Randomness tests: theory and practice

Andrei Romashchenko, Alexander Shen[1]

[1]LIRMM, University of Montpellier, CNRS, Montpellier, France. Supported by RaCAF ANR-15-CE40-0016-01 grant

To Yury Gurevich, with whom we have had a lot of interesting and sometimes heated discussions on many topics, including randomness

Abstract

The mathematical theory of probabilities does not refer to the notion of an *individual* random object. For example, when we toss a fair coin n times, all 2^n bit strings of length n are equiprobable outcomes and none of them is more "random" than others. However, when testing a statistical model, e.g., the fair coin hypothesis, we necessarily have to distinguish between outcomes that contradict this model, i.e., the outcomes that convince us to reject this model with some level of certainty, and all other outcomes. The same question arises when we apply randomness tests to some hardware random bits generator.

A similar distinction between random and non-random objects appears in algorithmic information theory. Algorithmic information theory defines the notion of an individual random sequence and therefore splits all infinite bit sequences into random and non-random ones. For finite sequences there is no sharp boundary. Instead, the notion of *randomness deficiency* can be defined, and sequences with greater deficiency are considered as "less random" ones. This definition can be given in terms of randomness tests that are similar to the practical tests used for checking (pseudo)random bits generators. However, these two kinds of randomness tests are rarely compared and discussed together.

In this survey we try to discuss current methods of producing and testing random bits, having in mind algorithmic information theory as a reference point. We also suggest some approach to construct robust practical tests for random bits, and describe software tools that we have developed.

The short version of this report appeared as [55].

Contents

Chapter 1

Statistics and tests

1.1 Testing a statistical hypothesis

Probability theory is nowadays considered as a special case of measure theory: a random variable is a measurable function defined on some probability space that consists of a set Ω, some σ-algebra of the subsets of Ω, and some σ-additive measure defined on this σ-algebra. A random variable determines a probability distribution on the set of possible values.

When probability theory is applied to some "real world" case, it provides a statistical hypothesis that is a mathematical model of the process. For example, for n trials and the fair coin the corresponding model is the uniform distribution on the set $\mathbb{B}^n$ of all possible outcomes, i.e., on all n-bit binary strings. Each string has probability 2^{-n}. The set $\mathbb{B}^n$ can be considered as a probability space, and ith coin tossing is represented by a random variable ξ_i defined on this space: $\xi_i(x_1x_2\ldots x_n) = x_i$.

Having a mathematical model for a real-life process, we need some way to check whether this model is adequate or not. Imagine that somebody gives us a coin, or a more advanced random bits generator. This coin can be asymmetric, and the bit generator can be faulty. To check whether this is the case, we need to perform some experiment and look at its results. Assume, for example, that we toss a coin and get 85 heads in a row, as it happened to Rosencrantz and Guildenstern in Stoppard's play [58, Act 1]. Should we reject the fair coin hypothesis? Probably we should — but how can we justify this answer? One can argue that for a fair coin such an outcome is hardly possible, since its probability is negligible, namely, equals 2^{-85}. However, any other sequence of 85 heads and tails has the same negligible probability — so why this reasoning cannot be applied to any other outcome?

To discuss this problem in a more general case, let us introduce suitable terminology. Consider some set X of possible outcomes. We assume that X is finite. Fix some *statistical model* P, i.e., a hypothetical probability distribution on X. A *randomness test* is an event $T \subset X$ that has small probability according to P. If an experiment produces an outcome that belongs to T, the test is not passed, and we may reject P. This approach has two main problems. The first problem, mentioned earlier, is that in most cases every individual outcome $x \in X$ has negligible probability, so the singleton $\{x\}$ is a test that can be used to reject P. We discuss this problem later. Now let us comment on the other problem: how to choose the threshold value for the probability, i.e., how small should be the probability of T to consider T as a valid randomness test.

In practice, the statistical model is often called the *null hypothesis*. Usually we have some experimental data that, as we hope, exhibit some effect. For example, we may hope that a new drug increases the survival rate, and indeed we see some improvement in the experimental group. However, this improvement could be just a random fluctuation while in fact the survival probability remains unchanged. We formulate the null hypothesis based on the old value of the survival probability and then apply some test. The rejection of the null hypothesis means that we do not consider the data as a random fluctuation, and claim that the new drug has at least some effect.[1]

[1]Of course, we simplified the situation in this example: rejecting the null hypothesis saying that the drug has no

In this approach, the choice of the threshold value obviously should depend on the importance of the question we consider. Any decision based on statistical considerations is inherently unreliable, but an acceptable level of this unreliability depends on the possible consequences of a wrong decision. The more important the consequences are, the smaller threshold for statistical tests is needed. The choice of the threshold value is often debated. For example, there is a paper [3] signed by 72 authors that proposes "to change the $\langle ... \rangle$ threshold for statistical significance from 0.05 to 0.005 for claims of new discoveries". It may look ridiculous — obviously both the old threshold and the new one are chosen arbitrarily — but it reflects the existing situation in natural sciences. Other people point out that fixing a threshold, whatever it is, is a bad practice [1].

1.2 Randomness tests

Now let us address the other problem mentioned above. After the experiment is made, we can find a set T of very small measure that contains the actual outcome of the experiment, and declare it to be a randomness test. For example, Rosencrantz could toss a coin 85 times, write down the sequence x of heads and tails obtained, and then try to convince Guildenstern that the fair coin hypothesis should be rejected, because the set $T = \{x\}$ is a test that has probability 2^{-85} according to the hypothesis and still the actual outcome is in T.

This argument is obviously wrong, but it is not that easy to say what exactly is wrong here. The simplest — and rather convincing — answer is that the *test should be fixed before the experiment*. Indeed, if Rosencrantz showed some sequence of heads and tails to Guildenstern and *after that* the coin tossing gave exactly the same sequence, this would be a very convincing reason to reject the null hypothesis of a fair coin.

Still this answer is not universal. Imagine that we buy a book called "A Million Random Digits"[2]. We open it and find that it is filled with zeros. Do we have reasons to complain? If we have said "Look, this book is suspicious; may be it contains only zeros" *before opening the book*, then we definitely do — but what if not? Or what if we find out that the digits form the decimal expansion of 8.5π? Note that this hypothesis hardly can come to our mind before we study the book carefully.

Sometimes the experiment is already in the past, so we look at the data already produced, like Kepler did when analyzing the planet observations. Probably, in this case we should require that the test is chosen without knowing the data, but it is (a) more subjective and (b) rarely happens in practice, usually people do look at the data before discussing them.

On the other hand, even the test formulated before the experiment could be dubious. Imagine that there are many people waiting for an outcome of the coin tossing, and each of them declares her own statistical test, i.e., a set of outcomes that has probability at most ε. Here ε is some small number; it is the same for all tests. After the experiment it is found that the outcome fails one of the tests, i.e., belongs to the small set declared by one of the observers. We are ready to declare that the null hypothesis is rejected. Should we take into account that there were many tests? One can argue that the probability to fail at least one of the N tests is bounded by $N\varepsilon$, not ε, so the result is less convincing than the same result with only one observer. This correction factor N is often called the *Bonferroni correction* and is quite natural. On the other hand, the observer who declared the failed test could complain that she did not know anything about other observers and it is a completely unacceptable practice if actions of other people beyond her control and knowledge are considered as compromising her findings. And it is difficult to answer in a really convincing way to this complaint.

In fact, this is not only a philosophical question, but also an important practical one. If a big laboratory with thousand researchers uses threshold value 0.05 for statistical significance, then we

effect is not enough. We want to be convinced that the drug effect is beneficial.

[2]By the way, one can still buy such a book [48] now (August 2019) for 50.04 euro, or 903.42 euro, if you prefer the first edition, but agree to get a second-hand copy.

could expect dozens of papers coming from this lab where this threshold is crossed — even if in fact the null hypothesis is true all the time.

There are no universally accepted or completely convincing answers to these questions. However, there is an important idea that is a philosophical motivation for algorithmic information theory. We discuss it in the next section.

1.3 "Remarkable" events as tests

Recall the example with zeros in the table of random numbers and the corresponding singleton test that consists of the zero sequence. Even if we have not explicitly formulated this test before reading the table, one can say that this test is so simple that it *could* be formulated before the experiment. The fact that all outcomes are heads/zeros is remarkable, and this makes this test convincing.

This question is discussed by Borel [7]. He quotes Bertrand who asked whether we should look for a hidden cause if three stars form an equilateral triange. Borel notes that nobody will find something strange if the angle between two stars is exactly $13°42'51.7''$, since nobody would ask whether this happens or not before the measurement ("car on ne se serait jamais posé cette question précise avant d'avoir mesuré l'angle"). Borel continues:

> La question est de savoir si l'on doit faire ces mêmes réserves dans le cas où l'on constate qu'un des angles du triangle formé par trois étoiles a une valeur *remarquable* et est, par exemple, égal à l'angle du triangle équilatéral ⟨...⟩ Voici ce que l'on peut dire à ce sujet : on doit se défier beaucoup de la tendance que l'on a à regarder comme *remarquable* une circonstance que l'on n'avait pas précisée *avant l'expérience*, car le nombre des circonstances qui peuvent apparaître comme remarquables, à divers points de vue, est très considérable [7, p. 112–113][3]

This quotation illustrates a trade-off between the probability of the event specified by some randomness test and its "remarkability": if there are N events of probability at most ε that are "as remarkable as the test event" (or "more remarkable"), then the probability of the combined test event is at most $N\varepsilon$. In other words, we should consider not an individual test, but the union of all tests that have the same probability and the same (or greater) "remarkability".

The problem with this approach is that we need to quantify somehow the "remarkability". The algorithmic information theory suggests to take into account the Kolmogorov complexity of the test, i.e., to count the number of bits needed to specify the test. More remarkable tests have shorter descriptions and smaller complexity. The natural way to take the complexity into account is to multiply the probability by $O(2^n)$ if the complexity of the test is n, since there is at most $O(2^n)$ different descriptions of size at most n bits and therefore at most $O(2^n)$ tests of complexity at most n.

However, the word "description" is too vague. One should fix a "description language" that determines which test corresponds to a given description, i.e., to a given sequence of bits. Algorithmic information theory does not fix a specific description language; instead, it defines a class of description languages and proves that there are *optimal* description languages in this class. Optimality is understood "up to $O(1)$ additive term": a language L is optimal if for any other language L' in the class there exist a constant c (depending on L') with the following property: if a test T has description of length k via L', it has a description of length at most $k + c$ via L. This implies that two different optimal languages lead to complexity measures that differ at most by $O(1)$ additive term.

[3] The question is whether we should have the same doubts in the case where one of the angles of a triangle formed by tree stars has some *remarkable* value, for example, is equal to the angle of an equilateral triangle. ⟨...⟩ Here we could say the following: one should resist strongly to the tendency to consider some observation that was not specified *before the experiment* as remarkable, since the number of circumstances that may look remarkable from different viewpoints is quite significant.

Probably this $O(1)$ precision is the best thing a mathematical theory could give us. However, if one would like to define "the gold standard" for valid use of statistical tests, this is obviously not enough, and one should fix some specific description language. It looks like a difficult task and there are no serious attempts of this type. Still one could expect that this language should be domain-specific and take into account the relations and constants that are "naturally defined" for the objects in question. This is discussed in details by Gurevich and Passmore [14]. The authors note that questions about statistical tests and their validity do arise in courts when some statistical argument is suggested as evidence, but there are no established procedures to evaluate statistical arguments.[4] One could also mention a similar (and quite important) case: statistical "fingerprints" for falsified elections. There are many examples of this type (see the survey [53] and references within). Let us mention two examples that illustrate the problem of "remarkable post factum observations". Figure 1.1 (provided by Kupriyanov [26]) presents the official results of the "presidential elections"

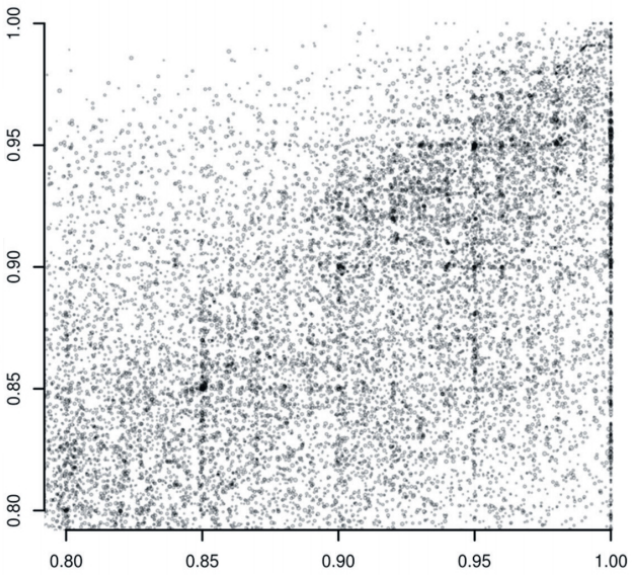


Figure 1.1: "Putin's grid" in 2018 [26]. Note that vertical and horizontal lines are formed by data points, they are not the added grid lines.

in Russia in 2018 and is constructed as follows: for every polling station where both the reported participation rate and the fraction of votes for de facto president of Russia (Putin) exceed 80%, a corresponding grey point is shown. If several points coincide, a darker/bigger point appears. Looking at the "grid lines" formed by these points (for integer percentages; more visible lines appear for the percentages that are multiples of 5), one probably agrees that such a remarkable grid has negligible probability to appear naturally for any kind of elections. However, it is far from obvious which statistical test should be considered here and how can we quantitatively estimate its complexity and its probability.

Related example is provided by "referendum results" in Crimea (Ukraine). As noted by Alexander Kireev [24], the "official results" in Sevastopol (Crimea) include the following data: total numbers of registered voters (306258), total number of ballots (274101) and the number of "yes for the annexation" votes (262041). The two main ratios (the participation rate and "yes" rate) are suspiciously round: 274101/306258 = 0.895000294 and 262041/274101 = 0.95600162. Indeed, in both cases the

[4] In this paper some thought experiments and one real story are considered as examples. One of the thought experiments is as follows: the wife of a president of a state lottery turns out to be its winner. Recently I learned that this example is not so far from the real life as one could think: in 2000 BBC reported that "Zimbabwean President Robert Mugabe has won the top prize [about $2600] in a lottery organised by a partly state-owned bank" (`http://news.bbc.co.uk/2/hi/africa/621895.stm`).

numerator can be obtained by multiplying the denominator by the integer number of promilles and rounding to the closest integer. Probably most statisticians would agree that this coincidence is remarkable and has very small probability, but it is difficult to agree on a specific quantitative estimate of the corresponding probability after a suitable Bonferroni correction.[5]

1.4 Randomness tests in algorithmic information theory

Algorithmic information theory (also called Kolmogorov complexity theory) is outside the scope of this survey[6], but let us mention a few results that have philosophical importance and should be kept in mind when discussing randomness at any level.

