.

2.3 Entropy gain

This section investigates the entropy quality that can be achieved by PariRandom if some nodes of the network are not malicious. In the first part, the algorithm is modelled to facilitate its analysis. The following part presents the theorems that describe PariRandom statistical properties from a theoretical point of view.

If the network is not completely compromised, PariRandom obtains entropy from three different sources:

- other nodes, particularly from their PRNG. An important observation is that some of them may have a real random entropy generator that shares part of its entropy to the network[6].
- Network structure (that influence contacted nodes), especially for interesting structures as *KAD* (used in *PariPari*).
- Network communications, even if some improvements can be done as actually this entropy is accumulated only from the *shift* function (order of received values), and there are better ways to extract randomness from latency delays and packet content.

To understand how each source influences the quality of the local seed we have to model the algorithm.

With some approximation the seed-update process can be seen as a single Davies–Meyer one-way compression function (2.3.1):

[6]This is similar to other algorithms (like *PariSync*) where some nodes have a higher quality. The advantage here is that there is no need of authoritative systems to certificate them.

> The Davies–Meyer single-block-length one-way compression function feeds each block of the message (m_i) as the key to a block cipher. It feeds the previous hash value (H_{i-1}) as the plaintext to be encrypted. The output ciphertext is then also XORed with the previous hash value (H_{i-1}) to produce the next hash value (H_i). In the first round when there is no previous hash value it uses a constant pre-specified initial value (H_0).[7]

In fact, each seed-update process consists of some consecutive XOR operations on the local seed.

On a higher view, all these update processes can be seen as a whole Davies-Meyer compression function operated on the *initialseed* and some inputs coming from nodes all around the network.

In [6] Robert Davies is interested in the effectiveness of the XOR operation to reduce bias (the deviation of the expectation from $\frac{1}{2}$). Bias is the first indicator of the quality of a PRNG, and is easy to use and esaminate.

A RNG that has no bias and whose generated bits are independent, is a perfect RNG.

Davies connects the notion of independency (see 2.2) to statistical analysis, particularly: in case of pairs of random bits "correlation = zero" and "independence" are equivalent. However when more than two random bits are involved, independence implies zero correlation but not vice versa.

Example: consider X, Y and X⊕Y where X and Y are random bits with expected value $\frac{1}{2}$, Each pair X,Y and X⊕Y are uncorrelated with each other, but X,Y and X⊕Y are not independent since given any two of these variables one can calculate the third.

A first result of Davies work is that XOR operation always reduce the bias when the component bits are independent.

In our algorithm, this means that if the are some good nodes, the quality of the seed improves. By using previous consideration 2 (par. 1) we can also state that at least the first time input values were independent (first XOR with *initialseed*), so bias can only be improved.

To model the entropy gain of our algorithm we will need some considerations:

- we will assume that different good nodes has independent initialseeds. This comes from initial seeds being generated by local RNG depending on entropy collected from running processes, network, device drivers or memory operations.[8]

[7] Quote from [10].

[8] A commonly used RNG for seeding in Unix-like operating systems is /dev/random. More information is available in [17].

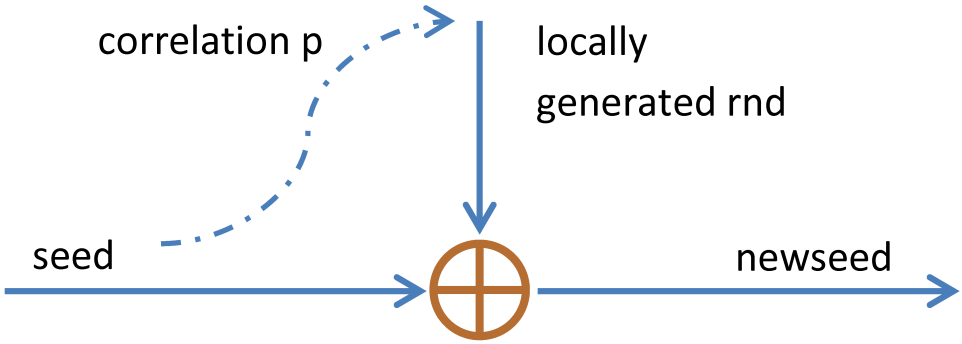

Figure 2.3.2: Node A updating seed without PariRandom algorithm.

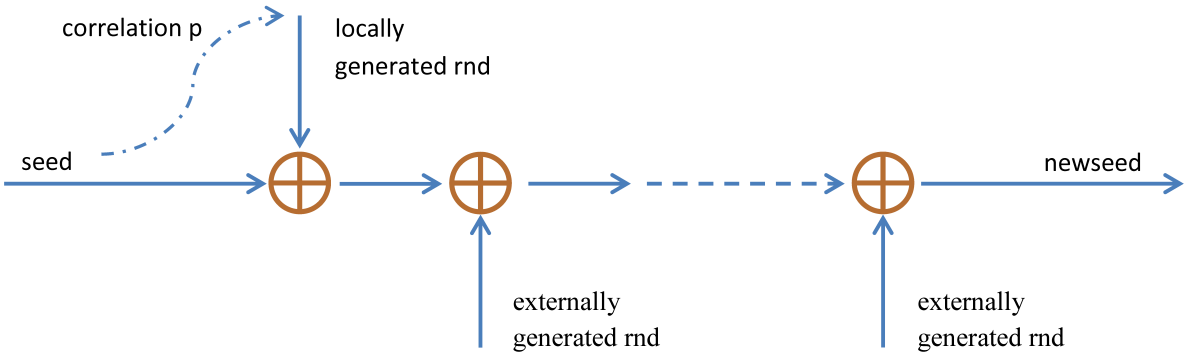

Figure 2.3.3: Node A updating seed with PariRandom algorithm.

- In each node, every generated value has a correlation p with its current seed.
- Every node A contacts nodes N_1,N_2,N_3,..N_n. Some of these nodes are malicious, we can assume from 1 to k (XOR is commutative). Remaining n to k nodes are good nodes.
- RND (bit stream) has an E(X)=ν very close to $\frac{1}{2}$. The generator used for initialseed has an E(Y)=μ very close to $\frac{1}{2}$.

Following results come from a statistical analysis based on calculation tricks described in section 2.5. To avoid breaking the flow of the Chapter, the proof of theorem 2.3.1 (and of all subsequent theorems) can be found at the end of this Chapter in Section 2.6.

Without PariRandom (2.3.2) we can prove following theorem:

Expectation of the seed's first bit after a seed-update without PariRandom (model described in Figure 2.3.2). 2.3.1.

$$\begin{aligned} E(newseed) &= E(oldseed \oplus v) \\ &= \frac{1}{2} - 2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - 2p\sqrt{\mu(1-\mu)\nu(1-\nu)} \\ &\approx \frac{1}{2} - 2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - \frac{1}{2}p \end{aligned} \tag{2.3.1}$$

(assuming μ,ν very close to $\frac{1}{2}$).

Correlation between oldseed and newseed first bit is defined by theorem:

Correlation between the seed's first bit before and after a seed-update without PariRandom (model described in Figure 2.3.2). 2.3.1.

$$\begin{aligned} corr(oldseed, newseed) &= \frac{cov(oldseed, oldseed \oplus v)}{\sqrt{var(oldseed)var(oldseed \oplus v)}} \\ &= \frac{p\sqrt{\mu(1-\mu)\nu(1-\nu)}(2\mu - 1) + \mu^2(2\nu - 1) + \mu(1 - 2\nu)}{\sqrt{var(oldseed)var(oldseed \oplus v)}} \end{aligned} \tag{2.3.2}$$

With PariRandom (2.3.3), if we have two good nodes sharing values, expectation and correlations are defined by following theorems:

Expectation of the seed's first bit after a seed-update with PariRandom (model described in Figure 2.3.3). 2.3.1.

$$\begin{aligned} E(newseed) &= E(oldseed \oplus v \oplus v' \oplus v'') \\ &= \frac{1}{2} + \left[-2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - 2p\sqrt{\mu(1-\mu)\nu(1-\nu)}\right](1 - 2\nu')(1 - 2\nu'') \end{aligned} \tag{2.3.3}$$

Correlation between the seed's first bit before and after a seed-update without PariRandom (model described in Figure 2.3.3). 2.3.1.

$$\begin{aligned} corr(oldseed, newseed) &= \frac{cov(oldseed, oldseed \oplus v \oplus v' \oplus v'')}{\sqrt{var(oldseed)var(oldseed \oplus v \oplus v' \oplus v'')}} \\ &= \frac{\left[\begin{array}{r} p\sqrt{\mu(1-\mu)\nu(1-\nu)}(2\mu - 1) \\ +\mu^2(2\nu - 1) + \mu(1 - 2\nu) \end{array}\right](2\nu' - 1)(2\nu'' - 1)}{\sqrt{var(oldseed)var(oldseed \oplus v \oplus v' \oplus v'')}} \end{aligned} \tag{2.3.4}$$

As proved by 2.3.1 the resulting bias of 2.3.1 is multiplied by a factor $(1-2x)$ for each external random received where x, being a bit expectation, must be between 0 and 1. So we have

$$-1 \leq (1 - 2x) \leq 1$$

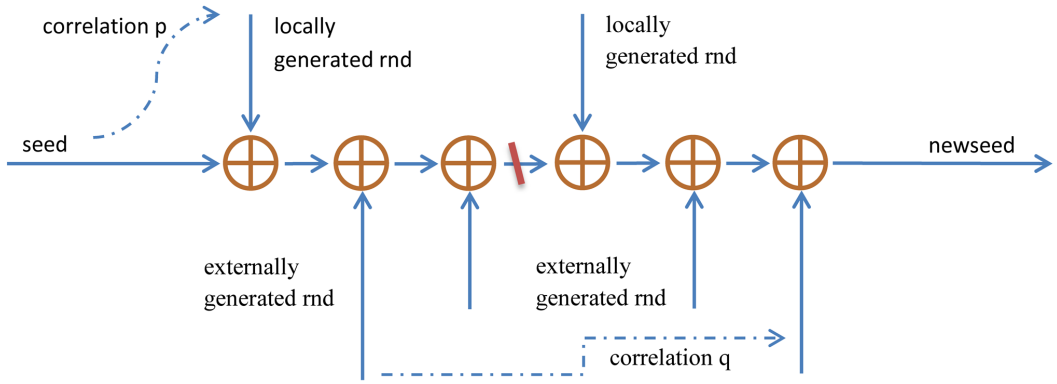

Figure 2.3.4: Two seed-update processes with PariRandom algorithm. One node has been contacted two times.

that means that *newseed* first bit has an equal or lower bias when using PariRandom algorithm.

We see that the correlation derived by 2.3.1 is the correlation derived by 2.3.1 multiplied by a factor:

$$\frac{(2\nu' - 1)(2\nu'' - 1)}{\sqrt{var(oldseed \oplus v \oplus v' \oplus v'')}} \cdot \frac{\sqrt{var(oldseed \oplus v)}}{1} \tag{2.3.5}$$

This factor has a value between -1 and 1, meaning that

$$|corr(o, o \oplus v \oplus v' \oplus v'')| \leq |corr(o, o \oplus v)|$$

so the *newseed* has an equal or lower correlation with oldseed when using PariRandom algorithm.

We can go further by modelling 2 seed-update processes using a node to get two random values, one for each seed update (2.3.4).

In this case, the bias using PariRandom algorithm is the bias without the algorithm multiplied by a factor

$$-1 \leq [(1 - 2v')(1 - 2v'''') + 4q]\,(1 - 2v'')(1 - 2v'''') \leq 1 \tag{2.3.6}$$

and the correlation ratio becomes:

$$\frac{[(1 - 2v')(1 - 2v'''') + 4q]\,(1 - 2v'')(1 - 2v'''')}{\sqrt{var(oldseed \oplus v \oplus v' \oplus v'' \oplus v''' \oplus v'''' \oplus v'''''')}} \cdot \frac{\sqrt{var(oldseed \oplus v' \oplus v''')}}{1} \tag{2.3.7}$$

that also has a value between -1 and 1, meaning that

$$|corr(o, o \oplus v \oplus v' \oplus v'' \oplus v''' \oplus v'''' \oplus v''''')| \leq |corr(o, o \oplus v \oplus v''')|$$

Above equations explain how bias and correlation of the first seed bit change with and without PariRandom. We can perform same computations for following bits, and particularly we can examine how the correlation between the 1st seed bit and the 2nd one changes using PariRandom during seed update.