Roughly speaking, algorithmic information theory says that *randomness is incompressibility*. More precisely, a binary string looks plausible as an outcome of a fair coin (does not convince us to reject the fair coin hypothesis) if it is incompressible, i.e., if there is no program that produces this string and is much shorter than the string itself. In other words, we

- define Kolmogorov complexity of a string as the minimal length of a program that produces it; in this definition we use some optimal programming language that makes complexity minimal up to an $O(1)$ additive term;
- note that all n-bit strings have complexity at most $n + O(1)$, since a trivial program "print x" has almost the same size as x;
- note that at most 2^{-c} fraction of n-bit strings have complexity less than $n - c$, so this is a very small minority for non-negligible values of c; we treat members of this minority as non-random strings.

This approach is consistent with what we said above about valid tests for randomness as simple sets of small probability. Namely, we consider a test that consists of highly compressible strings, and note that this test is universal in some sense, i.e., it is as sensitive as any other test, up to an $O(1)$-constant.

Technically speaking, there is a result that relates complexity to the randomness deficiency in terms of tests. We state this result for a simple case (uniform distribution on the set of strings of given length; see the textbook [56, Section 14.1] for a more general statement). It uses the notion of *conditional complexity* $C(x|u)$ of a string x given some u (an integer) defined as the minimal length of a program that produces x given u as an input.

Consider some integer function $d(x)$ defined on bit strings. Call it a *deficiency function* if it satisfies two requirements:

- $d(x)$ is *lower semicomputable*, i.e., $d(x)$ can be presented as a limit of a non-decreasing computable sequence of integers (uniformly in x), and
- for every k, the fraction of n-bit strings such that $d(x) > k$, is $O(2^{-k})$.

Such a deficiency function determines, for every n, a series of tests for uniformly distributed n-bit strings, where the kth test set for n-bit strings consists of strings of length n such that $d(x) >$

[5] A rough estimate is attempted in the survey mentioned above [53, p. 49–50]. It takes into account other information about the case. Both anomalies, the integer grid and round percentages, appeared in earlier "elections", so it is not really fair to call them "post factum observations". For the Crimea's "referendum results" the upper bound for the probability after the correction is estimated as 0.1%.

[6] The short introduction can be found in the lecture notes [52] or in the introductory part of the textbook [56]. The algorithmic statistics, the part of algorithmic information theory that deals specifically with the statistical hypotheses and their testing, is discussed in two surveys [60, 61].

k. In terms of the next section, $2^{d(x)}$ is a probability bounded test up to a constant factor. The second requirement guarantees that the test sets have small probability according to the uniform distribution. The first one means, informally speaking, that the test is "semi-effective": if x has some peculiar property that makes it non-random, then we will ultimately discover this property and $d(x)$ will become large, but we never can be sure that x does *not* have properties that make it non-random, since $d(\cdot)$ is not required to be computable.

Proposition 1. *Among the deficiency functions there is a maximal one up to $O(1)$, i.e., a deficiency function d such that for every other deficiency function d' we have $d(x) \geqslant d'(x) - c$ for some c and all x. This maximal function is equal to $n - \mathrm{C}(x|n) + O(1)$ for n-bit strings x.*

This result shows that the difference between length and complexity is the "universal measure of non-randomness" that takes into account all regularities that make a string x non-random. It is easy to prove also that if a string x belongs to a simple small set, then its deficiency is large, thus confirming the informal idea that a small set exhibits non-randomness of its elements.

There are many results about randomness deficiencies in this sense, but we cannot go into the details here and return instead to some other topics that are important for practical randomness tests.

1.5 Families of tests and continuous tests

The law of large numbers says that for independent Bernoulli trials, e.g., for fair coin tossing, the number of successful trials is with high probability close to its expectation, i.e., to $n/2$ for n coin tossings. Therefore, large deviation is a rare event and can be used as a randomness test: as the deviation threshold increases, the probability of the event "the deviation exceeds this threshold" decreases, usually rather fast.

Instead of fixing some significance level and the corresponding threshold one could consider a family of tests: for every significance level ε we consider a set T_ε of measure at most ε. It consists of the outcomes where deviation exceeds some threshold that depends on ε. As ε decreases, the threshold increases, and the set T_ε and its measure decrease.

Such a family of tests can be combined into one non-negative function $t(x)$ defined on the set of possible outcomes, if we agree that T_ε is the set of outcomes where $t(x) \geqslant 1/\varepsilon$. Here we use $c = 1/\varepsilon$ as the threshold instead of ε to simplify the comparison with expectation-bounded tests discussed below. In this language the bound for the probability of T_ε can be reformulated as

$$\Pr[t(x) \geqslant c] \leqslant 1/c \quad \text{for every } c > 0 \qquad (*)$$

Informally speaking, $t(x)$ measures the "rarity", or "randomness deficiency" of an outcome x: the greater $t(x)$ is, the less plausible is x as an outcome of a random experiment.

Functions t that satisfy the condition $(*)$ are called *probability-bounded randomness tests* [4]. Sometimes it is convenient to use the logarithmic scale and replace t by $\log t$. Then the condition $(*)$ should be replaced by the inequality $\Pr[t(x) \geqslant d] \leqslant 2^{-d}$. Note that a similar condition was used for deficiency functions in Section 1.4.

The condition $(*)$ is a consequence of a stronger condition $\int t(x)\, dP(x) \leqslant 1$ where P is the probability distribution on the space of outcomes. In other terms, this stronger requirement means that the expected value of t (over the distribution P) is at most 1, and the condition $(*)$ is its consequence, guaranteed by Markov's inequality. The functions t that satisfy this stronger condition are called *expectation-bounded randomness tests* [4].

In fact these two notions of test are rather close to each other: if t is a probability-bounded test, then $t/\log^2 t$ is an expectation-bounded test up to $O(1)$-factor. Moreover, the following general result is true:

Proposition 2. *For every monotone continuous function* $u : [1, +\infty] \to [0, \infty]$ *such that* $\int_1^\infty u(z)/z^2\, dz \leqslant 1$ *and for every probability-bounded test* $t(\cdot)$ *the composition* $u(t(\cdot))$ *is an expectation-bounded test.*

The statement above [10] is obtained by applying Proposition 2 to $u(z) = z/\log^2 z$.

Chapter 2

Randomness tests

2.1 Where do we get randomness tests?

As we have mentioned, different classical results of probability theory can be used as randomness tests. Take for example the law of large numbers. It says that some event, namely, a large deviation from the expected value, has small probability. This event can be considered as a test set. Mathematical statistics provides a whole bunch of tests of this type for different distributions, including χ^2-test, Kolmogorov–Smirnov test, and others.

Another source of statistical tests, though less used in practice, is provided by the probabilistic existence proofs. Sometimes we can prove that there exists an object with a given combinatorial property, say, a graph with good expansion properties, and the proof goes as follows. We consider a probabilistic process that constructs a random object. In our example this object is a graph. We prove that with high probability this random object satisfies the combinatorial property. Now we use the bit source that we want to test as a source of random bits for the algorithm. If we find out that the object constructed by the algorithm does *not* have the combinatorial property in question, we conclude that a rare event happened, and our source of random bits failed the test.

One should also mention tests inspired by algorithmic information theory (see, e.g., a 1992 paper by Maurer [34]) Each file compressor (like `zip`, `bzip`, etc.) can be considered as a random test. Assume that we have a sequence of bits, considered as a file, i.e., a sequence of bytes. If this file can be compressed by n bytes for some non-negligible n, say, by a dozen of bytes, then this bit sequence fails the test and can be considered as non-random one. Indeed, the probability of this event is at most 256^{-n}, up to a factor close to 1. The Bonferroni correction here says that we should multiply this probability by the number of popular compressors, but even if we assume that there are thousands of them in use, it usually still keeps the probability astronomically small.

There is also a general way to construct probability-bounded tests. It is called "p-values", and the two previous examples of randomness tests can be considered as its special cases. Consider an arbitrary real-valued[1] function D defined on the space of outcomes. The value $D(x)$ is treated as some kind of "deviation" from what we expect, so we use the letter D. Then consider the function

$$p_D(x) = \Pr[\{y : D(y) \geqslant D(x)\}].$$

(defined on the same set of outcomes). In other words, for every threshold d we consider the set

$$T_d = \{y : D(y) \geqslant d\}$$

of all outcomes where the deviation is at least d, and measure the probability of this event, thus "recalibrating" the deviation function. In this language, $p_D(x)$ is the probability of the event $T_{D(x)}$,

[1] This trick in a more general situation where the values of D are elements of some linearly ordered set, is considered in a paper by Gurevich and Vovk [15].

the chance to have in a random experiment of the given type the same deviation as it happened now, or a larger one.

Proposition 3.

(a) *For every $c \geqslant 0$, the probability of the event $p_D(x) \leqslant c$ is at most c.*

(b) *If each value of function D has probability at most ε, then the probability of the event $p_D(x) \leqslant c$ is between $c - \varepsilon$ and c.*

Proof. The probability $p_d = \Pr[T_d]$ decreases (more precisely, does not increase) as d increases. The function $d \mapsto p_d$ is left-continuous since the inequalities $D \geqslant d'$ for all $d' < d$ imply $D \geqslant d$. However, it may not be right-continuous, and a similar argument shows that the gap between the value of p_d and the right limit $\lim_{d' \to d+0} p_{d'}$ is the probability of the event $\{x : D(x) = d\}$. Now we add to this picture some threshold c, see Figure 2.1. It may happen (case 1) that c is among the values

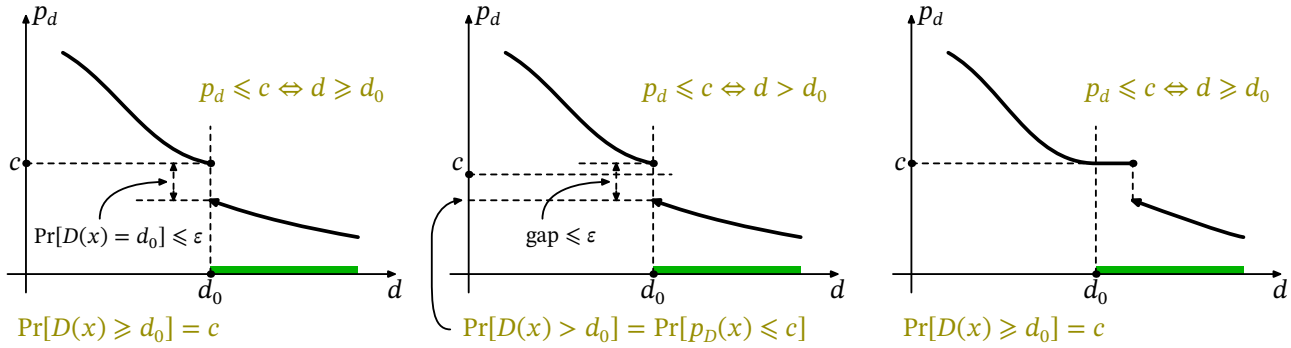


Figure 2.1: Proof of Proposition 3

p_d. This case is shown on the left and right pictures (Figure 2.1). On the right picture the function p_d is constant on an interval where there are no values of D or these values have probability 0. Case 2: the threshold c may fall in a gap between some value of p_d and the right limit in the same point (denoted by d_0), as shown in the middle picture. The size of the gap is the probability of the event $D(x) = d_0$ and is at most ε according to our assumption. For the left and right pictures the inequality $p_D(x) \leqslant c$ means that $D(x) \geqslant d_0$, and the probability of the event $p_D(x) \leqslant c$ is exactly $p_{d_0} = c$, so both statements (a) and (b) are true. In this case the value of ε does not matter. For the middle picture $p_D(x) \leqslant c$ when $d > d_0$, and the probability of this event is not the value of p_d when $d = d_0$ but the right limit of p_d as $d \to d_0 + 0$. Still the difference does not exceed ε according to the assumption in (b), and again both statements are true. □

Remark 1. This proof is given for the general case (X may be infinite); in the finite case the function p_d has only finitely many values, and the graph is a finite family of horizonal lines.

Remark 2. Proposition 3 obviously implies that the function $1/p_D(x)$ is a probability-bounded test. This observation allows us to construct many probability-bounded tests, starting from almost any random variable D. For example, we get a test from a probabilistic existence proof if we let D be the function that appears in the combinatorial statement, e.g., the second eigenvalue for the probabilistic proof that expander graphs exists. The only caveat is that we need to compute the function $d \mapsto p_d$, and this is usually not so easy. This function is often replaced by some its approximation, and this may lead to problems; see the discussion below.

Remark 3. If we apply this procedure to a function D that is already a probability-bounded test, then by definition we get some new test $t = 1/p_D$ such that $t(x) \geqslant D(x)$ for all x. In general, the function t could exceed D if the inequality in the condition $(*)$, Section 1.5, is strict.

2.2 Secondary tests

There is an important type of randomness tests that can be called "secondary tests". Tests of this type were already mentioned in the classical book of Knuth [25, Section 3.3.1, B] and are extensively used in practical test suites [29]. Recall Proposition 3 and assume for a while that every individual value of D has very small probability, so we may assume that there are (almost) no gaps in the graph of p_d. Then Proposition 3 says that the random variable p_D is uniformly distributed on $[0, 1]$. Repeat the test N times, using fresh random bits (from the generator we are testing) for each repetition. Assuming that the null hypothesis is true, we get N independent reals that are uniformly distributed in $[0, 1]$. Then we may apply any test for the independent uniformly distributed variables, e.g., the Kolmogorov–Smirnov test for this distribution.