Ratio between the correlation of the first bit before seed update and the second bit after seed update, with and without PariRandom, (model described in Figure 2.3.4). 2.3.1.

$$\frac{[(1-2v_2')(1-2v_2'''')+4q]\,(1-2v_2'')(1-2v_2''')}{\sqrt{var(oldseed_2 \oplus v_2 \oplus v_2' \oplus v_2'' \oplus v_2''' \oplus v_2'''' \oplus v_2''''')}} \cdot \frac{\sqrt{var(oldseed_2 \oplus v_2' \oplus v_2''')}}{1} \tag{2.3.8}$$

The ratio between correlations described in theorem 2.3.1 also has a value between -1 and 1, meaning that

$$|corr(o_1, o_2 \oplus v_2 \oplus v_2' \oplus v_2'' \oplus v_2''' \oplus v_2'''' \oplus v_2''''')| \leq |corr(o_1, o_2 \oplus v_2 \oplus v_2''')|$$

We could go further computing unparameterized values with different models, anyway this is beyond the scope of this study. Those calculations would need to introduce more parameters (covariances, ecc) and all the instruments needed to perform those computations has been presented. Some useful equations are available in section 2.4.

2.4 Statistical basis

Symbol $\oplus$ in following proofs is used to denote the exclusive-or operation. So $X \oplus Y = 1$ if just one of X and Y is equal to 1; otherwise $X \oplus Y = 0$.

The XOR operation is commutative ($X \oplus Y = Y \oplus X$) and associative ($X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$).

Some equations used to compute expected value or mean value (E(X)), variance (var(X)), covariance (cov(X,Y)) and correlation (corr(X,Y)).

- $var(X) = E[\{X - E(X)\}^2]$
- $cov(X,Y) = E[\{X - E(X)\}\{Y - E[Y]\}] = E(XY) - E(X)E(Y)$
- $corr(X,Y) = \frac{cov(X,Y)}{\sqrt{var(X)var(Y)}}$
- $var(XY) = E[X^2Y^2] - E^2[XY] = var(X)var(Y) + E^2(Y)var(X) + E^2(X)var(Y)$
- $E(XYZ) = E(X)E(Y)E(Z) + E(X)cov(Y,Z) + E(Y)cov(X,Z) + E(Z)cov(X,Y) + E\{[X - E(X)][Y - E(Y)][Z - E(Z)]\}$
- $E\left[\prod_{s=1}^{N} X_s\right] = \prod_{s=1}^{N} E[X_s] + cov\left(\prod_{s=1}^{N-1} X_s, X_n\right)$
  --→ $+ \sum_{s=1}^{N-2} \left[\prod_{k=1}^{s} E[X_{N-k+1}] cov\left(\prod_{m=1}^{N-s-1} X_m, X_{N-s}\right)\right]$
- deriving from previous formula - $E(XYZ) = E(X)E(Y)E(Z) + cov(XY,Z) + E(Z)cov(X,Y)$
- $cov(XY,UV) = E(X)E(U)cov(Y,V) + E(X)E(V)cov(Y,U) + E(Y)E(U)cov(X,V)$
  --→ $+E(Y)E(V)cov(X,U) + E[D(X)D(Y)D(U)D(V)]$
  --→ $+E(X)E[D(Y)D(U)D(V)] + E(Y)E[D(X)D(U)D(V)]$
  --→ $+E(U)E[D(X)D(Y)D(V)] + E(V)E[D(X)D(Y)D(U)]$
  --→ $-cov(X,Y)cov(U,V)$
  where $D(X) = x - E(X)$

2.5 Analysing XOR operation

Suppose X can take values 0 or 1.

Let $a(X) = 1 - 2X$. So $X = \{1 - a(X)\}/2$ and $a(X)$ takes the values 1 and -1 corresponding to X's 0 and 1. Then

$$a(X \oplus Y) = a(X)a(Y)$$

The usefulness of this is that we know how to manipulate multiplication in probability calculations but doing XOR calculations directly is awkward and unfamiliar. Also

- $E\{a(X)\} = 1 - 2E(X)$
- $var\{a(X)\} = 4var(X)$
- $cov\{a(X), a(Y)\} = 4cov(X, Y)$
- $corr\{a(X), a(Y)\} = corr(X, Y)$

so we can transform expectations and variances between X and a(X).

Also note that $a(X)^2 = 1$.

A useful considerations is that

- $-1 \leq E\{\prod_{\forall i} a(X_i)\} \leq 1$ as $-1 \leq a(X_i) \leq 1$ for every X_i.

2.6 Theorems Proofs

This section presents a list of proofs for the theorems presented in section 2.3.

Proof 2.3.1

Suppose o $(=oldseed)$ and v have covariance c.

$$\begin{aligned}
E(newseed) &= E(oldseed \oplus v) \\
&= \frac{1}{2} - \frac{1}{2}E\left\{a(o \oplus v)\right\} \\
&= \frac{1}{2} - \frac{1}{2}E\left\{a(o)a(v)\right\} \\
&= \frac{1}{2} - \frac{1}{2}\left[E\left\{a(o)\right\}E\left\{a(v)\right\} + 4c\right] \\
&= \frac{1}{2} - \frac{1}{2}(1-2\mu)(1-2\nu) - 2c \\
&= \frac{1}{2} - 2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - 2c \\
&= \frac{1}{2} - 2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - 2p\sqrt{\mu(1-\mu)\nu(1-\nu)}
\end{aligned}$$

Proof 2.3.2

Suppose o $(=oldseed)$ and v have covariance c.

$$\begin{aligned}
corr(oldseed, oldseed \oplus v) &= \frac{cov(oldseed, oldseed \oplus v)}{\sqrt{var(oldseed)var(oldseed \oplus v)}} \\
&= \frac{\frac{1}{4}cov\left\{a(o), a(o \oplus v)\right\}}{\sqrt{var(o)var(o \oplus v)}} \\
&= \frac{\frac{1}{4}cov\left\{a(o), a(o)a(v)\right\}}{\sqrt{var(o)var(o \oplus v)}} \\
&= \frac{\frac{1}{4}\left[E\left\{a(o)a(o)a(v)\right\} - E\left\{a(o)\right\}E\left\{a(o)a(v)\right\}\right]}{\sqrt{var(o)var(o \oplus v)}} \\
&= \frac{\frac{1}{4}\left[E\left\{a(v)\right\} - E\left\{a(o)\right\}\left[E\left\{a(o)\right\}E\left\{a(v)\right\} + 4c\right]\right]}{\sqrt{var(o)var(o \oplus v)}} \\
&= \frac{\frac{1}{4}\left[1 - 2\nu - (1-2\mu)\left[(1-2\mu)(1-2\nu) + 4c\right]\right]}{\sqrt{var(o)var(o \oplus v)}} \\
&= \frac{c(2\mu - 1) + \mu^2(2v-1) + \mu(1-2v)}{\sqrt{var(oldseed)var(oldseed \oplus v)}} \\
&= \frac{p\sqrt{\mu(1-\mu)\nu(1-\nu)}(2\mu-1) + \mu^2(2v-1) + \mu(1-2v)}{\sqrt{var(oldseed)var(oldseed \oplus v)}}
\end{aligned}$$

Proof 2.3.3

Suppose o ($=oldseed$) and v have covariance c.

$$\begin{aligned}
E(newseed) &= E(oldseed \oplus v \oplus v' \oplus v'') \\
&= \frac{1}{2} - \frac{1}{2}E\left\{a(o \oplus v \oplus v' \oplus v'')\right\} \\
&= \frac{1}{2} - \frac{1}{2}E\left\{a(o)a(v)a(v')a(v'')\right\} \\
&= \frac{1}{2} - \frac{1}{2}\left[E\left\{a(o)\right\}E\left\{a(v)\right\} + 4c\right]E\left\{a(v')\right\}E\left\{a(v'')\right\} \\
&= \frac{1}{2} - \frac{1}{2}\left[(1-2\mu)(1-2\nu) + 4c\right](1-2\nu')(1-2\nu'') \\
&= \frac{1}{2} + \left[-2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - 2c\right](1-2\nu')(1-2\nu'') \\
&= \frac{1}{2} + \left[-2(\mu - \frac{1}{2})(\nu - \frac{1}{2}) - 2p\sqrt{\mu(1-\mu)\nu(1-\nu)}\right](1-2\nu')(1-2\nu'')
\end{aligned}$$

Proof 2.3.4

Suppose o ($=oldseed$) and v have covariance c.

$$
\begin{aligned}
corr(o, o \oplus v \oplus v' \oplus v'') &= \frac{cov(oldseed, oldseed \oplus v \oplus v' \oplus v'')}{\sqrt{var(oldseed)var(oldseed \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\frac{1}{4}cov\left\{a(o), a(o \oplus v \oplus v' \oplus v'')\right\}}{\sqrt{var(o)var(o \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\frac{1}{4}cov\left\{a(o), a(o)a(v)a(v')a(v'')\right\}}{\sqrt{var(o)var(o \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\frac{1}{4}\begin{bmatrix} E\left\{a(o)a(o)a(v)a(v')a(v'')\right\} \\ -E\left\{a(o)\right\}E\left\{a(o)a(v)a(v')a(v'')\right\} \end{bmatrix}}{\sqrt{var(o)var(o \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\frac{1}{4}\begin{bmatrix} E\left\{a(v)\right\}E\left\{a(v')\right\}E\left\{a(v'')\right\} \\ -E\left\{a(o)\right\}\left[E\left\{a(o)\right\}E\left\{a(v)\right\}+4c\right]E\left\{a(v')\right\}E\left\{a(v'')\right\} \end{bmatrix}}{\sqrt{var(o)var(o \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\frac{1}{4}\begin{bmatrix} (1-2\nu)(1-2\nu')(1-2\nu'') \\ -(1-2\mu)\left[(1-2\mu)(1-2\nu)+4c\right](1-2\nu')(1-2\nu'') \end{bmatrix}}{\sqrt{var(o)var(o \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\left[c(2\mu-1)+\mu^2(2\nu-1)+\mu(1-2\nu)\right](2\nu'-1)(2\nu''-1)}{\sqrt{var(oldseed)var(oldseed \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\left[p\sqrt{\mu(1-\mu)\nu(1-\nu)}(2\mu-1)+\mu^2(2\nu-1)+\mu(1-2\nu)\right]}{} \\
&\quad \frac{(2\nu'-1)(2\nu''-1)}{\sqrt{var(oldseed)var(oldseed \oplus v \oplus v' \oplus v'')}}
\end{aligned}
$$

Proof 2.3.5

$$
\begin{aligned}
\frac{corr(o, o \oplus v \oplus v' \oplus v'')}{corr(o, o \oplus v)} &= \frac{\left[E\left\{a(v')\right\}E\left\{a(v'')\right\}\right]\sqrt{var(o \oplus v)}}{\sqrt{var(o \oplus v \oplus v' \oplus v'')}} \\
&= \frac{\left[E\left\{a(v')\right\}E\left\{a(v'')\right\}\right]\sqrt{\left[E\left\{\left[a(o)a(v)\right]^2\right\}-E^2\left\{a(o)a(v)\right\}\right]}}{\sqrt{E\left\{\left[a(o)a(v)a(v')a(v'')\right]^2\right\}-E^2\left\{a(o)a(v)a(v')a(v'')\right\}}} \\
&= \frac{\left[E\left\{a(v')\right\}E\left\{a(v'')\right\}\right]\sqrt{\left[E\left\{\left[a(o)a(v)\right]^2\right\}-E^2\left\{a(o)a(v)\right\}\right]}}{\sqrt{E\left\{\left[a(o)a(v)\right]^2\right\}E\left\{\left[a(v')a(v'')\right]^2\right\}-E^2\left\{a(o)a(v)\right\}E^2\left\{a(v')a(v'')\right\}}} \\
&= \frac{\left[E\left\{a(v')\right\}E\left\{a(v'')\right\}\right]\sqrt{1-X}}{\sqrt{1-XY}}
\end{aligned}
$$

where

$$0 \leq X = E^2\left\{a(o)a(v)\right\} \leq 1$$

$$0 \leq Y = E^2\{a(v')a(v'')\} \leq 1$$

and

$$0 \leq \frac{\sqrt{1-X}}{\sqrt{1-XY}} \leq 1, -1 \leq [E\{a(v')\}E\{a(v'')\}] \leq 1$$

obtaining

$$-1 \leq \frac{corr(o, o \oplus v \oplus v' \oplus v'')}{corr(o, o \oplus v)} \leq 1$$

Proof 2.3.6

Without PariRandom we have:

$$E(oldseed \oplus v \oplus v''') = \frac{1}{2} - \frac{1}{2}E\{a(o)a(v)a(v''')\}$$

While with PariRandom we have:

$$E(oldseed \oplus v \oplus v' \oplus v'' \oplus v''' \oplus v'''' \oplus v''''') = \frac{1}{2} - \frac{1}{2}\begin{bmatrix} E\{a(o)a(v)a(v''')\}E\{a(v')a(v''''')\} \\ E\{a(v'')\}E\{a(v'''')\} \end{bmatrix}$$

Obtaining the bias ($|E(X) - \frac{1}{2}|$) difference factor:

$$E\{a(v')a(v''''')\}E\{a(v'')\}E\{a(v'''')\}$$

where every factor E(X) has a value between -1 and 1 and:

$$E\{a(v')a(v''''')\}E\{a(v'')\}E\{a(v'''')\} = [(1-2v')(1-2v''''') + 4q](1-2v'')(1-2v'''')$$

Proof 2.3.7

$$\frac{corr\left(o, \begin{array}{c} o \oplus v \oplus v' \oplus v'' \\ \oplus v''' \oplus v'''' \oplus v''''' \end{array}\right)}{corr(o, o \oplus v \oplus v''')} = \frac{[E\{a(v'')\} E\{a(v'''')\} E\{a(v')a(v''''')\}] \sqrt{var(o \oplus v \oplus v''')}}{\sqrt{var(o \oplus v \oplus v' \oplus v'' \oplus v''' \oplus v'''' \oplus v''''')}}$$

$$[E\{a(v'')\} E\{a(v'''')\} E\{a(v')a(v''''')\}]$$

$$= \frac{\sqrt{\left[E\left\{[a(o)a(v)a(v''')]^2\right\} - E^2\{a(o)a(v)a(v''')\}\right]}}{\sqrt{\begin{array}{l} E\left\{[a(o)a(v)a(v')a(v'')a(v''')a(v'''')a(v''''')]^2\right\} \\ -E^2\{a(o)a(v)a(v')a(v'')a(v''')a(v'''')a(v''''')\} \end{array}}}$$

$$[E\{a(v'')\} E\{a(v'''')\} E\{a(v')a(v''''')\}]$$

$$= \frac{\sqrt{\left[E\left\{[a(o)a(v)a(v''')]^2\right\} - E^2\{a(o)a(v)a(v''')\}\right]}}{\sqrt{\begin{array}{l} E\left\{[a(o)a(v)a(v''')]^2\right\} E\left\{[a(v')a(v'')a(v'''')a(v''''')]^2\right\} \\ -E^2\{a(o)a(v)a(v''')\} E^2\{a(v')a(v'')a(v'''')a(v''''')\} \end{array}}}$$

$$= \frac{[E\{a(v'')\} E\{a(v'''')\} E\{a(v')a(v''''')\}] \sqrt{1-X}}{\sqrt{1-XY}}$$

where

$$0 \leq X = E^2\{a(o)a(v)a(v''')\} \leq 1$$

$$0 \leq Y = E^2\{a(v')a(v'')a(v'''')a(v''''')\} \leq 1$$

and

$$0 \leq \frac{\sqrt{1-X}}{\sqrt{1-XY}} \leq 1, -1 \leq [E\{a(v'')\} E\{a(v'''')\} E\{a(v')a(v''''')\}] \leq 1$$

obtaining

$$-1 \leq \frac{corr(o, o \oplus v \oplus v' \oplus v'' \oplus v''' \oplus v'''' \oplus v''''')}{corr(o, o \oplus v \oplus v''')} \leq 1$$

Proof 2.3.8

Proceeding similarly to 2.6:

$$\frac{corr\left(o_1,\ \begin{matrix} o_2 \oplus v_2 \oplus v_2' \oplus v_2'' \\ \oplus v_2''' \oplus v_2'''' \oplus v_2''''' \end{matrix}\right)}{corr(o_1, o_2 \oplus v_2 \oplus v_2''')} = \frac{\left[E\left\{a(v_2'')\right\} E\left\{a(v_2'''')\right\} E\left\{a(v_2')a(v_2''''')\right\}\right]\sqrt{1-X}}{\sqrt{1-XY}}$$

where

$$0 \leq X = E^2\left\{a(o_2)a(v_2)a(v_2''')\right\} \leq 1$$

$$0 \leq Y = E^2\left\{a(v_2')a(v_2'')a(v_2'''')a(v_2''''')\right\} \leq 1$$

and

$$0 \leq \frac{\sqrt{1-X}}{\sqrt{1-XY}} \leq 1, -1 \leq \left[E\left\{a(v_2'')\right\} E\left\{a(v_2'''')\right\} E\left\{a(v_2')a(v_2''''')\right\}\right] \leq 1$$

obtaining

$$-1 \leq \frac{corr(o_1, o_2 \oplus v_2 \oplus v_2' \oplus v_2'' \oplus v_2''' \oplus v_2'''' \oplus v_2''''')}{corr(o_1, o_2 \oplus v_2 \oplus v_2''')} \leq 1$$

Chapter 3

PariRandom: Development and Testing

While the theoretical analysis in section 2.3 provides an accurate way to study algorithm flaws and capabilities, understanding the performance of PariRandom on a large network using that approach becomes unfeasible, as equations size grows exponentially with the number of connected nodes.

Section 3.1 describes the implementation of PariRandom as a plugin for PariPari. After the spread of PariPari, this will allow to test it in a large real network. Until this time, theoretical analysis have been integrated with practical tests on a large amount of values generated using a simulator.

Sections 3.2 and 3.3 respectively describes the ability of statistical tests to evaluate entropy quality for a given sequence and some of these tests.

Section 3.4 explains which testing suites has been used to perform this tests.

Sections 3.5 and 3.6 summarize the development and the implementation of the simulator used to produce tested sequences.

Finally, section 3.7 exhibits and reasons over the results obtained by running testing suites over the data produced by PariRandom simulator.

3.1 PariRandom implementation

This section describes the structure of the library implementing PariRandom.

The PariRandom Library (written in JAVA™) has been designed to work both as a standard library or as a PariPari plugin.

First of all, a main PariRandom class is used to start two different threads:

- PariRandomInternalServerThread: answers to requests performed locally by other applications and/or other PariPari plugins;
- PariRandomExternalServerThread: is responsible of sharing the local entropy with the network.

Both these threads are represented by interfaces. In this way, it that can be used to easily deploy a customization of existing standard implementations. For example, a class implementing PariRandomInternalServerThread may automatically place the entropy generated by PariRandom in the kernels pool.

Standard PariRandomInternalServerThread implementation listen to messages asking for a random number generator and replies with an istance of the class PariRandomNumberGenerator.

PariRandomNumberGenerator extends java.util.Random and mantains 2 different local seeds (the normal Random seed and another seed called *networkSeed*). Every time an application calls the next() method of this class (which generates the next pseudorandom value and is used by all other methods, like nextByes, nextLong, etc.) a static counter is incremented. If this counter reaches a certain limit, PariRandomNumberGenerator starts a thread called PariRandomSeedUpdate that is responsible of the seed-update process.

After receiving a list of other nodes running PariRandom from the PariRandomNetwork class, PariRandomSeedUpdate performs following operations:

- asks a random value to every node of the list;
- XOR every received value using the shift function defined in section 2.1;
- XOR obtained value with a new random value (PariRandomNumberGenerator.*nextLong()*) and with the last seed (PariRandomNumberGenerator.*getSeed()*);
- set this value as the new seed of PariRandomNumberGenerator (PariRandomNumberGenerator.*setSeed()*).

This procedure is repeated to update networkSeed.

PariRandomNetwork is an abstract class containing a method (*getNeighbours()*) used to retrieve a list of those nodes that can be contacted by PariRandomSeedUpdate. The current implementation of this abstract class is PariRandomNetworkBucket that contains a public Vector variable where the application running PariRandom has to keep an updated list of the addresses of reachable nodes running PariRandom (and/or a part of them, as used in PariPari due to the DHT structure).

PariRandomExternalServerThread listens for requests of random values and calls PariRandomNumberGenerator.*getNetworkLong()* to generate the response. This method generates a random value using the networkSeed.

Finally, class PariSyncSettings is used to define the main parameters of PariRandom, while ensuring that these parameters are supported by other nodes in the network using this library. Important parameters of this class are:

- the number of random values generated by PariRandomNumberGenerator before starting a seed-update process;
- an alternative class to java.util.Random to be extended by PariRandomNumberGenerator. In this way PariRandom may improve the quality of a custom pseudo-random generation algorithm.

PariSyncSettings is also used to define the way to interact with the network. In fact PariRandom libraries contains four interfaces (PariRandomPacket, PariRandomPiggyback, PariRandomSocket, PariRandomSocketAddress) implemented using standard Java classes (PariRandomPiggyback is used to extend PariRandomPacket) that can be customized to work in different contexts, e.g. with JXTA[1].

[1]JXTA (Juxtapose) is an open source peer-to-peer protocol specification begun by Sun Microsystems in 2001. The JXTA protocols are defined as a set of XML messages which allow any device connected to a network to exchange messages and collaborate independently of the underlying network topology. As JXTA is based upon a set of open XML protocols, it can be implemented in any modern computer language. JXTA peers create a virtual overlay network which allows a peer to interact with other peers even when some of the peers and resources are behind firewalls and NATs or use different network transports.[28]

3.2 Testing approaches

This section explains existing approaches for statistical testing of random sequences and their validity, that is the ability of these tests to effectively evaluate the entropy of the generator that produced those sequences.

An initial test is the simple visual analysis, by creating an image with produced values. Humans are really good at spotting patterns. One example is shown in figure 3.2.1[2].

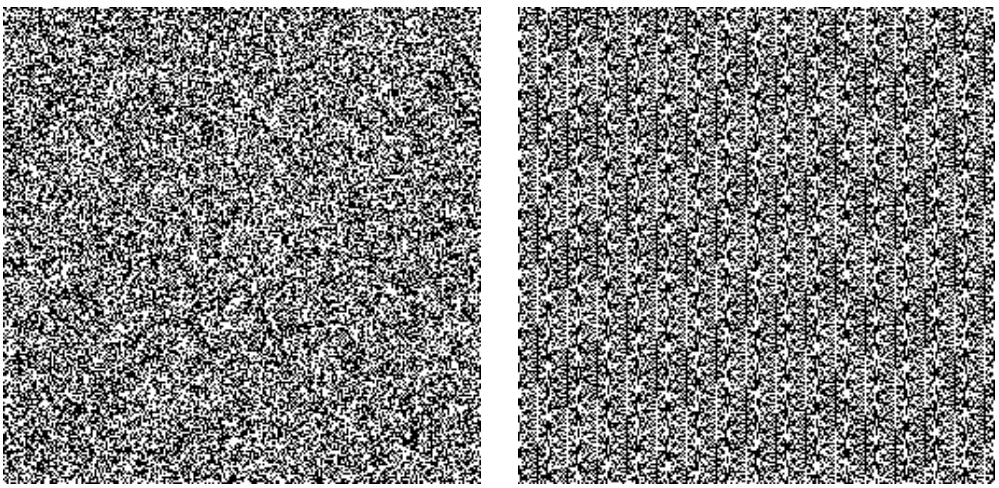

Figure 3.2.1: Visualisation of Random.org compared with PHP rand() on Microsoft Windows.