This procedude converts any p-value test for a random bits generator that has negligible probabilities of individual values into a "secondary test" that could be much more sensitive. Knuth describes a similar trick, but he does not use the recalibration using p-values and applies the Kolmogorov–Smirnov (KS) test directly to the values of D and the distribution that should appear if the null hypothesis is true:[2]

> ...We should observe that the KS test may be used in conjunction with the χ^2 test... Suppose we have made, say, 10 independent χ^2 tests of different parts of a random sequence, so that values $V_1, V_2, \ldots, V_{10}$ have been obtained. It is not a good policy simply to count how many of the V's are suspiciously large or small. This procedure will work in extreme cases, and very large or very small values may mean that the sequence has too much local nonrandomness; but a better general method would be to plot the empirical distribution of these 10 values and to compare it to the correct distribution... This would give a clearer picture of the results of the χ^2 tests, and in fact the statistics K_{10}^+ and K_{10}^- [from KS test] could be determined as an indication of the success or failure... [Speaking about an example discussed earlier:] Notice that *all 20 observations in Fig. 4 (c)* [a figure from Knuth's book that we do not reproduce] *fall between the 5 and 95 percent levels*, so we would not have regarded *any* of them as suspicious, individually; yet collectively the empirical distribution shows that these observations are not at all right [25, Section 3.3.1, p. 50–51].

We return to the use of secondary tests in practical test suites in the next section.

2.3 Testing (pseudo)randomness in practice

There are several suits of randomness tests that are often used. The early history of randomness tests (as well as pseudorandom number generators) is described by Knuth [25, Section 3.3]. He starts with χ^2 and Kolmogorov–Smirnov tests, explains secondary testing (see the quote in the previous section) and also describes several *ad hoc* tests.

2.3.1 Diehard

Later George Marsaglia developed a `diehard` series of tests that were included (as C and Fortran sources) in a CD that he prepared [29]. That CD also included a collection of files with "random" bits, constructed by combining the output of hardware random bits generators with some deterministic pseudorandom sequences, see below Section 4.1. The description of the tests could be found in Marsaglia's papers [28, 31]; see also the file `tests.txt` in the source code of the tests [29].

[2] This is an equivalent approach since KS-test gives the same result after any monotone recalibration of the empirical values and theoretical distribution.

However, there are some problems with these tests. They heavily use the secondary test approach but not always in a correct way. First, one of the tests computes p-values for data that are not independent, as the following description, copied verbatim from the source code, shows:

```
           This is the BIRTHDAY SPACINGS TEST
Choose m birthdays in a year of n days.  List the spacings
between the birthdays.  If j is the number of values that
occur more than once in that list, then j is asymptotically
Poisson distributed with mean m^3/(4n).  Experience shows n
must be quite large, say n>=2^18, for comparing the results
to the Poisson distribution with that mean.  This test uses
n=2^24 and m=2^9,  so that the underlying distribution for j
is taken to be Poisson with lambda=2^27/(2^26)=2.  A sample
of 500 j's is taken, and a chi-square goodness of fit test
provides a p value.  The first test uses bits 1-24 (counting
from the left) from integers in the specified file.
      Then the file is closed and reopened. Next, bits 2-25 are
used to provide birthdays, then 3-26 and so on to bits 9-32.
Each set of bits provides a p-value, and the nine p-values
provide a sample for a KSTEST.
```

As we see from this description, the different p-values use overlapping bits (2–25, 3–26, etc.) of the same numbers. There is no reason to expect that they are independent, contrary to the requirements of the Kolmogorov–Smirnov test. This description also exhibits another problem that appears in many tests from `diehard` suite. We use some asymptotic approximation, in this case the Poisson distribution, instead of the true distribution, ignoring the approximation error for which we have no upper bounds. Moreover, even if the error can be upper-bounded for the primary test, this upper bound does not translate easily into a bound for an error in the secondary test where we use the approximate distribution for recalibrating the deviations. The same problem appears in rank 6×8 test (Kolmogorov–Smirnov test is applied to dependent variables).

Sometimes even the parameters of approximate distribution are only guessed. For example, in the description of one of the tests (named OQSO) Marsaglia writes about the distribution: "The mean is based on theory; sigma comes from extensive simulation". For the other one (called "parking lot test") even the mean is based on simulation: "Simulation shows that k should average 3523 with sigma 21.9 and is very close to normally distributed. Thus $(k - 3523)/21.9$ should be a standard normal variable, which, converted to a uniform variable, provides input to a KSTEST based on a sample of 10". Here KSTEST is the Kolmogorov–Smirnov test for uniform distribution. The arising problem is described by Marsaglia as follows:

> NOTE: Most of the tests in DIEHARD return a p-value, which should be uniform on $[0, 1)$ if the input file contains truly independent random bits. Those p-values are obtained by $p = F(X)$, where F is the assumed distribution of the sample random variable X — often normal. But that assumed F is just an asymptotic approximation, for which the fit will be worst in the tails. Thus you should not be surprised with occasional p-values near 0 or 1, such as .0012 or .9983. When a bit stream really FAILS BIG, you will get p's of 0 or 1 to six or more places. By all means, do not, as a Statistician might, think that a $p < .025$ or $p > .975$ means that the RNG has "failed the test at the .05 level". Such p's happen among the hundreds that DIEHARD produces, even with good RNG's. So keep in mind that "p happens".

This note combines two warnings. One is quite general and is related to the question of many tests applied to one sequence, see the discussion of the Bonferroni correction above, Section 1.2. The

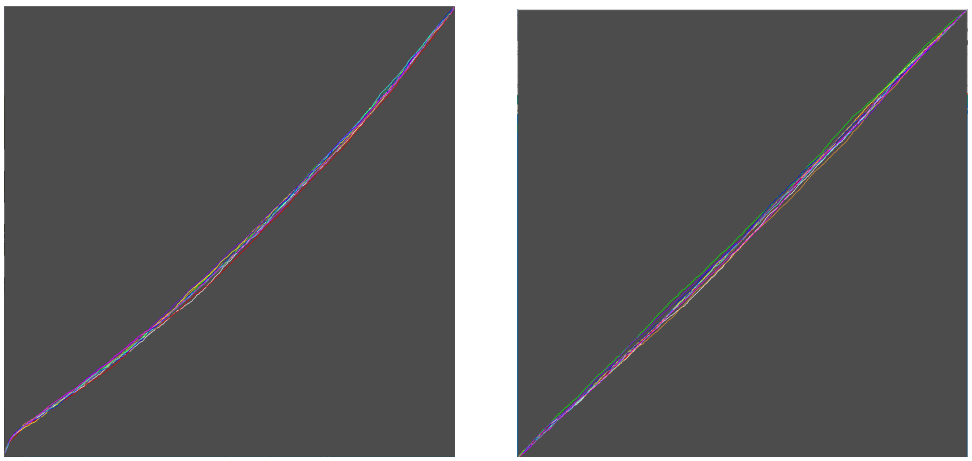


Figure 2.2: Many tests are based on approximation. The graph (a) shows the distributions of the output of Knuth's run test [25, Section 3.3.2] for several runs on data from `bittbabbler` generator with sample size 1000. They all seem to deviate from the uniform distribution on $[0, 1]$ significantly (and in a similar way). Is this because of a bad generator or a bad approximation? Probably the latter, since for samples of size 10000 (as recommended by Marsaglia [29]) the curve is much closer to the diagonal showing the theoretical uniform distribution on $[0, 1]$.

other one that should be separated from the first (but is not) is that `diehard` tests are not really tests in statistical sense, since they use the approximate distribution for recalibration (cf. Figure 2.2) and therefore small p-values could appear more often than they should.

Some other tests (not included in `diehard`) were later suggested by Marsaglia and Tsang [30] (as well as other authors, see below).

2.3.2 Dieharder

A decade later Robert Brown [9] produced an extended version of the `diehard` test suite, called `dieharder`. The code was rewritten and published under GNU public license, integrated with GNU statistical library and parametrized, so now one can vary the sample size and the number of p-values easily. New tests were added and other improvements made. The resulting package is supported by mainstream Linux distributions. The package involves an extensive documentation. In particular, the `man` page says:

> A failure of the distribution of p-values at any level of aggregation signals trouble. ⟨...⟩ The question is, trouble with what? Random number tests are themselves complex computational objects, and there is a probability that their code is incorrectly framed or that roundoff or other numerical — not methodical — errors are contributing to a distortion of the distribution of some of the p-values obtained.

In this quote two problems are noted: the coding errors in the tests, and the problems related to the mathematical flaws in the approximate data used to construct the tests. The suggested solution for both problems is the same: testing these tests on "reference" random number generators. The `man` page says:

> There are a number of generators that we have theoretical reasons to expect to be extraordinarily good and to lack correlations out to some known underlying dimensionality, and that also test out extremely well quite consistently. By using several such generators and not just one, one can hope that those generators have (at the very least) different correlations and should not all uniformly fail a test in the same way and with

> the same number of p-values. When all of these generators consistently fail a test at a given level, I tend to suspect that the problem is in the test code, not the generators, although it is very difficult to be certain...
>
> Tests (such as the diehard `operm5` and `sums` test) that consistently fail at these high resolutions are flagged as being "suspect" ⟨...⟩ and they are strongly deprecated! Their results should not be used to test random number generators pending agreement in the statistics and random number community that those tests are in fact valid and correct so that observed failures can indeed safely be attributed to a failure of the intended null hypothesis.

Unfortunately, `dieharder-3.31.1`, the last version available (as of March 2021), also has some problems (see Section 7.1 for details). One of them, affecting almost all tests, is the incorrect code that computes the Kolmogorov – Smirnov statistic. This code produces incorrect values, sometimes even impossibly small values, and in this case the computation of p-value gives 1. Indeed, in this case with probability 1 the deviation will be bigger than this impossibly small value. This is (correctly) interpreted as the test failure. Fortunately, it seems that for larger sample sizes the error in the statistics computation becomes less important.

2.3.3 NIST test suite

In 2000 the National Institute of Standards published a description of a test suite for randomness, including the source code. Now there exists an updated version [37].[3]

The description starts with some general words about randomness: "For example, a physical source such as electronic noise may contain a superposition of regular structures, such as waves or other periodic phenomena, which may appear to be random, yet are determined to be non-random using statistical tests" (p. 1-2). Then the authors speak about two types of possible errors: Type I (rejecting a good generator) and Type II (accepting a bad one) and about probabilities of these errors. However, their comments are misleading. Authors explain that a Type I error probability is a probability for a random sequence to get into the rejection set under a null hypothesis H_0 — and this is correct. But then they say something confusing about the Type II errors: "Type II error probability is ⟨...⟩ P(accept $H_0|H_0$ is false)" [37, p.1-4]. While H_0 is a statistical hypothesis (model), namely, the assumption that the bits are independent and uniformly distributed, the words "H_0 is false" do not define any distribution, so one cannot speak about this conditional probability. The authors acknowledge this by saying "The probability of a Type II error is denoted as β. ⟨...⟩ Unlike α [the probability of a Type I error], β is not a fixed value. ⟨...⟩ The calculation of Type II error β is more difficult than the calculation of α because of the many possible types of non-randomness" — but still one could conclude from what the authors say that Type II error probability is well defined and is some number, though difficult to compute. This is a gross misunderstanding.

Then the authors explain the meaning of p-values, but again their explanations sound confusing, to say the least: "If a *P-value* for a test is determined to be equal to 1, then the sequence appears to have perfect randomness" (page 1-4). In reality the value 1 is not much better than the value 0, since the correctly computed p-values have uniform distribution. Even more strange is the following remark: "For a *P-value* $\geqslant 0.001$, a sequence would be considered to be random with a confidence of 99.9%. For a *P-value* < 0.001, a sequence would be considered to be non-random with a confidence of 99.9% [37, p. 1-4, line 6 from below]. The second part could be interpreted in a reasonable way, though one should be cautious here, especially in the case of many tests. But the first part is completely misleading. Of course, one test that did not fail convincingly does not mean that the sequence is random with high confidence!

[3] The original version contained 16 randomness tests. In 2004 some errors in two tests were pointed out [23]. Correcting these errors, the revised version (2010) deleted one of the tests (the Lempel – Ziv test) and corrected the other one (the Fourier spectral test).

General remarks about tests constitute Part I of [37]. Parts II and III consist of the description and commentary for 15 tests. Some are similar to the tests in `diehard` while some other are different. The final Part IV, "Testing strategy and the Result Interpretation", recommends two ways to analyze the results of the tests. When several runs of a test produce a sequence of p-values, two forms of analysis of this sequence are recommended: "Proportion of Sequences Passing a Test" (4.2.1) and "Uniform Distribution of *P-values*" (4.2.2). Both are some variants of secondary tests: assuming that the distribution of p-values is uniform in $[0, 1]$, authors recommend to look at the proportion of values exceeding some threshold and to compare it with the Bernoulli distribution (4.2.1), or divide the interval $[0, 1]$ into some number of bins and apply χ^2-test (4.2.2). This approach replaces the Kolmogorov–Smirnov test used by Marsaglia in `diehard`, and in `dieharder`.

As for the case of `dieharder`, many tests from the NIST collection use approximations for computing p-values. The only warning about the consequences of this approach appears in the last section (p. 4-3):

> In practice, many reasons can be given to explain why a data set has failed a statistical test. The following is a list of possible explanations. The list was compiled based upon NIST statistical testing efforts.
>
> (a) An incorrectly programmed statistical test. ⟨...⟩
>
> (b) An underdeveloped (immature) statistical test.
>
> There are occasions when either probability or complexity theory isn't sufficiently developed or understood to facilitate a rigorous analysis of a statistical test. Over time, statistical tests are revamped in light of new results. Since many statistical tests are based upon asymptotic approximations, careful work needs to be done to determine how good an approximation is.
>
> (c) An improper implementation of a random number generator. ⟨...⟩
>
> (d) Improperly written codes to harness test input data. ⟨...⟩
>
> (e) Poor mathematical routines for computing *P-values*. ⟨...⟩
>
> (f) Incorrect choices for input parameters.

It is hardly surprising that with such a relaxed approach to statistical testing ("over time, statistical tests are revamped") the authors have included two bad tests in the original version of the document, see the paper [23] where the errors are noted. It is instructive to look at the errors in these tests. The first error, in the Fourier spectral test, happened because the expectation and variance of the approximating normal distribution were computed incorrectly. The second is more interesting, since two different (and quite predictable) problems with the p-values approach appeared at the same time. First, the distribution of the test statistics based on the Lempel–Ziv compression algorithm was not computed exactly but was approximated using some presumably good pseudo-random number generator as reference. The experiments with other generators made in the paper [23] showed that this approximation is dubious. The other problem could be related to the non-negligible probability of individual values. As we have mentioned, in this case the distribution of the p-values differs from the uniform one. The results of numerical experiments described in the paper suggest that this could be the reason for the rejection of truly random sequences. In the current version of the NIST document[4] this test is excluded (it contains 15 tests instead of 16).

[4]Unfortunately, the current version of the NIST report [37] was not checked carefully either. For example, the description of the serial test (section 2.11.4, (5)) contains conflicting instructions for computing p-values: the example includes division by 2 that is missing in the general formula. The C code follows the example, not the general formula. Also the values of `igamc` function in this section are incorrect, while the correct values do appear few lines later, in Section 2.11.6.

Chapter 3

Robust randomness tests

3.1 Comparison with a trusted randomness source

There are different ways to deal with errors in statistical tests. Finding and correcting the coding errors is a general problem for all software, and tests are no exceptions here. One may argue that, since errors are anyway possible for many reasons (see the list above), we should not insist on the mathematical correctness of tests, just deleting the tests when they are discovered to be incorrect. Still the other approach is to do whatever we can to avoid errors that could be avoided. In this section we explain, following the technical report [54], how one could avoid problems related (a) to the approximation errors while computing p-values, and (b) to the non-negligible probabilities of individual outcomes. This will be done in two steps.

Let us assume for now (in this section) that a reference generator that is truly random is available. But instead of using the reference generator to find the approximate distributions or to look for suspicious tests, as suggested by the authors of `dieharder` and NIST tests, we use it directly. Recall that the we used the Kolmogorov–Smirnov test to check that the distribution of p-values is consistent with the uniform distribution, and our problem was that due to approximation errors and non-zero probabilities of individual deviation values the distribution of p-values is not exactly uniform under the null hypothesis. However, there is a version of the Kolmogorov–Smirnov test that deals with *two* samples. Here the null hypothesis is that the two samples are formed by independent random variables with the same distribution, but nothing is assumed about this distribution.

Therefore, we may proceed as follows. The first sample of p-values is constructed using the random numbers generator that we test, as before. The second sample is constructed exactly in the same way but using the reference generator. Then we apply the Kolmogorov–Smirnov test for two samples. This procedure remains valid even if the formulas used to convert the deviations into p-values are only approximately true, or even completely wrong, since even completely wrong formulas would be the same for both generators, the one we test and the reference one. So we can omit the recalibration step completely and just consider the samples of deviations.

Remark 4. In fact, only the ordering is important, so the Kolmogorov–Smirnov test for two samples can be presented as follows. We have two arrays of reals, $x_1, \ldots, x_n$ (one sample) and $y_1, \ldots, y_m$ (another sample). Then we combine them into one array of length $n + m$ and sort this array, keeping track of the origin: elements that came from the first and second samples are marked by letters X and Y respectively. In this way we get a sequence of $n + m$ letters X and Y that contains n letters X and m letters Y, and consider some test statistic for the following null hypothesis: *all* $\binom{n+m}{m}$ *sequences of this type are equiprobable.* The Kolmogorov–Smirnov test uses some specific statistic, namely, the maximal difference between the frequencies of X's and Y's in all prefixes, but the same approach can be used with any other test for this distribution.

The procedures to compute Kolmogorov–Smirnov p-values for two samples are included in the R source code [47]. There is a simple combinatorial algorithm (the follows the definition directly)

that works quite well; it would be exact if not the rounding errors for floating-point reals. Also the R source includes some asymptotic approximation, but no error bounds for it, so it should not be trusted too much. For more certainty, we rewrote rewrite the exact algorithm using integers of arbitrary lengths (it is still reasonably fast for samples that contain several thousands values), though floating point version is much faster. We used it for preliminary estimates (and check the findings with exact algorithm). See Section 7.2 for details.

Remark 5. In this way we get a test that does not depend on the approximations to the distributions that we do not know how to compute. However, there is some price for it. Since now the reference generator is an additional source of random variations, we need more samples to get the same sensitivity of the test. This increase, however, is rather modest.

3.2 Testing without a trusted randomness source

We constructed a randomness test that does not rely on unproven assumptions about distributions that we cannot compute exactly. However, it uses a reference generator that is assumed to be truly random, and this is crucial. Obviously, if we use a faulty reference generator to test a truly random one, the test will fail (the procedure is symmetric, so we get exactly the same result when testing a faulty random generator against a truly random reference). In some sense, we constructed only a "randomized test of randomness". This is unsatisfactory, but can be easily avoided using the following trick.

Let us consider n deviation values $d_1, \dots, d_n$ obtained by using bits from the generator we are testing. Then construct the other sample, $d'_1, \dots, d'_m$, but this time let us use not the reference generator but the bitwise `xor` of the bits from the reference generator and fresh bits from the generator we are testing. Then we apply the Kolmogorov–Smirnov test to these two samples. If

1. the generator that we are testing is truly random, and
2. the reference generator and the test generator are independent,

then the probability to fail the test is guaranteed to be small due to Kolmogorov–Smirnov's result.

Remark 6. Of course, if the reference generator is not independent with the one that we want to test, the correctness claim is no more true. For example, if during the second part the reference generator produces the same bits as the generator we are testing, the `xor` bits will be all zeros. However, the independence condition looks much easier to achieve. For example, the reference bits could be produced in different place, or in advance, or can even be an output of a fixed deterministic pseudorandom generator.

Remark 7. If the reference generator produces truly random bits that are independent from the output of the generator we are testing, then `xor`-bits are also truly random. So our new test is as sensitive as the previous one (the comparison with the reference generator) if the reference generator is truly random, but the correctness of the new test does not depend on the assumption of true randomness for the reference generator.

Remark 8. There is one small problem that we have not mentioned yet: while sorting the array of deviations, we may have ties. If some value from the first sample coincides with some value from the second sample, then the letter (X or Y in our notation, see above) is not well defined. However, we can break the ties randomly, using the bits of the generator we are testing. In this way we may assume that these bits are truly random when bounding the probability of Type I error.

Remark 9. The same idea can be used even for "informal" tests. For example, imagine that we construct some image based on the bits we test, and then people look at this image and decide whether it looks similar to the pictures of the same type that use reference generator or there are

some visible differences[1]. Recalling the interactive non-isomorphism proof and using the same trick as before, we can make a robust test. Take $2n$ disjoint bit blocks from the generator under testing. Use n of them to create images, and do the same for other n blocks but use the bitwise `xor` of these blocks and n blocks of the same size from some other origin. As a source of these auxiliary blocks one may use a reference generator, or the binary representation of π, or any other source. The only requirement is that the auxiliary blocks should be fixed before sampling our generator. If a human expert (or a machine-learning algorithm), looking at the resulting $2n$ images, can correctly classify them into two groups according to their origin, then the generator fails the test, and the probability of this for a true random bits generator is 2^{-2n}, up to a poly(n)-factor.

One can also consider a more advanced version of this test. Several experts say how "random" the images are. Then the images are ordered according to their approval ratings and Kolmogorov–Smirnov test for two samples is used.

In the following sections we show two specific examples of this approach used in our experiments.

3.3 An example of a robust estimate based on χ^2-tests

We analyzed the random bits produced by the quantum random bits generator produced by IDQuantique, using the observations made in [19] (see also [20]. These observation are discussed in Section 6.6. For now it is enough to know that there is an utility called `ent` that can be applied to any file and produces some floating point number, called χ^2*-value*. According to the man page (Ubuntu 20.04) for this utility, "the chi-square distribution is calculated for the stream of bytes in the file"; we have not looked at the source code to check this since for us it is enough to know that this number is some (deterministic) function of the file contents.

We get 40 files from Quantis device (without postprocessing), each containing 200 megabytes. Then we applied `ent` tool to the first 20 of them, getting 20 χ^2-values (or whatever the `ent` tool computes). For the other 20 files, we performed `xor`-operation with some fixed 200 megabytes file. This file was obtained from BitBabbler generator described in Section 6.8, but it does not matter for the validity of the test.

Assuming that the null hypothesis is true, i.e., Quantis device produces independent random bits, we conclude that all these 40 files are obtained as independents samples from the same distribution (uniform distribution on the bit sequences of size 200 megabytes). Therefore, the `ent` values of these files are 40 independent samples from some distribution. But the experimental data clearly contradict this assumption: for the first 20 files the results were

> 615.07, 555.99, 624.12, 611.18, 575.19, 559.00, 557.75, 674.92, 670.97, 624.57, 622.26, 696.87, 619.98, 606.93, 590.63, 527.77, 692.10, 586.60, 635.59, 720.00,

while for the second group of 20 files (`xor`-files) the results were

> 249.60, 236.93, 241.85, 279.96, 235.00, 240.56, 246.62, 268.19, 243.42, 238.83, 281.28, 239.48, 265.41, 247.28, 269.42, 252.13, 262.96, 263.75, 243.03, 247.77.

Note that all the values from the second group are smaller that all values of the first group. This event has probability $1/\binom{40}{20} < 10^{-11}$ under our null hypothesis. Note that this test was chosen before the experiments, and no other runs with less significant results were observed. This p-value is robust; it does not depend on any approximations used in the derivation of χ^2-test. The results cannot be affected by (theoretically possible) bugs in `ent` utility, though, of course, we still need to be sure that

[1] This is not a purely theoretic possibility: the documentation for some hardware random bit generators contains pictures of this type.

the files were really obtained from the Quantis device, correctly `xor`-ed with the same file, and the same `ent` utility was applied to all of the 40 files.

Note also a significant gap between the values in both groups. This difference *per se* does not guarantee anything since we want to avoid any assumptions about the behavior of the `ent` tool, but still is remarkable.

In fact, the deficiencies of the Quantis device are visible on much smaller (though still large) samples than 8 Gbytes (used in this example). One of the robust test described in Section 3.3, namely the `sts_serial` test, provides (among about 60 p-values) values smaller that 10^{-8}, so even after Bonferroni correction the p-value is less than 10^{-6}. (This happened for a dozen for 1 Gb files in our experiments, so one can combine these p-values to get an astronomically small one.) See also [19, 20] and the discussion of their results in Section 6.6.

3.4 Spectral tests

The approach described above can use any function. So one can just write an arbitrary program (or use arbitrary utility instead of `ent`) and get reliable test results (a small p-value will indicate problems with the generator, and this conclusion does not rely on any assumption about the function.

Still the sensitivity of the tests cannot be guaranteed in any sense. The extreme case: if the function is constant, that all the p-values (returned by 2-sample Kolmogorov–Smirnov test as described) equal 1. So we need to look for a "non-trivial" functions that have a good chance to make a good test. A natural source of candidates is provided by probabilistic arguments.

There are many cases where the existence of a combinatorial structure with required properties

Besides other applications, random generators are used to produce "random" combinatorial structures (matrices, graphs, and so on). For example, most random graphs of small degree enjoys the properties of *expanders* [18], which are used in various applications (it is enough to mention that expander are used to construct error correcting codes with good properties). So it is interesting to verify whether pseudo-random generators passing standard statistical tests can produce pseudo-random generators that are in some natural sense similar to a "typical" random graph.

On of the approaches to the definition of expanders uses spectral properties of graphs: an expander is a graph where all eigenvalues except for one are relatively small. We tested this property for graphs produced by pseudo-random generators mentioned in the previous section. Notice that such an experiment can be interpreted as one more non-conventional test of randomness.

Observed results : The pseudo-random generators that pass the standard test of Dieharder produce graphs with spectral properties that look indistinguishable with the graphs obtained from good physical generators of random bits. With the generators that fail standard test of Dieharder, we obtained pseudo-random graphs with visibly unusual spectral properties, so that we can reliably distinguish these graphs from ones obtained with good physical random generators. However, they do not fail the "spectral" tests in the same way.

Generator 3 described in Section 6.11 fails the tests of DieHarder, but it provides graphs with pretty good spectral properties (similar to those of the "truly" random graphs). On the other hand, Generator 4 fails the spectral test dramatically (we obtain graphs with spectra that are significantly bigger than what we have with truly random graphs).

Even if this difference is large enough to conclude that these generators are not perfect, such an imperfection of the spectral radius can be arguably ignores in some applications where we need random graphs with "expander" properties. We believe that this is an interesting avenue for the future research.

Interpretation: The results of our experiments suggest that graphs with good enough "expander properties" can be obtained with a rather simple pseudo-random generators, with cheap computations

and very short random seed. Possibility of usage of such expanders in applications (e.g., in constructions of error correcting codes) deserves a more thorough research. In particular, it is interesting to investigate spectral properties of pseudo-random uniform and bipartite graphs with fixed degrees of vertices, and the maximal sizes of pseudo-random graphs (with respect to the seed provided to a generator) that allows to maintain necessary spectral properties.

3.5 Kolmogorov – Smirnov and other robust tests

One could use Kolmogorov – Smirnov test directly, without using any other test. Let us consider the bit stream we are testing as a sequence $x_0, x_1, x_2, ...$ of 16-bit integers (or bytes, or blocks of some other fixed size). Consider an arbitrary property of an integer (a block). Then we use this property as a selection rule: we split our sequence into two subsequences. If x_{2i} satisfies the property, we put x_{2i+1} in one subsequence, if not, we put to the other one. For a truly random sequence of bits we get two random subsequences with the same distribution, so we can apply Kolmogorov – Smirnov test to them. In this way we get a robust test that does not need to use any reference generator.

A similar approach may be used to test independence of two generators (say, left and right channel in stereo-noise, or several hardware generators in the same device of in different devices). For that we use the data from one generator to split the data from the other generator into two subsequences. However, in this way the null hypothesis involves the independence between two generators and also between different samples from the same generator. Therefore a small p-value can appear also for two independent generators if different samples from one of the generators are dependent (the extreme case is when both sequences are chosen randomly and independently between $0, 1, 0, 1, ...$ and $1, 0, 1, 0, ...$).

There are also other theoretical approaches that could give valid p-values. First, the compression algorithms provide such a test: if n-bit file is compressed by more than c bits, the probability of this event (for a random file) is at most 2^{-c}. Second, one can consider martingales, i.e., functions $x \mapsto m(x)$ defined on binary strings with non-negative values such that $m(\Lambda) = 0$ for empty string Λ, and

$$m(x) = \frac{m(x0) + m(x1)}{2}.$$

The martingale inequality guarantees that the event "m is at least c on some prefix of a infinite sequence generated by a fair coin" has probability at most $1/c$. See [56] for more details.

Chapter 4

Hardware randomness generators

4.1 How it started

Randomness is ubiquitous — from the coin tossing and cosmic rays to the thermal noise in audio and video recordings, Brownian motion, quantum measurements and radioactive decay. So one may think that constructing a good randomness generator is an easy task. However, if we require that the output distribution is guaranteed with high precision, the problem becomes much more difficult. Coins may be biased, the independence between the two consecutive coin tosses may be not absolute, the circuit with the noise is affected also by some undesirable signals that may not be random, etc. In addition, some technical errors could happen.