Randomness is a probabilistic property; that is, the properties of a random sequence can be characterized and described in terms of probability. The likely outcome of statistical tests, when applied to a truly random sequence, is known a priori and can be described in probabilistic terms.

PariRandom has been tested with different random testing suites. These suites are batteries of statistical hypothesis tests for measuring the quality of a random number generator. They analyse the distribution pattern of a set of data.

In stochastic modelling, as in some computer simulations, the expected random input data can be verified to show that tests were performed using randomized data. In some cases, data reveals an obvious non-random pattern, as with so-called "runs in the data" (such as expecting random 0–9 but finding "4 3 2 1 0 4 3 2 1..." and rarely going above 4). If a selected set of data fails the tests, then parameters can be changed or other randomized data can be used which does pass the tests for randomness.

There are many practical measures of randomness for a binary sequence.

Given a binary sequence s, we want to establish whether or not s passed or failed a statistical test. [8] identifies three evaluation approaches, that are not exhaustive

[2]Example images has been taken from [22]

and are intended to illustrate contrasting techniques:

1. Threshold Values: computing test statistics for a binary sequence and comparing it to a threshold value. The decision rule in this case states that a binary sequence fails this test "whenever the value of c(s) falls below the threshold value".

2. Fixed Ranges: still computing a statistic test as before, but in this case the decision rule states that "s fails a test if the test statistic falls outside a range".

3. Probability values: involves computing a test statistic for s and its corresponding probability value (P-value). Typically, test statistics are constructed so that large values of a statistic suggest a non-random sequence. The P-value is the probability of obtaining a test statistic as large or larger than the one observed if the sequence is random. Thus, small values (e.g. <0.05 or <0.01) are interpreted as evidence that a sequence is unlikely to be random. The decision rule in this case states that "for a fixed significance value α, s fails the statistical test if its P-value $< \alpha$".

The first two methods are lacking because they do not represent sufficiently stringent measures in most cases and because of their needs of pre-computed significance levels and acceptable ranges. If significance levels are modified in the future, the range values must be recomputed.

The third method, instead, while not being trivial in some cases, has the added advantage that it does not require the specification of significance levels. Once a P-value has been computed, the P-value can be compared to an arbitrary α.

Typically the P-Value are computed using special functions according to the Central Limit Theorem[3] such as the error function:

$$p = erfc\left(\frac{|\mu - x|}{\sigma\sqrt{2}}\right)$$

This is the P-value associated with the *null hypothesis* (H_0, the sequence being tested is random). Associated with this null hypothesis is the alternative hypothesis (H_a), which is that the sequence is not random. We assume that the generator is good, create a statistic based on this assumption, determine the probability of obtaining that value for the statistic if the null hypothesis is correct, and then interpret the probability as success or failure of the null hypothesis. This tests are conclusion-generation procedures that have two possible outcomes: either accept H_0 or accept H_a. Table 3.1, from [12], shows the possible conclusions.

[3]For a random sample of size n from a population with mean μ and variance σ^2, the distribution of the sample means is approximately normal with mean μ and variance σ^2/n as the sample size increases.

TRUE SITUATION	CONCLUSION	
	Accept H_0	Accept H_a (reject H_0)
Data is random (H_0 is true)	No error	Type I error
Data is not random (H_a is true)	Type II error	No error

Table 3.1: Status of the data at hand to the conclusion arrived at using the testing procedure.

If the data is, in truth, random, then a conclusion to reject H_0 will occur in a small percentage. That's what is commonly known[4] a Type I error. The probability of this type of error is usually called the *level of significance* of the test, and can be denoted with α. It can be set before the test and it is the probability that the test will indicate that the sequence is not random when it really is random (sequence appears having non-random properties even when produced by a "good" generator). Common values for α are in the range [0.001,0.01].

The probability of Type II error is denoted as β and is the probability that the test will indicate that the sequence is random when it is not; that is, a "bad" generator produced a sequence that appears to have random properties.

These two probabilities have an important different property: unlike α, β is not fixed and can take on many different values because there are an infinite number of ways that a data stream can be non-random, and each different way yields a different β. That's why the calculation of the Type II error is more difficult than the calculation of Type I error: there are many possible types of non-randomness.

All the suites considered for testing try, as a primary goal, to minimize the probability of a Type II error. Obviously, the probabilities α and β are related to each other and to the size of n in such a way that if two of them are specified, the third value is automatically determined.

Resuming, once a significance level α has been chosen, if P-value$\geq\alpha$, then the null hypothesis is accepted (sequence appears to be random). If P-value$\leq\alpha$, then the null hypothesis is rejected (the sequence appears to be non-random. With an α of 0.01 one could expect 1 sequence in 100 sequences to be rejected by the test if the sequence was random. A P-value$\geq$0.01 would mean that the sequence would be considered to be random with a confidence of 99%. A P-value$\leq$0.01 would mean that the conclusion was that the sequence is non-random with a confidence of 99%.

Usually, if a p-value is very, very low ($<10^{-6}$) we are pretty safe in rejecting the null hypothesis and concluding that the RNG is "bad". We could be wrong, but the chances are a million to one against a good generator producing the observed value of P. This consideration is insufficient if the value is not so close to 0. By itself the P-value from a single trial tells little in most cases. Repeated tests must

[4] These terms are precise technical terms used in statistics to describe particular flaws in testing process (see [25]).

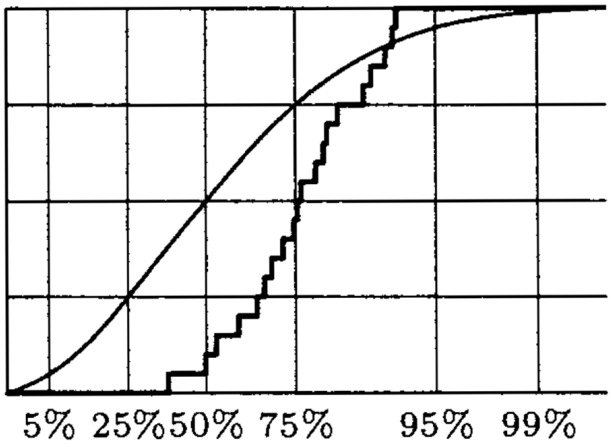

Figure 3.2.2: Examples of empirical distributions, from [14].

be done, and we need to examine the probability that the resulting sequence of P-values might occur if the underlying distribution of P-values is in fact uniform (as a new null hypothesis).

Used suites apply Kolmogorov-Smirnov (KS)[5] test to the p-values observed to determine the probability of obtaining them in a random sampling of a uniform distribution. This is itself a P-value, but now it applies to the entire series of independent and identically distributed trials.

In 3.2.2 we see an example of 20 observations with p-values falling between the 5 and 95 per cent levels, so we would not have regarded any of them suspicious, individually; yet collectively the empirical distribution shows that these observations are not at all right. KS is able to reject null hypothesis for this sequence of p-values.

By using this process (collect p-values, test them with KS) we get two parameters, the number of samples and the number of trials. If we are able to adjust these parameters and repeatedly obtain very low overall p-values (the p of the distribution of the p-values of the distribution of p-values of the experiment), we can safely reject the null hypothesis. If we can't do that, we are justified in concluding that the RNG passes that type of test.

This doesn't mean that the null hypothesis is correct. It only means that the test is unable to prove the alternative hypothesis.

This is the common idea among almost every RNG test, with few exceptions[6].

[5]Kolmogorov-Smirnnov is a test for the equality of continuous, one-dimensional probability distributions that can be used to compare a sample with a reference probability distribution.

[6]Like the bit persistence test, common in many suites, that does successive exclusive-or tests of succeeding unsigned integers returned by a RNG. After some trials, the result is a bitmask showing if some bits did not change throughout the sequence.

3.3 Common statistical tests

In the previous section the p-value has been described as a way to evaluate if the null-hypothesis can be refused or not. This section contains a list of common tests that are used in the testing suites described in section 3.4 and explains how these tests generate p-values.

Chi-square (χ^2)

Most common one is probably the "Chi-square" test, that represents a basic method also used in connection with many other tests.

Considering a particular example, we can apply this test to die throwing. For an unbiased die (a die verifying H_0) each integer should occur with probability $P(i) = 1/6$ for $i \in [1, 6]$. Getting too many 1's or 2's in a large trial would suggest that the dice is not truly random, as rolling (say) twenty sixes in a row can happen one in about 3.66 x 10^{15} trials using a random die. Even obtaining exactly 1M in all bins over 6M rolls would also suggest that die was not random, as some fluctuation should occur compared to this special outcome.

The χ^2 test determines the probability distribution of observing any given excursion from the expected value if the die is presumed to be an unbiased perfect die.

In general, suppose that every observation can fall into one of k categories. We take n independent observations; this means that the outcome of one observation has absolutely no effect on the outcome of any of the others. Let p_s be the probability that each observation falls into category s, and let Y_s be the number of observations that actually do fall into category s. We form the statistic (Pearson's chi-squared test statistic [14, 15]) 3.3.1.

$$V = \sum_{s=1}^{k} \frac{(Y_s - np_s)^2}{np_s} \tag{3.3.1}$$

Comparing the resulting value with values of the chi-square distribution with same degrees of freedom we get a p-value, valid if n is large enough[7].

Frequency (Monobit) Test

Based on Chi-square, it tests the proportion of zeroes and ones in the entire sequence. First of all, zeros and ones of the input sequence (ε) are converted to values of -1 and +1. In a sequence of identically distributed Bernoulli random variables (X's or ε's, where $X = 2\varepsilon - 1$, and so $S_n = X_1 + \ldots + X_n = 2(\varepsilon_1 + \ldots + \varepsilon_n) - n$), the

[7] Knuth [14] suggests, as a rule of thumb, to take n large enough so that each of the expected values np_s is five or more.

probability of ones is 1/2. Using the classic De Moivre-Laplace theorem ([16]) for a sufficiently large number of trials, the distribution of the binomial sum, normalized by √n, is closely approximated by a standard normal distribution.

Thus, we get a P-value

$$2\left[1-\Phi\left(\frac{\|X_1+\ldots+X_n\|}{\sqrt{n}}\right)\right]=erfc\left(\frac{\|X_1+\ldots+X_n\|}{\sqrt{n}\sqrt{n}}\right)$$

where erfc is the complementary error function

$$erfc(z)=\frac{2}{\sqrt{\pi}}\int_z^{\infty}e^{-u^2}du$$

A variant of this test is the *Frequency Test within a Block*, where the original string is partitioned into N substrings, each of length M ($n=NM$). A Chi-square test is applied to every subsequence for a homogeneous match of empirical frequencies to the ideal 1/2. For each of these substring, the probability of ones is estimated by the observed relative frequency of 1's, π_i, i=1,...,N. With N degrees of freedom we get a P-value

$$\frac{\int_{\frac{\chi^2(val)}{2}}^{\infty}e^{-u}u^{\frac{N}{2-1}}du}{\Gamma\left(\frac{N}{2}\right)}=igamc\left(\frac{N}{2},\frac{\chi^2(val)}{2}\right)$$

with

$$\chi^2(val)=4M\sum_{i=1}^{N}\left[\pi_i-\frac{1}{2}\right]^2$$

and where igamc is the incomplete gamma function ([12, sec. 5.5.3]).

Runs Test

This test looks at "runs" defined as substrings of consecutive 1's and consecutive 0's, and considers whether the oscillation among such homogeneous substrings is too fast or too slow.

Test checks the distribution of the total number of runs V_n.

$$V_n=\sum_{k=1}^{n-1}r(k)+1$$

where, for $k=1,...,n-1$, $r(k)=0$ if $\epsilon_k=\epsilon_{k+1}$ and $r(k)=1$ if $\epsilon_k\neq\epsilon_{k+1}$.

Finally we get a P-value

$$erfc\left(\frac{|V_n(obs)-2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$$

A similar test is the *Largest Run of Ones in a Block*, that measures the length of the longest consecutive subsequence of ones after partitioning the sample n=NM into N substrings. Again, χ^2 is used to get a P-value.