For an illustration one may look at the sequences of bits provided by Marsaglia [29]. Among them there are two bit sequences that he got from two devices he bought (one from Germany, one from Canada) and a third bit sequence produced (as Marsaglia says) by some hardware random generator in California. The names are `canada.bit`, `germany.bit` and `calif.bit`. The device makers claimed that the bits produced by their devices are perfectly random. However, applying `diehard` tests to these sequences, Marsaglia found that they are far from being random. In fact, looking at two of them (Canada and Germany), one could guess one of the reasons for their non-randomness [11]. Namely, splitting the bit sequences into bytes (integers in $0 \ldots 255$ range) and searching for the two-byte substring 10 10, we find that this substring does not appear at all (at least among the first 10^6 bytes I tested), while the expected number of occurrences is several dozens. To get an idea why this happens, one may also count the substrings of the form 13 x and find out that one of them appears much more often than the others: the substring 13 10 occurs more than four thousand times (instead of expected few dozens).

If the reader worked a lot with Unix and MSDOS computers in 1990s, she would immediately see a plausible explanation: the file was converted as a text file from Unix to MSDOS encoding. In Unix the lines of a text file were separated just by byte 10, while in MSDOS they were separated by 13 10. Converting 10 to 13 10, we make substrings 10 10 impossible and drastically increase the number of substrings 13 10.

The third file `calif.bit` probably had some other history that did not involve Unix to MSDOS conversion, but still fails the tests for the other reasons.[1]

Marsaglia solved this problem by combining (`xoring`) the output of the hardware random generators with pseudorandom sequences obtained by some deterministic generators [29, file `cdmake.ps`]

> The sixty 10-megabyte files of random numbers are produced by combining two or more of the most promising deterministic generators with sources of random noise from three physical devices (white noise), for those who feel that physical sources of randomness

[1] We also tested the corrected files, replacing groups 13 10 by 10 in `canada.bit` and `germany.bit`. They still fail many tests in the `dieharder` suite.

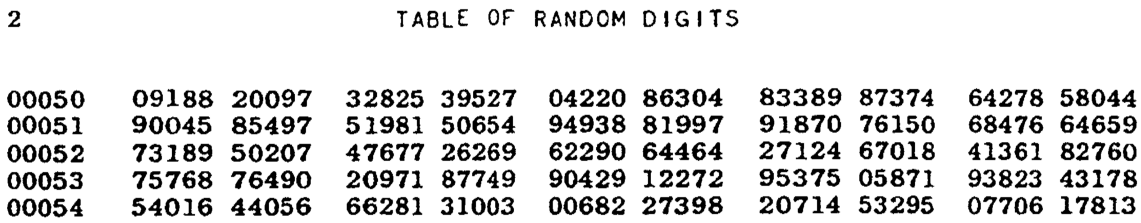


Figure 4.1: A fragment of table of random digits from [48]

> are better than deterministic sources. Some of the files have white noise combined with black noise, the latter from digital recordings of rap music. And a few of the files even had naked ladies thrown into the mix, from pixel files on the network. The last two, digitized music and pictures, are thrown in to illustrate the principle that a satisfactory stream of random bits remains so after combination [`xor`-ing] with the bits of any file.

A similar combination of the hardware source of somehow random bits and post-processing was used for the table of random digits published in 1955 [48] (see Figure 4.1 for a small fragment of it):

> The random digits in this book were produced by rerandomization of a basic table generated by an electronic roulette wheel. Briefly, a random frequency pulse source, providing an average about 100,000 pulses per second, was gated about once per second by a constant frequency pulse. Pulse standardization circuits passed the pulses through a 5-place binary counter. In principle the machine was a 32-place roulette wheel which made, on the average, about 3000 revolutions per trial and produced one number per second. A binary-to-decimal converter was used which converted 20 of the 32 numbers (the other twelve were discarded) and retained only the final digit of two-digit numbers; this final digit was fed into an IBM punch to produce finally a punched card table of random digits.
>
> Production from the original machine showed statistically significant biases, and the engineers had to make several modifications and refinements of the circuits before production of apparently satisfactory numbers was achieved. The basic table of a million digits was then produced during May and June of 1947. This table was subjected to fairly extensive tests and it was found that it still contained small but statistically significant biases. ⟨...⟩
>
> [Comparing the results of tests before and after one month of continuous operations:] Apparently the machine had been running down despite the fact that periodic electronic checks indicated that it had remained in good order.
>
> The table was regarded as reasonably satisfactory because the deviations from expectations in the various tests were all very small — the largest being less than 2 per cent — and no further effort was made to generate better numbers with the machine. However, the table was transformed by adding pairs of digits modulo 10 in order to improve the distribution of the digits. There were 20,000 punched cards with 50 digits per card; each digit on a given card was added modulo 10 to the corresponding digit of the preceding card to yield a rerandomized digit. It is this transformed table which is published here ⟨...⟩
>
> These tables were reproduced by photo-offset of pages printed by the IBM model 856 Cardatype. Because of the very nature of the table, it did not seem necessary to proofread every page of the final manuscript to catch random errors of the Cardatype. All pages were scanned for systematic errors, every twentieth page was proofread ⟨...⟩

We see that the same scheme was used here. However, the post-processing algorithms used in both cases are far from perfect. The sum modulo 10 used by RAND is almost reversible (if we know the resulting table and the first card, then we can reconstruct all the cards), so it cannot significantly change the entropy or the Kolmogorov complexity of the data string. This entropy is probably insufficient if simple tests fail on the string. The same can be said about `xor`-ing with a (deterministic) pseudorandom sequence used by Marsaglia. In the latter case, the rap music and naked ladies could save the day, assuming that these strings have enough complexity (generating processes have enough entropy) and are independent from the data from the electronic devices. But obviously one would like to have less frivolous and more regular procedure.

4.2 Random source and post-processing

Noise sources are cheap and easy to find. A classical example is a Zener diode which costs few cents; the noise generated by it is strong enough to be captured by an inexpensive audio card that has microphone inputs, and one can then try to convert this noise to a high-quality random bits ("white noise") using some processing called "conditioning" or "whitening". Many commercial devices uses this scheme (usually with a higher frequency and a lower precision than used in typical audio cards); here is the description of one of devices of this type:

> The TrueRNG Hardware Random Number Generator uses the avalanche effect in a semiconductor junction to generate true random numbers. The avalanche effect has long been used for generation of random number / noise and is a time-tested and proven random noise source. The semiconductor junction is biased to 12 volts using a boost voltage regulator (since USB only supplies 5V), amplified, then digitized at high-speed. The digitized data is selected and whitened internal to the TrueRNG and sent over the USB port with more than 400 kilobits/second of throughput. ⟨...⟩
>
> The new entropy mixing algorithm takes in 20 bits of entropy and outputs 8 bit to ensure that maximum entropy is maintained. The algorithm uses multiplication in a Galois field similar to a cyclic redundancy check to mix the ADC inputs thoroughly while spreading the entropy evenly across all bits [59].

On the other hand, one should be careful here, since the properties of Zener diodes are not guaranteed[2], as a simple experiment shows (Figure 4.2). It is quite possible that noise properties may change over time and depend on the environment (exact voltage and current, temperature etc.) The post-processing should somehow be robust enough to convert these varying types of noise into random bits with the same uniform distribution.

The scheme of such a hardware random number generator is shown in a picture from NIST publication [39] (Figure 4.3). In addition to the analog source and the conditioning block this scheme also provides a "health tests" block, with the obvious goal to raise an alarm when, for example, the analog noise source becomes broken for some reason, or drastically changed its parameters. The circuit is called an "entropy source", not a random bits generator, and the conditioning block is optional, since in [39] a more complicated scheme is considered: the output of this block is subjected to the next layer of conditioning before being sent to the customer. See below Section 4.4.

4.3 What we would like to have

The ideal situation can be described as follows. There exist

[2]The manufacturers of Zener diodes do not care much about the noise since the primary purpose of Zener diodes is different (and somehow opposite): to produce a stable voltage.

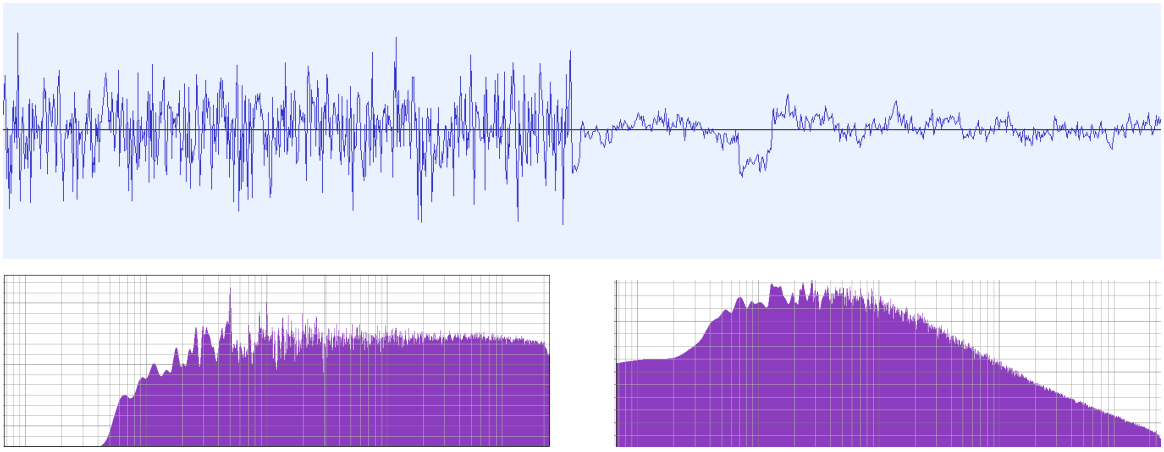


Figure 4.2: The noise signal and its spectrum for two Zener diodes from the same roll, digitized by the same sound card (Behringer 1204usb) and analyzed by the same program (`audacity`).

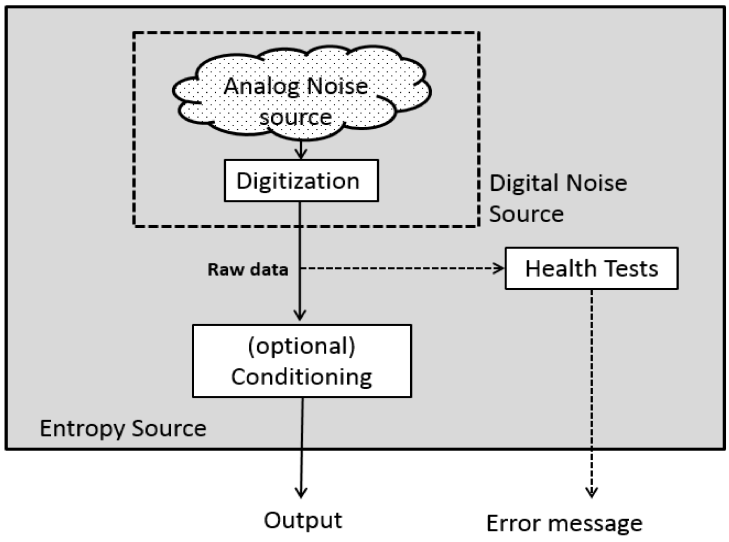


Figure 4.3: A general scheme of a hardware entropy sources [39, p. 5]

- some mathematical property (E) of the output distribution of the (digital) noise source; its informal meaning is that "there is enough randomness in the output of the noise source";
- some hardware device for which the physicists guarantee (E) unless the device is visibly broken;
- a deterministic transformation ("conditioning") and a mathematical theorem that guarantees that the output of this transformation is distributed almost uniformly if its input has the property (E).

Unfortunately, the current practice is rather far from this ideal. The NIST publication mentioned above suggests the property "min-entropy is large" as (E). This means that each individual outcome has small probability (by definition, the min-entropy of a distribution is at least k if every outcome has probability at most 2^{-k}). Here is what they say:

> The central mathematical concept underlying this Recommendation is entropy. Entropy is defined relative to one's knowledge of an experiment's output prior to observation,

> and reflects the uncertainty associated with predicting its value — the larger the amount of entropy, the greater the uncertainty in predicting the value of an observation. There are many possible measures for entropy; this Recommendation uses a very conservative measure known as min-entropy, which measures the effectiveness of the strategy of guessing the most likely output of the entropy source [39, p. 4].

However, min-entropy, being a very important notion, is still not enough to guarantee the good distribution after any (deterministic) conditioning transformation. Namely, for any transformation $T : \mathbb{B}^n \to \mathbb{B}$ that maps n-bit strings into bits, there is a random variable ξ with values in $\mathbb{B}^n$ that has min-entropy at least $n-1$ (almost maximal) such that $T(\xi)$ is a constant. Indeed, one of the preimages $T^{-1}(0)$ and $T^{-1}(1)$ has size at least 2^{n-1}, and we may let ξ be uniformly distributed in this preimage.

Moreover, even a stronger requirement than high min-entropy, introduced long ago by M. Santha and U. Vazirani [50], is not enough. This requirement, for a sequence of random Boolean variables $\xi_1, \xi_2, \dots, \xi_n$, says that

$$\Pr[\xi_m = 1 | \xi_1 = x_1, \dots, \xi_{m-1} = x_{m-1}] \in \left(\frac{1}{2} - \delta, \frac{1}{2} + \delta\right)$$

for every $m \leqslant n$ and for every $(m-1)$-bit string $x_1 \dots x_{m-1}$. Here $\delta \in (0, 1/2)$ is some constant. Assume for example that $\delta = 1/6$; then the requirement says that whatever bits we have observed, the conditional probabilities to have 0 and 1 as the next bit differ at most by factor 2, being in the interval $(1/3, 2/3)$. This implies that min-entropy (and Shannon entropy) grows linearly with n, but is a much stronger condition, saying that in no circumstances we may predict the next bit reliably. Still, as proven in [50], this condition is not enough to extract even one "whitened" bit: there is no whitening algorithm that is better than the trivial one (taking the first bit). This claim can be slightly generalized: no way to extract k bits is better than the trivial one (just taking the first k bits). Here is the exact statement that implies this result.

Proposition 4. *Let $A \subset \mathbb{B}^n$ be a subset that has uniform probability p and let $\delta \in (0, 1/2)$. Then there exists a sequence of n random Boolean variables $\xi_1, \dots, \xi_n$ that satisfies the Santha–Vazirani condition for this δ, such that*

$$\Pr[\xi_1 \dots \xi_n \in A] \geqslant p^\alpha,$$

where α is a number such that $(1/2)^\alpha = \left(\frac{1}{2} + \delta\right)$.

Therefore, if the uniform probability of A is $(1/2)^k$ for some k, then the probability of the event $\xi_1 \dots \xi_n \in A$ guaranteed by Proposition 4 (for some Santha–Vazirani source) is at least $(1/2+\delta)^k$. This means that no transformation $T : \mathbb{B}^n \to \mathbb{B}^k$ can be better (in terms of extracting min-entropy from Santha–Vazirani source) than taking the first k bits. Indeed, one of the points in $\mathbb{B}^k$ has T-preimage in $\mathbb{B}^n$ of uniform probability at least $(1/2)^k$, and applying Proposition 4 to this preimage we conclude that for some Santha–Vazirani source $\xi_1, \dots, \xi_n$ the min-entropy of $T(\xi_1 \dots \xi_n)$ is not better than just for $\xi_1 \dots \xi_k$: probability of some point in the image distribution is at least $(1/2 + \delta)^k$.

Proof. We need to construct a distribution on $\mathbb{B}^n$ that satisfies Santha–Vazirani condition and assigns large probability to A. We do it inductively, following the suggestion by Ruslan Ishkuvatov. The case $n = 1$ is obvious. For $n > 1$, we split A into two parts $0A_0$ and $1A_1$ according to the first bit, where A_0 and A_1 are subsets of $\mathbb{B}^{n-1}$ that can be considered as two faces of the Boolean cube. Let p_0 and p_1 be the probabilities of A_0 and A_1 according to the uniform distribution in $\mathbb{B}^{n-1}$, so their average is p. Assume that $p_0 \leqslant p_1$, so $p_0 = p - x$ and $p_1 = p + x$ for some $x \in [0, p]$. The induction assumption gives two Santha–Vazirani distributions on $\mathbb{B}^{n-1}$ that give large probabilities to A_0 and A_1, namely, at least p_0^α and p_1^α. They can be combined into one distribution on $\mathbb{B}^n$, we only need to chose the probability (between $1/2 - \delta$ and $1/2 + \delta$) for the first bit, and then use the two Santha–Vazirani

distributions provided by the induction assumption as conditional distributions. To maximize the resulting probability, we should put maximal allowed weight on the face where the probability is greater. It remains to prove then that

$$\left(\frac{1}{2}-\delta\right)p_0^\alpha+\left(\frac{1}{2}+\delta\right)p_1^\alpha=\left(\frac{1}{2}-\delta\right)(p-x)^\alpha+\left(\frac{1}{2}+\delta\right)(p+x)^\alpha\geqslant p^\alpha.$$

For $\alpha\leqslant 1$, the function $t\mapsto t^\alpha$ is concave, therefore the left hand side is a concave function of x, and it is enough to check this inequality for endpoints $x=0$ (where it is obvious), and $x=p$. In the latter case we need to prove that $(1/2+\delta)(2p)^\alpha\geqslant p^\alpha$, and this follows directly from the definition of α. □

Remark 10. A very simple proof for the case of 1-bit output [49] goes as follows. If $p\geqslant 1/2$, then for some Santha–Vazirani distribution with parameter δ one can achieve probability at least $1/2+\delta$. Why? It is enough to spread the probability $1/2+\delta$ uniformly on a subset $A'\subset A$ with uniform probability 1/2, and spread the remaining probability $1/2-\delta$ uniformly on the complement of A'.

So the situation is far from ideal: large min-entropy and even stronger Santha–Vazirani condition are not enough to guarantee the correct distribution after whitening, for any fixed whitening function. Still some practical solutions, i.e., some common sense recommendations that help us to avoid obviously faulty generators, are needed, even if no theoretical guarantees are provided. In the next section we look at the NIST approach to this problem.

4.4 Practical approaches

There are three documents produced by NIST that cover different aspects of random bits generation. The first, SP 800-90A [38], deals with algorithmic pseudorandom bits (or numbers) generators, called there *deterministic random bits generators*[3]. Here the relevant mathematical theory is not the algorithmic information theory but the complexity theory where the notion of (cryptographically strong) pseudorandom number generator was introduced by Manuel Blum, Silvio Micali and Andrew Yao [6, 62]. This theory goes far beyond the scope of our survey.

Roughly speaking, such a generator is a polynomial-time algorithm that maps a truly random seed into a (much longer) sequence of bits that is "indistinguishable" from a random one by polynomial-size circuits. The indistinguishability means that no polynomial-size circuit can have significantly different probabilities of (a) accepting the output of the generator for a truly random seed, and (b) accepting a sequence of truly random bits. An equivalent definition says that there is no way to predict by a polynomial-size circuit the next bit of the output sequence (for a random seed) significantly better than by guessing.

The existence of generators with these properties is equivalent to the existence of one-way functions [16], and this existence is an unproven assumption. This assumption implies P ≠ NP, while the reverse implication is not known. For this reason we do not know any cryptographically strong pseudorandom number generator for which this property can be proven. Moreover, since the constructions from [16] are quite complicated, practical pseudorandom generators may use stronger assumptions, like hardness of factoring, or just have no theoretical justification at all. In fact, NIST [38] not only recommends but also insists on using "allowed" methods to generate random bits from the seed, and these methods are far from being justified mathematically, even in a very weak sense. For example, one of the methods uses hash values for consecutive bit strings (see Figure 4.4). As explained in [38, page 37], "mechanisms specified in this Recommendation have been designed to use any **approved** hash function and may be used by consuming applications requiring various security strengths, providing that the appropriate hash function is used and sufficient entropy is obtained

[3] Note an oxymoron.

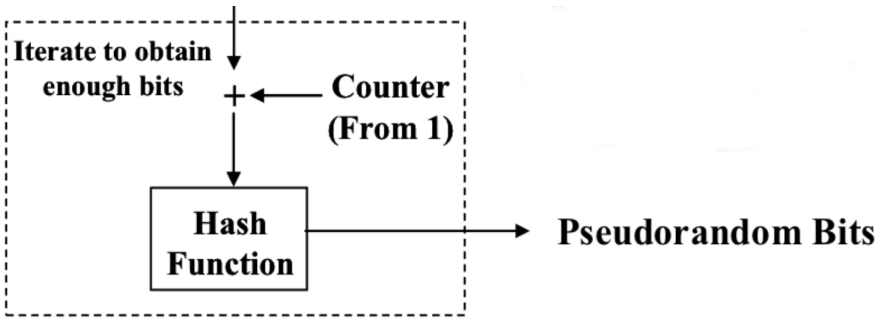


Figure 4.4: Part of Figure 8 on p. 39 in [38] related to the random bit generation using a hash function (the initialization part is omitted).

for the seed". On the next page a list of these "approved" hash functions is provided that includes SHA-1, SHA-224, SHA-512/224, SHA-256, SHA512/256, SHA-384, SHA-512. According to NIST [41]:

> An approved hash function is expected to have the following three properties:
>
> 1. Collision resistance: It is computationally infeasible to find two different inputs to the hash function that have the same hash value. That is, if *hash* is a hash function, it is computationally infeasible to find two different inputs x and x' for which $hash(x) = hash(x')$. Collision resistance is measured by the amount of work that would be needed to find a collision for a hash function with high probability. If the amount of work is 2^N, then the collision resistance is N bits ⟨...⟩
> 2. Preimage resistance ⟨...⟩
> 3. Second preimage resistance ⟨...⟩

Obviously, for a specific function like SHA-1 or any other mentioned in the list above, the collision resistance requirement makes no sense if understood literally: computation infeasibility means high complexity of some function, and here we have no function. If somebody comes with a collision pair x, x', the collision resistance in the naïve sense disappears starting from this moment, so it is not a mathematical property of a hash function (a mathematical property cannot suddenly become false), but some property of the current state of art (still measured in bits!). Moreover, the hash functions mentioned above are obtained by a complicated *ad hoc* construction and there are no reasons to believe that something similar to collision resistance can be proven. Finally, as it is mentioned [38, p. 89],

> Hash_DRBG's [the random generator based on hash functions] security depends on the underlying hash function's behavior when processing a series of sequential input blocks. If the hash function is replaced by a random oracle, Hash_DRBG is secure. It is difficult to relate the properties of the hash function required by Hash_DRBG with common properties, such as collision resistance, pre-image resistance, or pseudorandomness.

Indeed, it is impossible to relate the "required" properties with "common" properties, since a function with no known collisions and high preimage resistance still may have much more 1s than 0s in most of its outputs, or have the last bit always equal to 1, therefore being completely unsuitable for Hash_DRBG.[4] So the reference to the security properties of the allowed hash functions can only create a false feeling of security.

The second NIST publication, SP 800-90B [39], describes the allowed constructions of the "entropy source" (see Fig. 4.3 above) while the third one [40] "addresses the construction of RBGs from

[4]And the claim about the random oracle model is obviously true and obviously irrelevant.

the mechanisms in SP 800-90A and the entropy sources in SP 800-90B" [39, p. 1]. The idea here is that the whitening (conditioning) process is splitted into two stages. The first stage, described in SP 800-90B, does only some "rough" conditioning and may not produce a distribution that is very close to the uniform one. We hope only that the entropy of its output is close to the output length or at least is a significant fraction of the length. Then the second stage that may involve deterministic random bits generators or not is used for "fine-tuning".[5]

But what is meant by "entropy" in this description? As we have said, the NIST recommendations claim to use min-entropy. Still there are some problems with this approach.

- There is no way to get a reliable lower bound for the min-entropy of a physical source. If there is some isolated value that appears with probability 2^{-k}, the min-entropy is at most k, but one needs to make $\Theta(2^k)$ trials to have a reasonable chance to see this value at least once.[6]

- Quite often the NIST recommendations treat the notion of entropy informally, as some mystical substance that can be present in a binary string (and not in a random variable) and even can be accumulated and/or condensed[7]. For example, it is written [39, p. 11] that "in all cases, the DRBG [deterministic random bits generator] mechanism expects that when entropy input is requested, the returned bitstring will contain at least the requested amount of entropy." The closest approximation to this interpretation is Kolmogorov complexity, but it is (a) non-computable and (b) defined up to a constant, and different reasonable optimal programming languages easily can give the values that differ by several thousands, so it does not make sense to ask whether the complexity exceeds (say) 512 or not.

- Moreover, in some cases even more enigmatic explanations are given: "For the purposes of this Recommendation, an n-bit string is said to have full entropy if the string is the result of an approved process whereby the entropy in the input to that process has at least $2n$ bits of entropy (see [ILL89] and Section 4.2)" [40, p. 11]. Here [ILL89] is the preliminary version of [16] and neither says anything about approved processes nor justifies the requirement about $2n$ bits of entropy.

- As mentioned above for a similar situation, the properties of the functions used for conditioning, in particular the standard requirements for the security of a hash function, do not guarantee, even informally, that the output distribution for the hash function applied to an input source of high min-entropy, is close to the uniform distribution. However, this is implicitly assumed in the recommendations when an approved process of obtaining a string of full entropy is described.

- The recommendations encourage the designer to use different combinations of "approved" constructions (see, e.g., Fig. 4.5); even if some good properties of one-stage construction are plausible, the claim that the composition of several stages will still have good properties, is much less founded.

All these critical remarks do not mean that NIST recommendations are unnecessary: they reflect the current state of technology, the existing practice and prevent the appearance of completely bogus

[5]The final stage may also use pseudorandom bit generators to provide additional "backup" layer if the physical source stops working. For example, one may follow Marsaglia and produce `xor` of the bit sequences from physical and deterministic sources. Note that this operation, hiding the problems with physical source, makes the testing of the output sequence almost useless; testing should be done before this last step.

[6]Things are much better for independent identically distributed (i.i.d.) variables [39, p. 11]; there are also some physical sources where i.i.d. assumptions are reasonable, and some tests that can detect some violations of i.i.d. property.

[7]"When the entropy [string?] produced by the entropy source(s) is very long (e.g., because the entropy rate of the entropy source(s) is very low), and the entropy bits may need to be condensed into a shorter bitstring, the **Get_Entropy** function in Section 10.3.1.1 or Section 10.3.1.2 shall be used to condense the entropy bits without losing the available entropy in the bit string." [39, Section 10.3.1, p. 44]

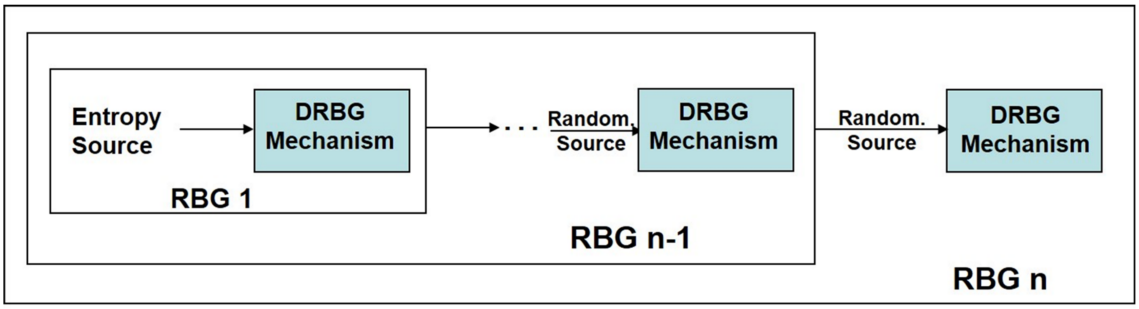


Figure 4.5: The construction of a random bits generator with several layers [40, p. 19].

generators, therefore playing a very important role. However, one should keep in mind that they are not based on any "hard science"; they sometimes use mathematical notions and results but only as hints and sources of inspiration.

Could we have better recommendations? This is a difficult question. One can hope for the *security through obscurity*: if a long sequence of different mathematical operations is performed, this could make an attack much more difficult. However, the idea that *random actions give random results* does not look as a good plan for designing random bits generators. It could be that the careful choice of a noise source plus one-layer conditioning procedure that is based on something more suitable than just hash functions, would give a better result than a complicated multi-layer approach using cryptographic primitives.

Chapter 5

Randomness extractors: theory and practice

5.1 Definitions

Randomness extractor is a function $F : \mathbb{B}^n \times \mathbb{B}^d \to \mathbb{B}^m$. The first input ($n$-bit string) is understood as a "long and somehow random" bit string. For example, one may assume that its min-entropy is at least some k (that is smaller than n but reasonably large, say, $n/2$ in some application). The second input (d-bit string) is a "short and perfectly random" string (say, d may be $O(\log n)$ in some applications). We want the output to be indistinguishable from m-bit random string. As a measure of indistinguishability we consider the statistical distance between two distributions, defined as

$$d(P,Q) = \sup_{A \subset \mathbb{B}^m} |P(A) - Q(A)|.$$

Here P and Q are probability distributions on $\mathbb{B}^m$, and this distance can be rewritten in terms of individual probabilities and becomes l_1-distance with factor 1/2.