Rank Test

Rank test, also called *Binary Matrix Rank Test*, checks for linear dependence among fixed-length substrings of the original sequence: constructs matrices of successive zeroes and ones from the sequence, and checks for linear dependence among the rows or columns of the constructed matrices. The test finally checks the deviation of the rank of these matrices from a theoretically expected value (using χ^2).

Spectral Test

Also called *Discrete Fourier Transform Test*, is one of the spectral methods. It is used to detect periodic features in the bit series (as explained in 1.2.6) that would indicate a deviation from the assumption of randomness (see Figure 3.3.1).

Template Matching

Counts the occurrences of a given aperiodic pattern, and checks if they are too many or too few. Pattern lengths normally go from 2 to 8. Central Limit Theorem and Chi-square distribution are used in this test. It comes in two variants: non-overlapping and overlapping template matching, the last one counting the occurrences of m-runs of ones, using a Compound Poisson distribution.

Universal Statistical Test (Maurer's Test)

The Universal Statistical Test ([31]) is a test introduced in 1992, closely related to the per-bit entropy, which is asserted to be (from Ueli Maurer, who introduced this test) "the correct quality measure for a secret-key source in a cryptographic application". Maurer's test is not designed to detect a specific pattern or a specific type of statistical defects. Instead, it is designed to detect a very general class of statistical defects "that can be modelled by an ergodic stationary source with finite memory", then subsuming some of the standard statistical tests.

This test is a compression-type test: the generator should pass the test if and only if its output sequence cannot be compressed significantly. This type of test, requiring a long sequence of bits, looks back through the entire sequence while walking through the test segment of L-bit blocks, checking for the nearest previous exact L-bit template match and recording the distance - in number of blocks - to that previous match. It then computes the average over all the expansion lengths by the number of test blocks.

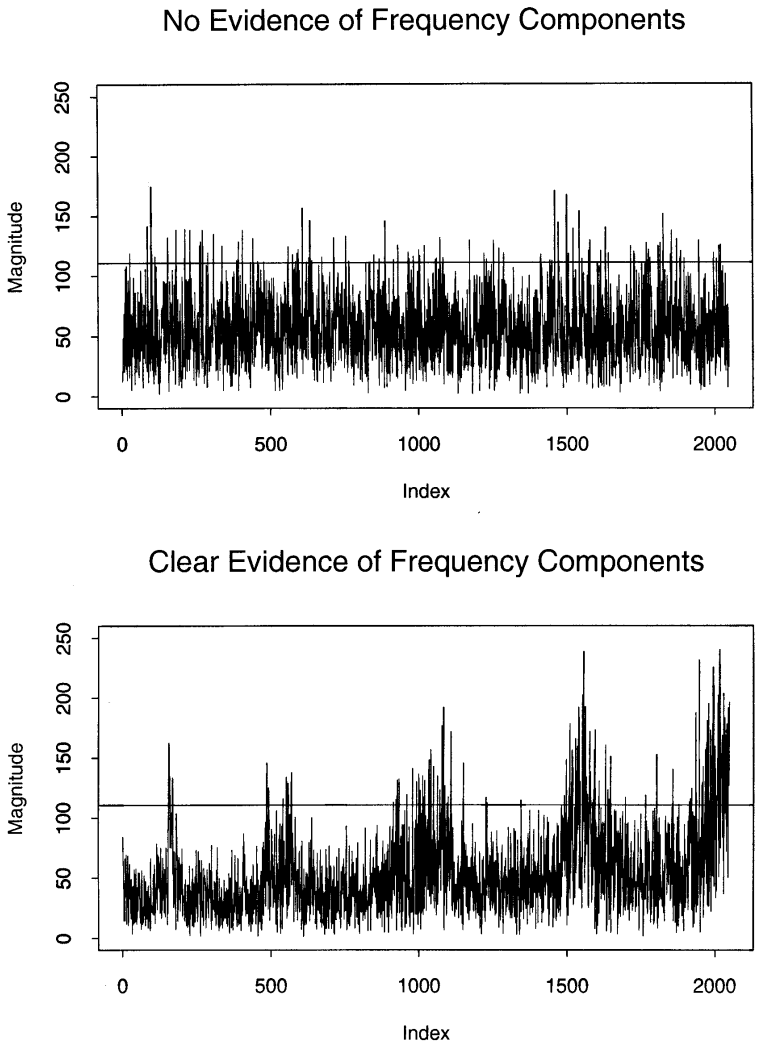

Figure 3.3.1: Two charts showing different results of Spectral Test.

Linear Complexity Test

This test uses linear complexity to test for randomness. Linear complexity is related to a part of keystream generators, the Linear Feedback Shift Registers. This register of length L consists of L delay elements, each having one input and one output. In every moment the output sequence $(\varepsilon_{L-1}, \ldots, \varepsilon_1, \varepsilon_0)$ satisfies the recurrent formula

$$\varepsilon_j = (c_1\varepsilon_j + c_2\varepsilon_{j-2} + \ldots + c_L\varepsilon_{j-L})\, mod 2$$

having initial state $(\varepsilon_L, \varepsilon_{L+1}, \ldots)$. Coefficients $(\mathrm{c}_1, \ldots, c_L)$ are coefficients of the polynomial connection corresponding to a given LSFR.

An LSFR is said to generate a given binary sequence if the sequence is the output of the LFSR for some initial state. The linear complexity of a given sequence is the length of the shortest LFSR that generates the sequence as its first n terms. There are some known formulas to evaluate the mean and variance linear complexity for truly random sequences. By using two different limiting distributions (for even and odd n, with n the length of the string) in conjunction to Chi-square it is possible to evaluate the P-values.

Serial Test or Bit Distribution Test

This test, as explained by [4], subsumes Frequency Test and many other tests (DNA, OPSO, OQSO), but it doesn't substitute them due to computational weight. It tests the uniformity of distributions of patterns of given lengths, by running through the set of all 2^{m} possible 0,1 vectors of lenght m. It uses a Chi-square type statistic (the distribution converges to the Chi-square distribution).

Approximate Entropy Test

The approximate entropy ApEn(m) measures the logarithmic frequency with which blocks of length m that are close together remain close together for blocks augmented by one position. Thus, small values of ApEn(m) imply strong regularity, or persistence, in a sequence. Alternatively, large values of ApEn(m) imply substantial fluctuation, or irregularity. For a fixed block length m, one should expect that in long random strings, ApEn(m)≈log2. With $\chi^2(obs) = n\,[log2 - ApEn(m)]$ the reported P-value is

$$igamc\left(2^{m-1}, \chi^2(obs)/2\right)$$

Cumulative Sums Test

It checks the maximum absolute value of the partial sums of the sequence with 0 and 1's respectively represented by -1 and +1. Large absolute values of this statistic indicate that there are either too many ones or too many zeros at the early stages. Instead, small values indicate that ones and zeros are intermixed too evenly. Dual test can be derived by reversing sequences with random walks.

With the statistic $z = max_{1 \leq k \leq n} |S_k| \, (obs)/\sqrt{n}$, the randomness hypothesis is rejected for large values of z, and the corresponding P-value is

$1 - H \left(max_{1 \leq k \leq n} |S_k| \, (obs)/\sqrt{n} \right)$ with $H(z) = \lim_{n \to \infty} P \left(\frac{max_{1 \leq k \leq n}}{|S_k|} \leq z \right)$.

Random Excursions Test

Test based on considering successive sums of the binary bits (as previously, ± 1) as a one-dimensional random walk. Thus, it detects deviations from the distribution of the number of visits of the random walk to a certain state.

Specific tests

Other tests were developed for the Diehard battery by George Marsaglia, and some of them are still available in Dieharder Suite:

- *Birthday spacings*: the spacings between randomly chosen points on a large interval should be asymptotically exponentially distributed (also named birthday paradox).
- *Overlapping permutations*: (subsumed by serial test), analyse sequences of five consecutive random numbers and checks the frequency of all 120 orderings.
- *Monkey tests*: some sequences are treated as "words". The number of words that don't appear should follow a known distribution.
- *Parking lot test*: randomly place unit circles in a 100 x 100 square. If a circle overlaps an existing one, it is replaced. After 12k trials, the number of successfully parked circles should follow a certain normal distribution.
- *Minimum distance test*: randomly place 8k points in a 10k x 10k square, then find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.
- *Random spheres test*: randomly choose 4k points in a cube of edge 1k. Center a sphere on each point, whose radius is the minimum distance to another point. The smallest sphere's volume should be exponentially distributed with a certain mean.

- *The squeeze test*: random floats on [0,1) multiplied with 2^{31} until 1 is reached. After repeating this 100k times, the number of floats needed to reach 1 should follow a certain distribution.

- *The craps test*: play 200k games of craps, counting the wins and the number of throws per game. Each count should follow a certain distribution.

3.4 Testing suites

Tests were performed using three different testing suites:

- Ent ([18]), a pseudorandom number sequence test program, based on 5 tests:
  - entropy (testing for optimum compression),
  - chi-square,
  - arithmetic mean,
  - Monte Carlo's method to compute π,
  - Serial correlation coefficient.
- NIST Statistical Testing Suite (STS, v. 2.2.1, [12]), created and maintained by the National Institute of Standards and Technology, containing almost every type of test described before, except for some Monte Carlo's tests.
- Dieharder (v. 3.31.1, [20]), a refactoring of the Diehard tests developed by George Marsaglia in 1995, containing every test described in 3.3. Every original Dieharder test has been revised to almost subsume NIST Suite ([4]).

Ent suite has been used to define the best parameters for LCG, using criteria expressed in 3.5, as the time needed to test every simulation with STS and Dieharder is much longer. These last suites are usually considered the state of art for random testing, covering the majority of the tests currently available.

3.5 Testbed description

Most of the suites explained in 3.4 needs a large amount of random data to output reliable statistical results. Furthermore, PariRandom performs better in big networks. Both these considerations lead us to the creation of a testbed that simulates the algorithm on a completely configurable network relying on a model that is as close as possible to a real environment running PariRandom.

Our testbed consists in a simulator mostly written in Java™and partly in bash scripting to interact with the testing suites. Every part of this simulator has been built to ensure the reproducibility of any experiment, that means that running simulator twice without changing parameters will produce same random values at any time.

A file, called *realrandom*, is built upon a large amount (~1GB size) of highly random bits[8]. This is used as a source of real entropy: *RealRandom* is a random generator that returns values sequentially taken from *realrandom* (the i byte generated by *RealRandom* is the i byte of *realrandom*).

The simulator creates a network of n nodes, each one with two local random generators:

- a *LCG* random generator;
- PariRandom applied to the same *LCG*.

LCG (Linear Congruential Generator) represents one of the oldest and best-known pseudorandom number generator algorithms. It's easy to implement and very fast.

It is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \pmod m$$

where X_n is the sequence of pseudorandom values, and
$m,\ 0 < m$ - the "*modulus*",
$a,\ 0 < a < m$ - the "*multiplier*",
$c,\ 0 \leq c < m$ - the "*increment*",
$X_0,\ 0 < X_0 < m$ - the "*seed*" or "*start value*"
are integer constants that specify the generator. LCG strength is extremely sensitive to the choice of the parameters c, m and a[9].

[8] This file derives from [22]. RANDOM.ORG is a true random number service that generates randomness via atmospheric noise.

[9] History of LCG and further information are available in [24].

m	2^{64}
a	1220703125
c	0

Table 3.2: Set of *LCG* parameters used by the simulator.

Setting up LGC parameters was probably the hardest problem during implementation. There is a long list of commonly used LCG parameters, but we needed two particular features:

- the generator has to output values in a space of 2^{64} bits, so it must be $m = 2^{64}$ [10];
- the quality of output values has to be enough to pass common statistical tests but also sufficiently low to show the improvement between this algorithm and PariRandom applied to the same algorithm.

Our final set of parameter (shown in table 3.2) is a small variation of the *APPLE LCG*[11].

As the idea of PariRandom algorithm is to capture the entropy of the network, we decided to model the different starting situation of each node of the network by setting a unique initial seed, generated by *RealRandom*.

To keep simulation as simple as possible, during *seed-update* process each node N_i (where i is the index of nodes in our array of nodes) contacts nodes $N_{j=i+1}$,N_{j+1},..N_{j+k-1}.

Compress functions f and g are simple XOR operations, without any use of the *shifting algorithm*.

More information about the structure of the simulator are available in following section 3.6.

[10] The long data type is a 64-bit signed two's complement integer.
[11] It is described in [23].

3.6 Testbed implementation

This section describes the implementation of the simulator described in previous section.

PariRandom Simulator is able to simulate the behaviour of the algorithm in a P2P network. It is written in Java TM®.

A MAIN class collects some inline parameters like the number of nodes in the network (n), the number of random values locally generated before a seed update, the size of the sequence of random values to be produced for each node and which testing suites has to be enabled. This helps automating executions with different parameters, due to the time needed by testing suites to produce results.

After doing that, this class calls the *startSimulation()* method in the ENGINE class. ENGINE.*startSimulation()* in order:

- instantiates a NETWORK object;
- instantiates n NODE objects, and adds each NODE to the NETWORK object;
- instantiates a RANDOMIZER object for each Node. RANDOMIZER class retrieve random values both from a "really random" file (method *getNextRealRandom()*) and a CUSTOMRANDOM class that extends different pseudo random algorithm like the custom LINEARCONGRUENTIALGENERATOR described in previous section. Each node is initialized with a random initial seed (generated by the "really random" file) used by LINEARCONGRUENTIALGENERATOR.
- starts a the main simulation loop:
  - a random NODE of the NETWORK is chosen randomly (with the "really random" file);
  - the chosen NODE generates two random values through the RANDOMIZER class, with and without *PariRandom* algorithm, and stores generated values through the STATS class for later analysis;
  - if the chosen NODE has generated enough values from last seed-update, it retrieves from NETWORK (through the *getSomeNeighbours()* method of NETWORK class) a list of neighbours to perform a seed-update by receiving random values from them. NETWORK.*getSomeNeighbours()* returns a defined number of nodes, chosen with the criteria described in section 3.5.
- after a defined number of iterations ENGINE stops the loop and asks STATS to perform the analysis of generated number values with different testing suites

(through methods STATS.*runEntStats()*, STATS.*runNistStats()* and STATS.*runDieharderStats()*).

Finally, a UTILS class is used to debug the simulator.

3.7 Testing results

We performed different type of tests to analyse the behaviour of PariRandom in different situations.

First of all, we analysed the difference between the quality of the entropy obtained using the LCG with and without PariRandom for different number of nodes.

Dieharder was run on files containing from 10M to 100M random "longs" generated by a random node (the first node instantiated by the simulator), after a startup period of 100K generations[12].

To evaluate the time needed to make the system operative, we can estimate a seed-update process every minute (really affordable if we are under a common P2P network with file sharing and other bandwidth consuming services), that is actually activated after 5 random values generated, meaning a total time of ~14 days[13].

Results of Dieharder battery of tests are available in figures 3.7.1,3.7.2, and table 3.3.

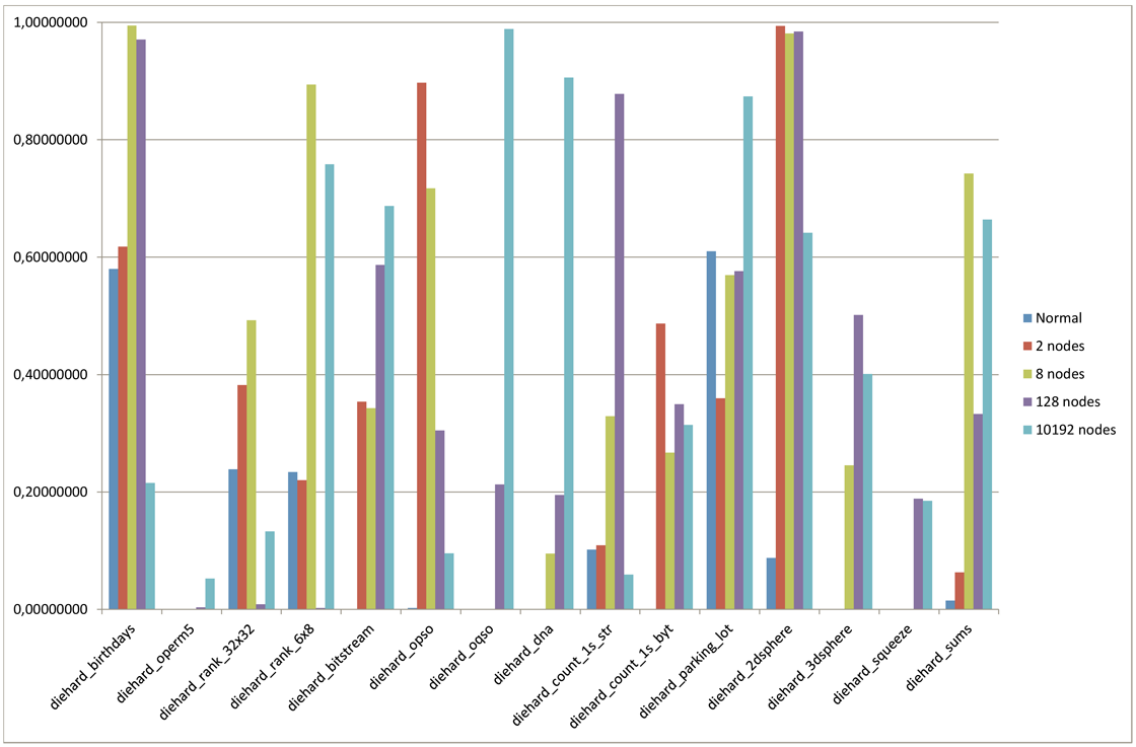

Figure 3.7.1: First part of Dieharder tests using LCG (3.2) without and with PariRandom for different amounts of nodes. 10M random cycles performed.

These charts exhibit how the algorithm consistently increase the quality of generated values, and how this quality is related to the number of nodes. Remembering

[12]The startup period has been introduced to misalign nodes. It represents a period where generated values are not stored for analysis. By introducing it we avoid having the first part of PariRandom generated random values too similar to values generated without PariRandom.

[13]100K (generated values)/5 (values per minute)/60 (minutes per hour)/24 (hours per day)

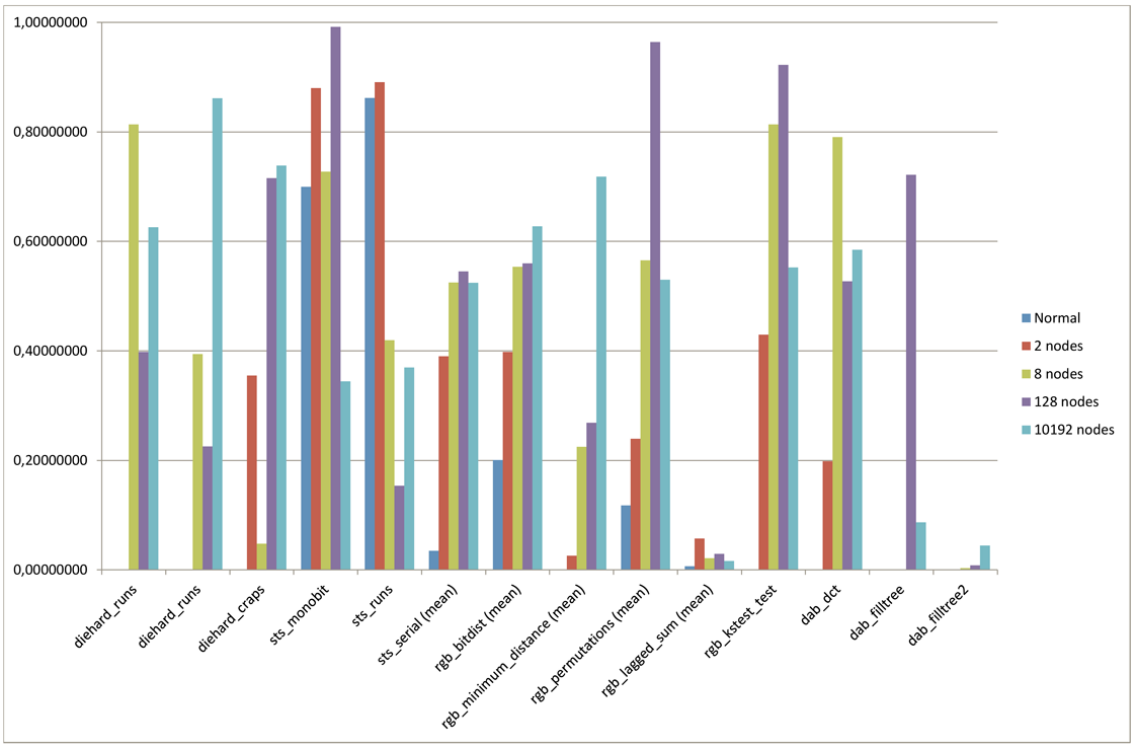

Figure 3.7.2: Second part of Dieharder tests using LCG (3.2) without and with PariRandom for different amounts of nodes. 10M random cycles performed.

that P-values shown are second order P-values (obtained by evaluating the uniformity of a sequence of P-values, as described in 3.2), we know that they should be in $[0 + \alpha, 1 - \alpha]$ to pass the test, with $\alpha = 0.01$ (and with $\alpha = 0.01$ to pass the test with some suspect). In Opso, Count 1st bit, and some rgb tests, after failing without PariRandom, the test passes by running PariRandom between two nodes. Tests like Filtree, DNA and Oqso pass only with PariRandom running on more than two nodes, showing how, as also stated in the theoretical analysis, the general performance of the algorithms depends on the amount of entropy available in all the network (and thus on the number of nodes).

Same results have been confirmed running NIST sts, as shown in figure 3.7.3 and in table 3.4.

In fact, we see the same behaviour for BlockFrequency (even with 8 nodes it doesn't pass the test), Rank, FFT, ApproximateEntropy and, partly, LinearComplexity tests.

A second run of tests were performed using a HRNG ([29]), described in appendix A. A random node, far from the one being analysed by testing suites (they never communicates directly), generated "real random" values. We evaluated the performance in this situation and compared them to the performance obtained before.

The results (figures 3.7.4, 3.7.5 and table 3.5) show that the algorithm is capa-

	Normal	2 nodes	8 n.	128 n.	10192 n.
diehard_birthdays	0,57993126	0,61809426	0,99469459	0,97088150	0,21561892
diehard_operm5	0,00000000	0,00000000	0,00008718	0,00365510	0,05228547
diehard_rank_32x32	0,23882055	0,38231099	0,49243469	0,00855250	0,13312920
diehard_rank_6x8	0,23402214	0,22016080	0,89410401	0,00270962	0,75809936
diehard_bitstream	0,00000000	0,35362321	0,34297846	0,58673746	0,68737222
diehard_opso	0,00260542	0,89731439	0,71716523	0,30494085	0,09529627
diehard_oqso	0,00000000	0,00000000	0,00000000	0,21289743	0,98845983
diehard_dna	0,00000000	0,00000000	0,09509280	0,19515615	0,90636024
diehard_count_1s_str	0,10188830	0,10909105	0,32924689	0,87821317	0,05930009
diehard_count_1s_byt	0,00000000	0,48679918	0,26723470	0,34964937	0,31460563
diehard_parking_lot	0,61013501	0,35982598	0,56923274	0,57621075	0,87393908
diehard_2dsphere	0,08779581	0,99412555	0,98150985	0,98442140	0,64128178
diehard_3dsphere	0,00000000	0,00000017	0,24523349	0,50140017	0,40128512
diehard_squeeze	0,00000000	0,00000087	0,00004177	0,18870815	0,18481533
diehard_sums	0,01522474	0,06279105	0,74277476	0,33300861	0,66400870
diehard_runs	0,00020759	0,00006878	0,81359683	0,39788941	0,62598481
diehard_runs	0,00000015	0,00064044	0,39402782	0,22546016	0,86181843
diehard_craps	0,00025752	0,35530424	0,04753820	0,71545365	0,73875300
sts_monobit	0,69977162	0,88022559	0,72711275	0,99157661	0,34457983
sts_runs	0,86189371	0,89084975	0,41932705	0,15376765	0,36966530
sts_serial (mean)	0,03503928	0,38982038	0,52505177	0,54492684	0,52435071
rgb_bitdist (mean)	0,20032988	0,39781967	0,55389280	0,55995752	0,62725530
rgb_minimum_distance (mean)	0,00019002	0,02563203	0,22468404	0,26854295	0,71816564
rgb_permutations (mean)	0,11780175	0,23945652	0,56555984	0,96447255	0,53006627
rgb_lagged_sum (mean)	0,00678588	0,05726743	0,02116248	0,02900679	0,01604308
rgb_kstest_test	0,00000000	0,42937348	0,81361929	0,92232610	0,55254503
dab_dct	0,00000000	0,19860244	0,79028643	0,52716764	0,58481847
dab_filltree	0,00000000	0,00000000	0,00000000	0,72150588	0,08664979
dab_filltree2	0,00000000	0,00017358	0,00345207	0,00822036	0,04446286

Table 3.3: Results of Dieharder tests using LCG (3.2) without and with PariRandom for different amounts of nodes. 10M random cycles performed.

ble of sharing the entropy of nodes in a network. Especially craps, runs monobit and filltree tests show a general improvement if the hardware number generator is available in the network. Thus, if a node has a good quality entropy source, it will improve the quality of every other random generator in the network.