The extractor requirement: for every random variable X with $\mathbb{B}^n$-values that has min-entropy at least k, and for independent uniform random variable $Y \in \mathbb{B}^d$, the statistical distance between the image distribution $F(X,Y)$ and the uniform distribution on $\mathbb{B}^m$ is at most ε. (The definition has parameters n, d, m, k, ε.)

Informally, there are at least k "bits of entropy" in X and d bits in Y, so we may hope to extract $k+d$ bits, so our dream is to have $m \approx k+d$. This length could be almost achieved (existence proven by a probabilistic argument) and there are explicit constructions with slightly worse parameters.

From the practical viewpoint, the obvious idea is the "bootstrapping", increasing the number of random bits by using weak random source of large size. However, if we just have some weak random source, it is not clear how exctractors could help. Note that nothing can be said about the distribution of $F(X,y)$ where X is a weak random source and y is some fixed string. The definition of extractor guarantees that (for given X) the *average* of all the distributions obtained for different values of y, is close to the uniform one, but does not say anything about individual distribution. For example, it is quite possible that the first bit of $F(x,y)$ coincides with the first bit of y, and for fixed y this first bit will be fixed — not suitable for a lottery use, for example.

5.2 Strong extractors

There is a stronger notion of extractor: we can require that not only $F(X,Y)$ has almost uniform distribution, but also the pair $\langle Y, F(X,Y)\rangle$ has almost uniform distribution (in $\mathbb{B}^{m+d}$).[1]

The existence of strong extractors (now for $m \approx k$, since d bits of Y are added to the output) also can be proven by a probabilistic argument. Here is the sketch of this argument.

[1]Equivalently, we could include Y in $F(X,Y)$ and require that first d bits of the extractor output coincide with its second input.

A mapping $F : \mathbb{B}^n \times \mathbb{B}^d \to \mathbb{B}^m$ can be considered as a family of 2^d mappings F_y of type $\mathbb{B}^n \to \mathbb{B}^m$ indexed by d-bit strings: $F_y(x) = F(x, y)$. The definition of statistical distance for the distribution of pairs $\langle y, F(x, y)\rangle$ uses a subset $A \subset \mathbb{B}^d \times \mathbb{B}^m$. This subset can be also considered as a family of subsets

$$A_y = \{z : \langle y, z\rangle \in A\}.$$

The probability of $\langle Y, F(X, Y)\rangle \in A$ is the average (over y) the probability of $F_y(X) \in A_y$. We should find F such that for all A this probability is close to the probability of A, i.e., the average probability of A_y (taken over y).

As usual, any probability distribution of min-entropy at least k can be represented as a convex combination of the extreme points, namely, the uniform distributions over a 2^k-element subset of $\mathbb{B}^n$. Preparing to use the union bound, we note that there are at most

$$(2^n)^{2^k} = 2^{n2^k}$$

subsets of this size, and there are at most $2^{2^{m+d}}$ sets A for which the condition (small difference between probabilities) should hold. Therefore we need to estimate the probability, for a random function F, that the difference is large, and show that it is less than

$$2^{-n2^k - 2^{m+d}},$$

for suitable parameters.

To compute the probability of large deviation, we use Azuma–Hoeffding inequality. Indeed, the random choice of F is equivalent to 2^{n+d} random selections of elements in $\mathbb{B}^m$. These selections are divided into 2^d groups indexed by $y \in \mathbb{B}^d$. For a given y, we look how many of the selected elements are in A_y. The difference (inside the group) is the difference between probability and frequency for 2^n Bernouilli trials with probability $\Pr[A_y]$. In total, we have a sequence of independent trials with different probabilities (i.e., with probabilities $\Pr[A_y]$ for different y). The total number of trials in 2^{k+d}, so the Azuma–Hoeffding bound says that the probability of the deviation at least ε is

$$2 \cdot 2^{-\Omega(2^{k+d}\varepsilon^2)}$$

So the condition of success is

$$2^{k+d}\varepsilon^2 > c(n2^k + 2^{m+d})$$

for some constant c, and this gives two conditions:

$$k + d \geqslant k + \log n + 2\log\frac{1}{\varepsilon} + c$$
$$k + d \geqslant m + d + 2\log\frac{1}{\varepsilon} + c,$$

(for some other c), so d should exceed $\log n + 2\log(1/\varepsilon)$ and k should exceed $m + 2\log(1/\varepsilon)$ by some constant.

5.3 Why strong extractors?

The philosophical reasons for practical use of strong extractors can be explained as follows. We construct some device that produces (say, using random noise) an n-bit string for rather large n. We do not know the exact distribution for the random output X of this device, but (based on the physics laws) believe that this distribution has significant min-entropy, at least k for some reasonable large k (say, $n/2$ or something like this). Now, having a strong extractor F, we know that the not only *the statistical distance between the uniform distribution U_m on $\mathbb{B}^m$ and the average of 2^d distributions $F_d(X)$ for different $d \in \mathbb{B}^k$ is at most ε*, but also that *the average statistical distance between $F_d(X)$ and*

U_m *is at most* ε (we can combine the sets A_d that achieve the statistical distance between $F_d(X)$ and U_m, into one set A). Therefore, for all d, except for $\sqrt{\varepsilon}$-fraction, the distance between $F_d(x)$ and U_m is at most $\sqrt{\varepsilon}$.

If ε is very small (say, 2^{-200}), then $\sqrt{\varepsilon}$ is also small enough (2^{-100}) to be practically sure that if we get d random bits from some high-quality source of randomness, we do not get "bad" d. And under this assumption $F(X_d)$ will be statistically close (distance 2^{-100} to U_m. Moreover, if we make many (say, 2^{20}) hardware random bit generators *using the same* d, using different physical source of randomness (say, different Zener diodes), we also can be practically sure (error probability 2^{-80}) that *all* these generators will have output distribition close to U_m.

Of course, the practical value of this construction depends on the complexity of F, and the probabilistic argument here is not very useful (we can apply the same reasoning to the choice of random F, but the number of bits used in F is astronomical, so there is no chance to store them in any device).

5.4 Strong extractors from hash functions

For the reader's convenience, in this section we reproduce the construction of a strong extractor from the breakthrough paper that derived pseudo-random generators from one-way functions [16]. This construction will appear in the analysis of one of the hardware random generators below (ID Quantique).

Assume that $h_r(x)$ is an universal family of hash functions. Here $x \in \mathbb{B}^n$ and r (an index of a hash function h_r) is taken from $\mathbb{B}^s$ for some s; the value $h_r(x)$ is in $\mathbb{B}^m$ for some m. The universality means that for every fixed $x \in \mathbb{B}^n$ and (uniformly) random $r \in \mathbb{B}^s$ the value of $h_r(x)$ is uniformly distributed in $\mathbb{B}^m$, and, moreover, for every two different x_1 and x_2 the random variables $h_r(x_1)$ and $h_r(x_2)$ (where randomness comes from r) are independent (in other words, $\langle h_r(x_1), h_r(x_2)\rangle$ is uniformly distributed in $\mathbb{B}^m \times \mathbb{B}^m$).

Another notion that is used in this construction is Rényi entropy of a random variable X with finitely many values. It is defined as $H_2(X) = \log_2(1/p)$, where p is the probability for the two independent copies of X to coincide. If X has k values with probabilities $p_1, \dots, p_k$, then $p = \sum_{i=1}^{k} p_i^2$. The Rényi entropy cannot be smaller than min-entropy: if all p_i are at most 2^{-u}, then $\sum p_i^2 \leqslant \sum 2^{-u} p_i \leqslant 2^{-u}$.

Lemma. Let X be a random variable with values in $\mathbb{B}^n$, and $h_r(x)$ be a universal family of hash functions where $x \in \mathbb{B}^n$, the index r ranges over some $\mathbb{B}^s$, and values are in some $\mathbb{B}^m$. If the Rényi entropy of X exceeds m by $2d$ for some d, then the statistical distance between the random variable $\langle r, h_r(X)\rangle$ (where r is uniformly distributed in $\mathbb{B}^s$ and independent from X) and the uniform distribution on $\mathbb{B}^s \times \mathbb{B}^m$ is at most 2^{-d-1}.

(The factor 1/2 appears because the statistical distance is one half of L_1-distance that appears in the derivation.)

Proof. If we ignore the factor 2, the statistical distance between two distribution P, Q on some product $U \times V$ is by definition $\sum |p_{u,v} - q_{u,v}|$ over all pairs $\langle u, v\rangle$ is a sum over u of the corresponding sums over v (or vice versa). Therefore, the distance is small if *for every* u the L_1-distance between the corresponding rows $p_{u,*}$ and $q_{u,*}$ is small.

Trying to use this argument, we may fix $r \in \mathbb{B}^s$ or $y \in \mathbb{B}^m$. The first approach does not help, since the distribution $h_r(X)$ can be very far from the uniform (if X is selected by an adversary for some fixed r). So we go the other way and show that

$$2^{-s} \sum_{r \in \mathbb{B}^s} \left| \mu_X(h_r^{-1}(y)) - \frac{1}{2^m} \right| \leqslant 2^{-d} \cdot 2^{-m} \qquad \text{for every } y \in \mathbb{B}^m$$

where μ_X stands for the distribution of the random variable X (and the factor 2^{-s} is moved out since it appears in both distributions we are comparing), and then take the sum over all $y \in \mathbb{B}^m$. The left

hand side is an average over r and it is enough to prove similar inequality for the root mean square that is an upper bound for the average. (In other terms, we apply Cauchy inequality.) So it is enough to prove that

$$2^{-s}\sum_{r\in\mathbb{B}^s}\left(\mu_X(h_r^{-1}(y))-\frac{1}{2^m}\right)^2\leqslant 2^{-2d-2m}\qquad\text{for every } y\in\mathbb{B}^m$$

for every y. Denoting the average over $r\in\mathbb{R}^s$ as E_r, we need to prove the upper bound 2^{-2d-2m} for

$$\mathsf{E}_r[(\mu_X(h_r^{-1}(y)))^2]-2\cdot 2^{-m}\,\mathsf{E}_r[\mu_X(h_r^{-1}(y))]+2^{-2m}.$$

Recall that $\mu_X(h_r^{-1}(y))$ is the probability of the event $h_r(X)=y$ (for fixed r,y and random X). Therefore, $\mathsf{E}_s[\mu_X(h_r^{-1}(y))]$ is the probability of the event $h_r(X)=y$ for fixed y and independent random X and r. Changing the order or averages and recalling the properties of the hashing functions ($h_r(x)$ is uniformly distributed in $\mathbb{B}^m$ for every fixed d), we see that $\mathsf{E}_s[\mu_X(h_r^{-1}(y))]=2^{-m}$, so the second term is $2\cdot 2^{-2m}$, and it remains to prove that

$$\mathsf{E}_r[(\mu_X(h_r^{-1}(y)))^2]-2^{-2m}\leqslant 2^{-2d-2m}.$$

The probabilistic meaning of $(\mu_X(h_r^{-1}(y)))^2$ (for given r and y) is the probability of the event $h_r(X_1)=h_r(X_2)=y$ where X_1 and X_2 are two independent copies of X. The average over r gives the probability for independent random r,X_1,X_2. This average can be then rewritten as

$$S=\sum_{x_1,x_2\in\mathbb{B}^n}\mu_X(x_1)\mu_X(x_2)\Pr_r[h_r(x_1)=h_r(x_2)=y].$$

Now we count separately the pairs where $x_1=x_2$ and $x_1\neq x_2$. If $x_1\neq x_2$, then $\Pr_r[h_r(x_1)=h_r(x_2)=y]=2^{-2m}$ due to universal hashing. The sum of $\mu_X(x_1)\mu_X(x_2)$ over all x_1 and x_2 is 1, so we can rewrite S as

$$S=\sum_{x_1=x_2\in\mathbb{B}^n}\mu_X(x_1)\mu_X(x_2)\left(\Pr_r[h_r(x_1)=h_r(x_2)=y]-2^{-2m}\right)+2^{-2m},$$

or

$$S=\sum_{x\in\mathbb{B}^n}\mu_X^2(x)\left(\Pr_r[h_r(x)=y]-2^{-2m}\right)+2^{-2m}.$$

The probability here is 2^{-m}; recalling that we have to prove that $S=2^{-2m}\leqslant 2^{-2d-2m}$, we need to prove

$$\sum_{x\in\mathbb{B}^n}\mu_X^2(x)\left(2^{-m}-2^{-2m}\right)\leqslant 2^{-2d-2m}$$

The term 2^{-2m} is small compared to 2^{-m}; we ignore it and prove the (slightly) stronger inequality

$$2^{-m}\sum_{x\in\mathbb{B}^n}\mu_X^2(x)\leqslant 2^{-2d-2m}$$

The sum is $2^{-H_2(X)}$ by definition, and it remains to note that $H_2(X)\geqslant m+2d$. □

To use this construction, one needs universal hash functions. There is a simple example: a random affine mapping $\mathbb{B}^n\to\mathbb{B}^m$ is a universal family. More precisely, let $s=mn+m$ and let r be a pair: a matrix A of size $n\times m$ that defines a linear mapping $\mathbb{B}^n\to\mathbb{B}^m$, and a vector $b\in\mathbb{B}^m$. We have to check that for any two $x,y\in\mathbb{B}^m$ the random variables $Ax+b$ and $Ay+b$ are independent and uniformly distributed on $\mathbb{B}^m\times\mathbb{B}^m$. The pair (u,v) is uniformly distributed if and only if the same is true for $(u,u-v)$, so we may consider the variables $A(x-y)$ and $Ay+b$ instead. The first is uniformly distributed, since $x-y$ is a fixed non-zero vector and A is a random linear mapping. It remains to show that the second is uniformly distributed with fixed $A(x-y)$ as a condition — but is uniformly distributed even if A is fixed (a more fine-grained condition).