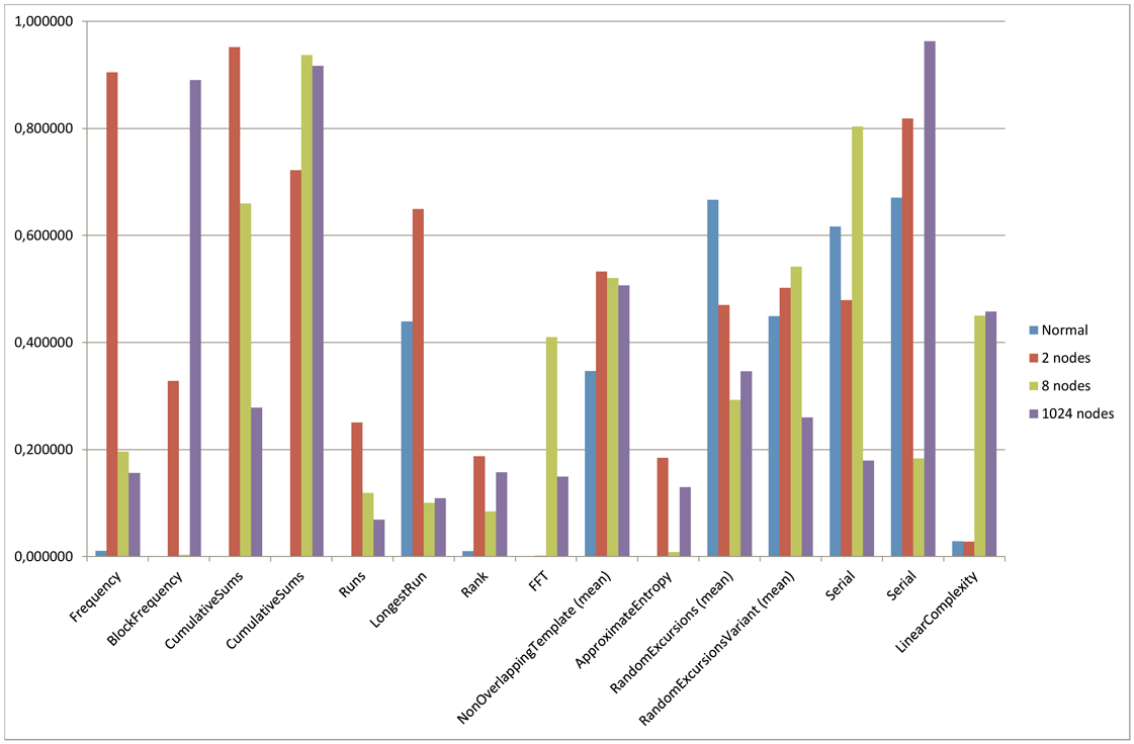

Figure 3.7.3: NIST tests using LCG (3.2) without and with PariRandom for different amounts of nodes. 10M random cycles performed.

	Normal	2 nodes	8 n.	128 n.
Frequency	0,010681	0,904708	0,195864	0,156373
BlockFrequency	0,000019	0,328297	0,003371	0,890582
CumulativeSums	0,000003	0,952152	0,660012	0,278461
CumulativeSums	0,000003	0,721777	0,936823	0,916599
Runs	0,000695	0,250558	0,118812	0,068999
LongestRun	0,439122	0,649612	0,100709	0,109435
Rank	0,010165	0,187581	0,084551	0,157731
FFT	0,000000	0,001679	0,410055	0,149495
NonOverlappingTemplate (mean)	0,346775	0,532811	0,520528	0,506477
ApproximateEntropy	0,000000	0,184549	0,008691	0,130025
RandomExcursions (mean)	0,666824	0,469759	0,292628	0,346434
RandomExcursionsVariant (mean)	0,448933	0,502349	0,542012	0,259871
Serial	0,616305	0,478839	0,803720	0,179584
Serial	0,670396	0,818343	0,183547	0,962688
LinearComplexity	0,028434	0,028244	0,450297	0,457825

Table 3.4: Results of NIST tests using LCG (3.2) without and with PariRandom for different amounts of nodes. 10M random cycles performed.

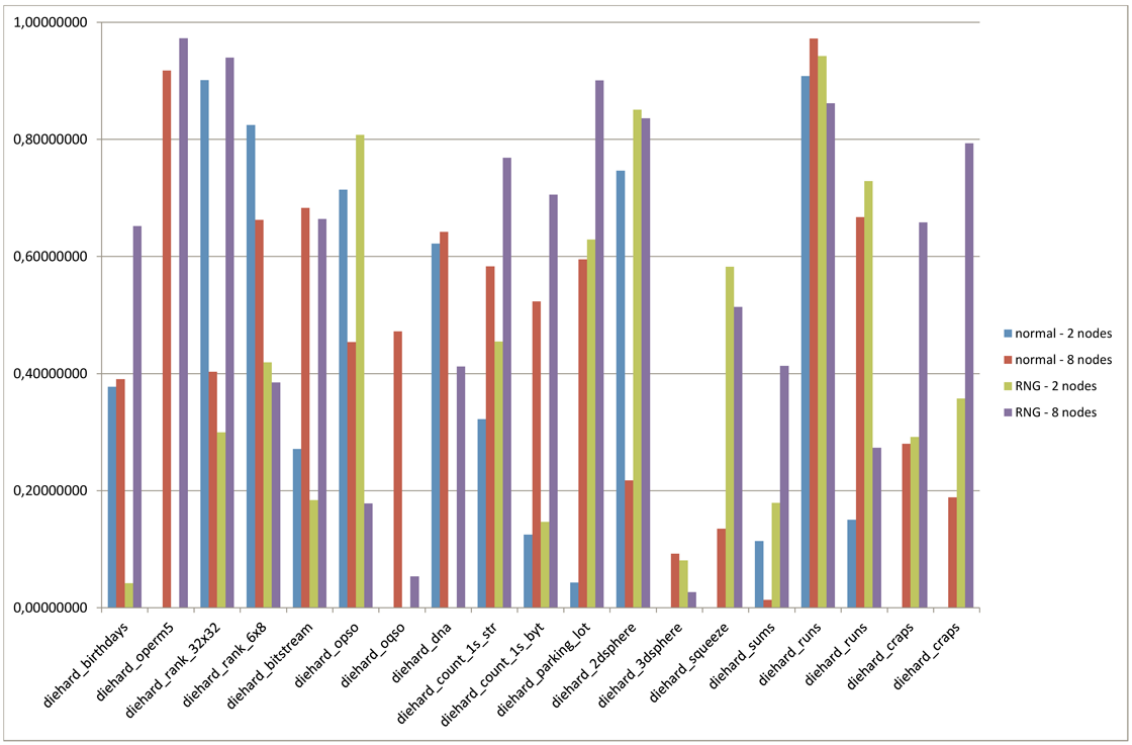

Figure 3.7.4: First part of Dieharder tests using LCG (3.2) without and with the hardware generator for different amounts of nodes. 100M random cycles performed.

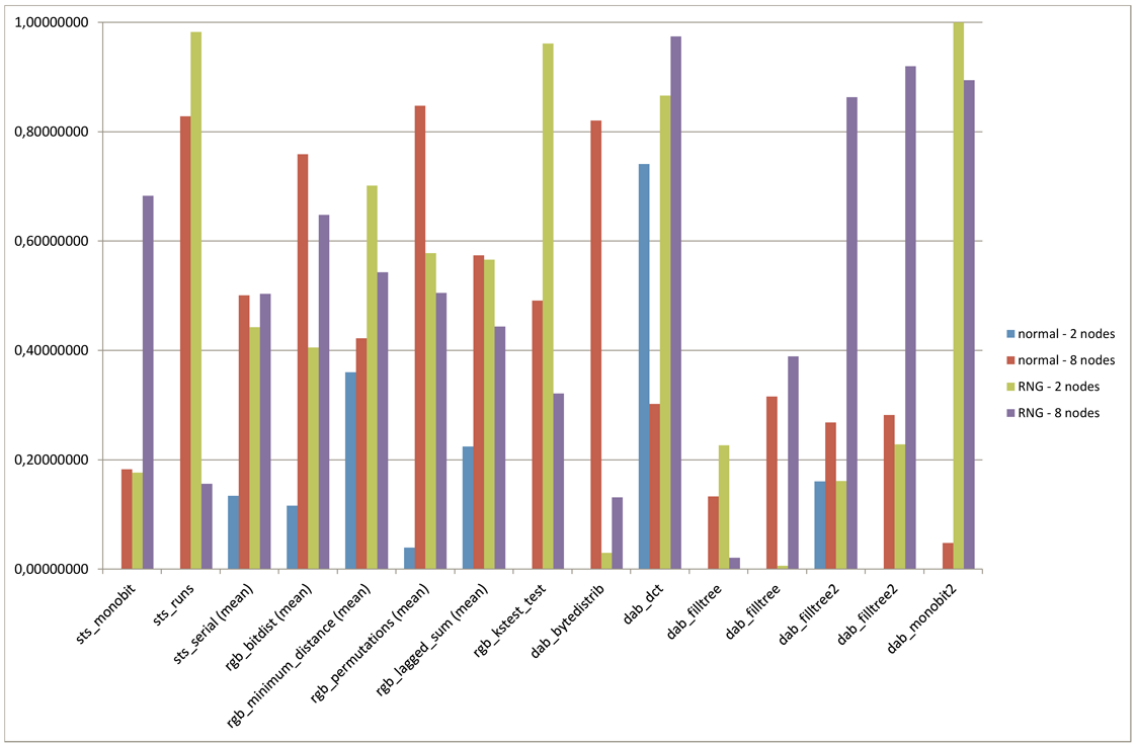

Figure 3.7.5: Second part of Dieharder tests using LCG (3.2) without and with the hardware generator for different amounts of nodes. 100M random cycles performed.

	normal - 2 nodes	normal - 8 nodes	RNG - 2 nodes	RNG - 8 nodes
diehard_birthdays	0,37756209	0,39065593	0,04184346	0,65181860
diehard_operm5	0,00000000	0,91778804	0,00000000	0,97266654
diehard_rank_32x32	0,90125439	0,40340301	0,29953394	0,93989281
diehard_rank_6x8	0,82455142	0,66281353	0,41886598	0,38509029
diehard_bitstream	0,27124515	0,68295332	0,18375017	0,66414935
diehard_opso	0,71418903	0,45356503	0,80759428	0,17791836
diehard_oqso	0,00000000	0,47203297	0,00000000	0,05354777
diehard_dna	0,62220146	0,64232614	0,00000069	0,41204709
diehard_count_1s_str	0,32224153	0,58302463	0,45506579	0,76890477
diehard_count_1s_byt	0,12521680	0,52322192	0,14639649	0,70592988
diehard_parking_lot	0,04286309	0,59516268	0,62905479	0,90083553
diehard_2dsphere	0,74674001	0,21753620	0,85092709	0,83613733
diehard_3dsphere	0,00000000	0,09230081	0,08097332	0,02668006
diehard_squeeze	0,00000000	0,13486156	0,58272724	0,51389152
diehard_sums	0,11378247	0,01362901	0,17916598	0,41328812
diehard_runs	0,90826541	0,97241983	0,94243966	0,86174036
diehard_runs	0,15032384	0,66722183	0,72883370	0,27332748
diehard_craps	0,00000000	0,28037417	0,29150115	0,65854803
diehard_craps	0,00000527	0,18871222	0,35732321	0,79342718
marsaglia_tsang_gcd	0,00000000	0,00002180	0,00009769	0,00000012
marsaglia_tsang_gcd	0,00000000	0,00996180	0,04396175	0,00000000
sts_monobit	0,00000000	0,18269410	0,17667644	0,68262041
sts_runs	0,00000000	0,82854738	0,98242959	0,15629937
sts_serial (mean)	0,09991802	0,69045539	0,49761617	0,57354135
rgb_bitdist (mean)	0,12141830	0,57599217	0,62120610	0,46745822
rgb_minimum_distance (mean)	0,02501114	0,27884205	0,00890176	0,08599449
rgb_permutations (mean)	0,25132752	0,72208269	0,40956714	0,58254932
rgb_lagged_sum (mean)	0,28907519	0,61340176	0,69944185	0,55268428
rgb_kstest_test	0,00000000	0,49122878	0,96135044	0,32119157
dab_bytedistrib	0,00000000	0,82051284	0,02973236	0,13130025
dab_dct	0,74105826	0,30237519	0,86629689	0,97461501
dab_filltree	0,00000000	0,13312368	0,22689051	0,02094536
dab_filltree	0,00000000	0,31580186	0,00640534	0,38929576
dab_filltree2	0,16037900	0,26860716	0,16120437	0,86321729
dab_filltree2	0,00000000	0,28168490	0,22828568	0,91953914
dab_monobit2	0,00000000	0,04786703	0,99999994	0,89423498

Table 3.5: Values of Dieharder tests using LCG (3.2) without and with the hardware generator for different amounts of nodes. 100M random cycles performed.

Chapter 4

Conclusions

PariRandom describes a new way to use a network as an entropy source. Previous solutions only involved transmission delays. Every RNG (either hardware or software) can work over the PariRandom framework, guaranteeing *at least* the same quality it would provide in the absence of PariRandom even if the entire network is *completely* compromised. The only drawback is the increase in network traffic but it can generally be reduced to negligible levels using the piggyback technique described in Section 2.1.

We presented a theoretical analysis to understand entropy gain performance, providing a computational model that can be used to perform statistical analysis, following the proofs of theorems described in sections 2.2 and 2.3. Due to the difficulty of using that approach for large network models, we performed tests using the best random testing suites available, after generating big random data files with the PariRandom simulator described in section 3.6 (simulating up to ~100K nodes).

Results obtained prove the quality demonstrated with the theoretical analysis, even if, as explained in 3.2, they may not be able to spot every pseudo algorithmic artifact.

Further work

There are three main working guidelines.

First, as explained in section 3.2, it's difficult to evaluate the real entropy gain using statistical testing suites. That's why the creation of a simulator for the theoretical model would allow a better investigation of the performance of PariRandom in big networks. This simulator should use theorems explained in 2.3 to mathematically evaluate the level of entropy of each node in the network after computing some statistical measures, like the bias of the seed in each node and the correlation between these seeds.

A second line regards the size of the simulations. A more powerful cluster should

be used to evaluate the entropy quality produced by PariRandom Simulator if millions of nodes are connected to the network. While the simulator is fast to generate this data, statistical testing suites needs weeks on a powerful computer to produce results.

Finally, the spread of PariPari software will raise the opportunity to test PariRandom on a real testbed made of nodes with different randomization techniques within a real network. Testing the entropy quality of some nodes in this network will show some realistic results that can't be obtained with previous two methods. It will also allow to evaluate with better confidence the startup time needed to reach a considerable entropy level.

Appendix A

Simtec Electronics Entropy Key

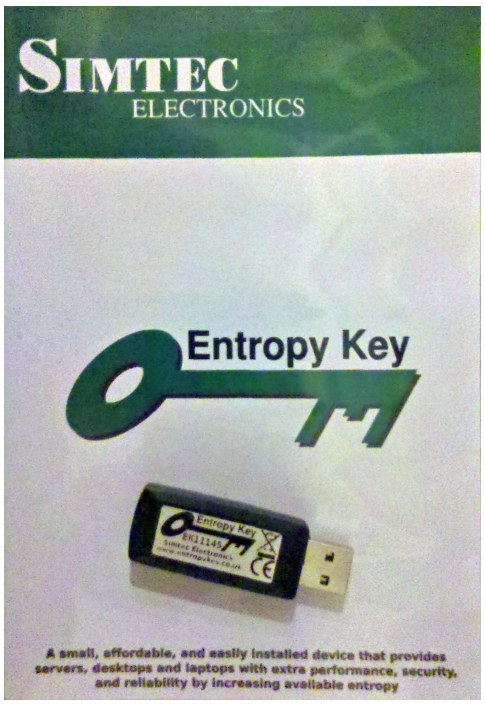

Figure A.0.1: Photo of Simtec Entropy Key

A hardware random generator has been used due to high amount of random data needed for some simulations.

The HRNG used was the Simtec Electronics Entropy Key, a USB device containing two high-quality quantum noise sources and an ARM Cortex CPU that actively measures and checks all generated random numbers, before encrypting them and sending them to the server. It also detects attempts to corrupt or sway the device. It aims towards FIPS-140-2 Level 3 compliance with some elements of Level 4, including tamper-evidence, tamper-proofing, role-based authentication, and environmental attacks.

It works by using a P-N semiconductor junctions reverse biased with a high enough voltage to bring them near to, but not beyond, breakdown in order to gen-

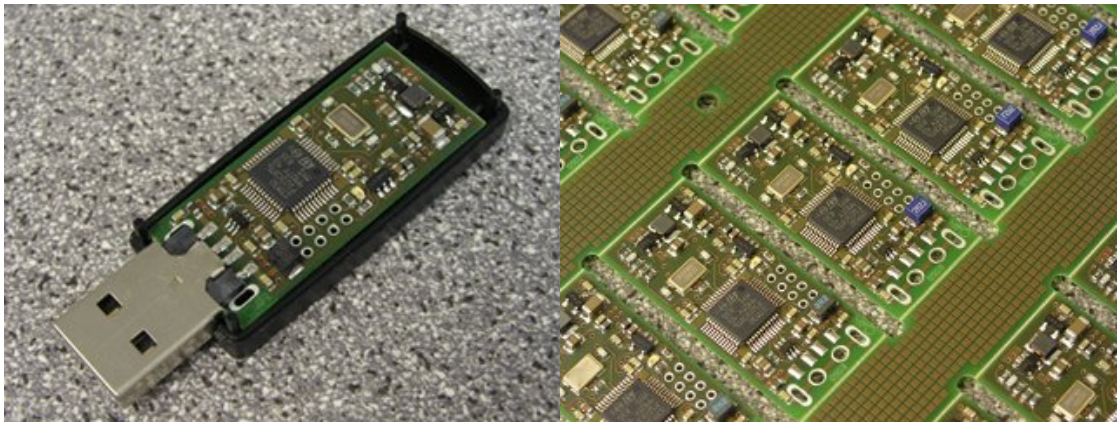

Figure A.0.2: Sections of the USB device.

erate noise. In other words, it has a pair of devices that are wired up in such a way that as a high potential is applied across them, where electrons do not normally flow in this direction and would be blocked, the high voltage compresses the semiconduction gap sufficiently that the occasional stray electron will quantum tunnel through the P-N junction (sometimes referred to as avalanche noise). It is impossible to predict when this happens, and this type of entropy is measured by the key.

Acknowledgments

The author would thank Enoch Peserico, Paolo Bertasi and Michele Bonazza for giving him the opportunity to carry out this work on an argument of full personal interest and for the collaboration in the creation of the algorithm.

Bibliography

[1] Baruch Awerbuch and Christian Scheideler, *Robust Random Number Generation for Peer-to-Peer Systems*, Theoretical Computer Science, v.410 n.6-7, p.453-466, February, 2009.

[2] Santha and Vazirani, *Generating Quasi-Random Sequences from Slightly-Random Sources*, 1984.

[3] Andrew C. You, *Theory and Applications of Trapdoor Functions*, 1982.

[4] Robert G. Brown, *DieHarder: A Gnu Public Licensed Random Number Tester*, Duke University Physics Department, 2008.

[5] M. Blum and S. Micali, *How To Generate Cryptographically Strong Sequences of Pseudo-Random Bits*, FOCS, 1982.

[6] Robert B Davie, *Exclusive OR (XOR) and hardware random number generators*, 2002.

[7] John Viega, *Practical Random Number Generation in Software*, Virginia Tech, 2003.

[8] Juan Soto, *Statistical Testing of Random Number Generators*, National Institute of Standards & Technology, 1999.

[9] *Bull Mountain Software Implementation Guide*, Intel, 2011. Available at http://software.intel.com/en-us/articles/download-the-latest-bull-mountain-software-implementation-guide/.

[10] Alfred J. Menezes, *Handbook of Applied Cryptography*, CRC Press, 1996.

[11] Marek Obitko, *Crossover and mutation*, 1998. Retrieved 10-3-2012 from http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php.

[12] NIST group, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, National Institute of Standards and Technology, 2010.

[13] Song-Ju Kim, Ken Umeno, and Akio Hasegawa. *Corrections of the NIST Statistical Test Suite for Randomness*, 2004.

[14] Donald Knuth, *The Art of Computer Programming, Seminumerical Algorithms*, Volume 2, 3rd edition, Addison Wesley, Reading, Massachusetts, 1998.

[15] R.L. Plackett, *Karl Pearson and the Chi-Squared Test*, International Statistical Review, 1983.

[16] W. Feller, An Introduction to Probability Theory and Its Applications, Wiley, 1968.

[17] Zvi Gutterman, Benny Pinkas and Tzachy Reinman, *Analysis of the Linux Random Number Generator*, 2006.

[18] John Walker, *ENT: A Pseudorandom Number Sequence Test Program*, 2008. Available at http://www.fourmilab.ch/random/.

[19] *NIST: Statistical Test Suite*. Available at http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.

[20] *Dieharder: A Random Number Test Suite*. Available at http://www.phy.duke.edu/~rgb/General/dieharder.php.

[21] Lenore Blum, Manuel Blum, and Michael Shub, *A Simple Unpredictable Pseudo-Random Number Generator*, SIAM Journal on Computing, 1986.

[22] Mads Haahr, *Random.org*. Retrieved 10-4-2012 from http://www.random.org/analysis/.

[23] Karl Entacher, *Classical LCGs*, 2000. Retrieved 10-4-2012 from http://random.mat.sbg.ac.at/results/karl/server/node4.html.

[24] Vivien Challis, *Pseudo-Random Number Generators*, 2008.

[25] Neyman and Pearson, *On the Use and Interpretation of Certain Test Criteria for Purposes of Statistical Inference*, Cambridge University Press, 1967.

[26] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods (2nd ed.)*, Springer, 2004..

[27] *USA - Federal Standard 1037C.* Retrieved 11-4-2010 from http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm.

[28] *JXTA*, The Language and Platform Independent Protocol for P2P Networking. Available at http://jxta.kenai.com/.

[29] *Simtec Entropy Key.* Available at http://www.entropykey.co.uk/.

[30] *Federal Information Processing Standards 140,* FIPS PUB 140-2, 2001. Retrieved 4-4-2012 from http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[31] Maurer, U. (1992). *A universal statistical test for random bit generators.* J. Cryptology, 5, no. 2, 89–105.

[32] Michele Bonazza, *PARICORE*, 2009.

List of Figures

List of Tables

Index