We get a uniform hashing with smaller s (namely, for $s = m+n-1$) if we consider $h_{A,b}(x) = Ax+b$ only for *Toeplitz matrices*, i.e., matrices of the form

$$\begin{Vmatrix} d & e & \ddots & & \\ c & d & e & \ddots & \\ \ddots & c & d & \ddots & \ddots \\ & \ddots & \ddots & \ddots & e \\ & & \ddots & c & d \end{Vmatrix}$$

where the numbers on diagonals are chosen randomly. To apply the same argument, we need to show that for a non-zero vector u the random variable Au (where A is a random Toeplitz matrix) is uniformly distributed. One can reformulate the question as follows: we have a non-zero vector u of length n and move it along the random vector of length $m+n-1$ computing the inner products in different positions; why these inner products are all independent? To see why, consider the right-most non-zero element in u. Due to this element, each shift of u to the right adds one fresh Toeplitz coefficient, and therefore the probability to get 0 and 1 (conditional to previous inner products) is 1/2.

5.5 Two-sources extractors

Another approach is to use two-source extractors. It looks promising since we often assume independence (in the probabilistic sense) for the outputs of different physical processes just because that happen in different place or in different times (without any plausible mechanism of influence).

The simplest case is `xor`ing the bits from two generators that are assumed to be independent. Is is often done in practice. A simple computation shows that if we have several slightly biased but independent coins, the `xor` of their bits will be much less biased. This approach is often used in practice (e.g., the `bitbabbler` generator, Section 6.8, uses four generators).[2] From the cryptography point of view this approach is also desirable: if only one of two `xor`ed generators is compromised by an attacker but the other is still perfect and they are still independent, the output will be perfect.

However, the `xor` appoach needs that different bits from the same generator are independent. (If two generators independently generate n-bit sequences that always contain even number of ones, their `xor` will have the same property.) One can use a more complicated construction that uses weaker assumptions. Still the construction should be not too complicated to be practical (and should not involve large constants). Some candidates are described in [12, 8]. The theoretical notion here is a *two-source extractor*. Consider a function $f(x,y)$ that takes two n-bit strings and produces a m-bit string (with $m \leqslant n$). Let $l \leqslant n$ be some bound. The function f is called (l,ε) *two-source* expander if for every two independent random variables X, Y with values in $\mathbb{B}^n$ and min-entropy at least l, the statistical distance between the distribution of $f(X,Y)$ and the uniform distribution on $\mathbb{B}^m$ is at most ε.

The following result is proven in [12, Section 3.1].

> If $A_1, \ldots, A_k$ are $n \times n$ matrices over two-element field such that every (nonempty) sum of these matrices has full rank, then the function
>
> $$\langle x,y\rangle \mapsto A_1(x,y)A_2(x,y)\ldots A_m(x,y)$$
>
> is a $(l, 2^{-s})$ two-source expander for every l and for
>
> $$s = \left(l - \frac{n}{2}\right) - \frac{m}{2} + 1.$$

[2] One should be careful here, since some common factor (like power supply noise) could affect all the generators at the same time. So one could prefer to use several USB devices plugged into different computers, or combine that data that are obtained not at the same time (`bitbabbler` uses this approach too, calling it *folding*).

Here x and y are n-bit vectors, and matrices A_i are interpreted as bilinear forms, i.e.,

$$A(x,y) = \sum_{i,j} a_{ij} x_i y_j$$

(all operations are performed in a two-element field, i.e., modulo 2).

What does it mean from a practical viewpoint? We need $l > n/2$: the more is the difference, the better trade-off we get for m and error probability $\varepsilon = 2^{-s}$. It is not unrealistic to consider sources with, say, $l = 0.9n$. For example, Santha–Vazirani sources where each next bit has conditional probability $50 \pm 3\%$ satisfy this condition. Then we can have $m = 0.5n$ and still have ε about $2^{-0.15n}$. Which gives quite practical values, say, for $n = 1000$: from two independent 1000-bit sequences with min-entropy 900 we get a 500-bit sequence whose distribution is 2^{-150}-close to the uniform distribution in the statistical sence (probably enough for most practical purposes).

For this construction we need some binary matrices $A_1, \ldots, A_m$ of size $n \times m$ that have full rank, and, moreover, any sum of a (nonempty) subset of these matrices has a full rank. There is a simple algebraic construction that provides such a matrices for $m = n$: consider $\mathbb{B}^n$ as a finite field and consider some basis $a_1, \ldots, a_n$ in this field as a linear space over two-element field. Then field multiplication by a_i is a linear mapping in this field; let A_i be the corresponding matrix (in any basis). The sum of matrices corresponds to the multiplication by the corresponding sum of vectors a_i, and this sum is non-zero (since a_i are linearly independent over two-element field). The multiplication by a non-zero field element is a bijection, so the corresponding matrix has a full rank. This gives a simple construction that is optimal: for $m > n$ there is a subset whose sum is not a full rank. Indeed, consider images of some fixed vector: we have more than m images, so some combination of them is zero, and the corresponding combination of matrices has non-zero kernel and cannot have full rank.

There is an important practical concern related to this construction. We rarely need some fixed number of random bits (except, may be, for the case of lotteries). We would prefer a long stream of bits from which we can take as many bits as we want. Still the size of the matrices in this construction cannot be very large. So we come to the following question. Assume that we have two generators that produce random bits. These bits are grouped into n-bit samples $x_1, x_2, \ldots$ (for the first generator) and $y_1, y_2, \ldots$ (for the second generator). Then we apply f and combine the resulting m-bit strings $f(x_1, y_1), f(x_2, y_2), \ldots$ into a long string of bits. Can we prove some randomness properties for this combined string?

The following simple observation helps. Let X, Y and X', Y' be two pairs of random variables with values in some product $\mathcal{X} \times \mathcal{Y}$. Assume that X and X' are statistically close (statistical distance at most ε). Assume also that for every $x \in \mathcal{X}$ the conditional distributions $(Y|X = x)$ and $(Y'|X' = x)$ are statistically close (the statistical distance is at most δ). *Then the statistical distance between pairs X, Y and X', Y' is at most $\varepsilon + \delta$.*

To see why it is true, consider an intermediate pair $X, \hat{Y}$ where the first variable has the same distribution as X while the second variable $\hat{Y}$ has the same *conditional* distribution for every fixed value of X. The statistical distance (up to a constant 2) can be defined as the supremum of the difference of expectations for every test function $t : X \times Y \to [-1, 1]$. When we switch from X, Y to $X, \hat{Y}$, we use the same distribution on the first coordinate for averaging, and use two δ-close distribution for averaging along the second coordinate, so the diffference is the average of the function bounded by δ, so it does not exceed δ. When we compare $X, \hat{Y}$ and X', Y', we use the same averaging along the second coordinate (conditional distributions are the same), so we take the averages of the same function (with values in $[-1, 1]$) according to X and X', and they differ by at most ε.

The same argument can be applied for longer tuples instead of pairs (and the bounds for statistical distances add up). It would be acceptable in practice if an exponentially small distance (like 2^{-150}) is multiplied by the number of groups used (like 10^{20}), still the product is very small. But we have to show that the small distance between *conditional* probabilities follows from some physically plausible assumptions.

Assume that we have two generators; the first one produces bit blocks x_1 and then x_2; the second one produces blocks y_1 and y_2. All four blocks have length n. We apply the extraction procedure twice and get m-bit blocks $z_1 = f(x_1, y_1)$ and $z_2 = f(x_2, y_2)$. What should we assume to guarantee small statistical distance between pair z_1, z_2 and uniform distibution on $\mathbb{B}^{2m}$? To apply the observation made above, we need two things.

- First, we need that the distribution of z_1 is close to the uniform distribution. This can be guaranteed by the result from [12] cited above.
- Second, we need that the *conditional distribution of z_2 under the condition that z_1 has some fixed value, is close to the uniform.* This is not a direct consequence of the result mentioned, since the condition $z_1 = Z_1$ (for fixed $Z_1 \in \mathbb{B}^m$) makes x_2 and y_2 dependent. Indeed, $z_1 = f(x_1, y_1)$, and x_2 may depend on x_1, and y_2 may depend on y_1. However, the event $f(x_1, y_1) = Z_1$ is a disjoint union of several events $x_1 = X_1, y_1 = Y_1$ (the union is taken over all pairs of strings X_1 and Y_1 such that $f(X_1, Y_1) = Z_1$). It is enough to show that for each of this events the conditional probability is close to the uniform distirbution (then the combined conditional probability, being a weighted sum, is also close to the uniform distribution). These conditional probability corresponds to the distribution of $f(x_2, y_2)$ if (x_2, y_2) has conditional distribution with condition $x_1 = X_1, y_1 = Y_1$. Let us assume (as before) that (x_1, x_2) is independent with (y_1, y_2). Then x_2 and y_2 are conditionally independent with conditions $x_1 = X_1$, $y_1 = Y_1$. So we can apply the result from [12] again. The only thing we need is the high *conditional* min-entropy of x_2 under condition $x_1 = X_1$, and of y_2 under condition $y_1 = Y_1$.

How realistic could be the latter assumption? It says that for each given value X_1 of x_1, the conditional probability of each specific value X_2 of x_2 is very small. In other words, we never are able, after seeing the value of x_1, guess the value X_2 of x_2 with more than 2^{-l} chances of success. For practical generators this is probably not true: even a good generator has some small probability of "general failure" that makes it producing (say) zeros. This probability can be small, say, 2^{-800} for a 1000-bit generator, so the min-entropy is still high (just a bit smaller than 800, since the zero sequence can also appear with a non-zero probability in a normal operation, say, with probability 2^{-1000}). But this general failure will make our assumption false: most of the cases when x_1 are all zeros are due to a general failure, so the conditional probability of $x_2 = 0 \ldots 0$ with condition $x_1 = 0 \ldots 0$ is close to 1. But we may still believe that "in the absense of some general failure" the condition is true. In other term, we may assume that (x_1, x_2) is statistically very close to a distribution with required properties (and this is enough for our argument, just one more small statistical distance is added).

The practical implementation of this approach is discussed in Section 7.6.

Chapter 6

Practical random number generators

As a part of the RaCAF project, we tried to get samples of random bit generators that exists on the market. There are some expensive ones (sometimes without price quotes, being sold as a part of some bigger contracts) which we skipped intentionally, but we have tried to get what was easily available in the price range 30–1000 euros. In this section we collected some remarks and observations.[1]

6.1 Altus Metrum

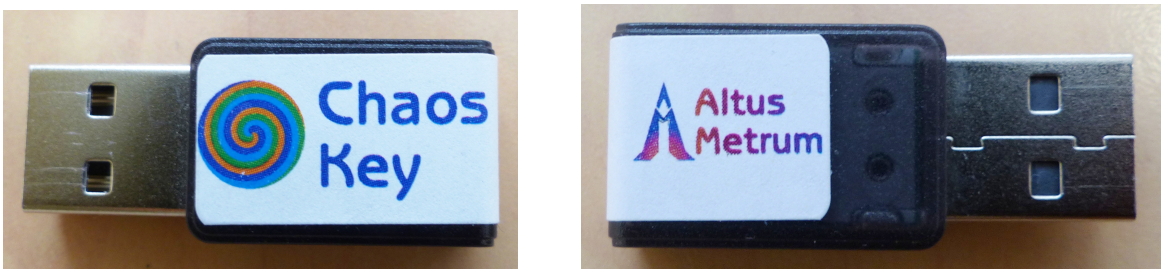


Figure 6.1: Altus metrum "Chaos Key" (both sides). There is a red LED inside blinking from time to time.

The Altus Metrum company produces some devices for flying rocket models, and also the random bit generator device (Figure 6.1[2]) called *Chaos Key*, see `https://altusmetrum.org/ChaosKey/`. According to the company description, this generator uses STN32F042 System-on-Chip microprocessor, 20V noise source (transistor junction) and a operational amplifier OPS356 (Texas Instruments, 200MHz gain-bandwidth product); the hardware designs are available. The price is 49 euros.[3]

One may get both "raw" bits or "cooked" bits from the device. The raw data resembles normal distribution (with rather small variance). The raw bits are produces rather fast (about 5 mbit/sec). The whitening (producing "cooked" bits) happens in the firmware, see file `ao_tnrg_send.c`, function `ao_trng_get_cooked`, and the definition of inline function `ao_crc_in_32_out_16` in `ao_crc.h`. Two 16-bit samples from analog-digital converter are concatenated to get a 32-bit integer. This integer is used to compute a running CRC32 checksum (for a sequence of 32-bit integers obtained so far). This checksum is a 32-bit integer; two halves of it are `xor`-ed to get a 16-bit integer that is sent to the output.

[1] Warning: we trusted the technical claims of the device makers and did not try to check the internal structure of the devices.

[2] All photos are of the devices we actually tested.

[3] The prices given do not always include shipping/handling/taxes/currency conversion fees etc. and should be considered only as rough estimates for the price range.

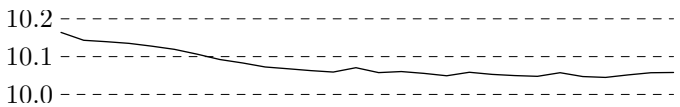


Figure 6.2: A long data file from one `chaoskey` run in a raw mode was split into 10Mbyte chunks. Each of 28 chunks was cut into 16-bit blocks and the entropy of corresponding distribution on $\mathbb{B}^{16}$ was computed. As we see, it initially decreases with time and then oscillates above 10 bits (out of 16). Not a surprise, since the parameters of Zener diode noise are not very stable and may differ for different units from the same roll.

Therefore, there is a $2:1$ ratio between used raw noise bits and output bits. It looks acceptable, since the entropy for raw stream shows about 10 bits for distribution of 16-bit integers (Figure 6.2), but the safety margin is not that large. The author of the firmware, Keith Packard, wrote (personal communication) that "The CRC computation is actually done in hardware, which is why I selected it. CRC32 isn't a proven whitening algorithm, but it passes all of the tests I have, given the output of the noise source", and our (limited) testing was consistent with this observation: nothing suspicious was found by tests of Section 7.3 for a 10 Gb sample.

The device is supported in the linux kernel (a driver is provided that uses the random bits from the device for system purposes).

6.2 Araneus

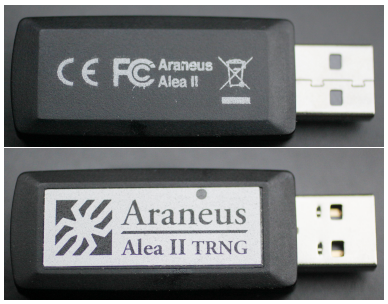


Figure 6.3: Araneus Alea II True Random Number Generator (both sides). The LED is visible near letter "e" in "Araneus".

Araneus Alea II device (Figure 6.3), produced by Araneus Information Systems Oy (`https://www.araneus.fi/en/`), according to the description in `https://www.araneus.fi/products/alea2/en/`, "uses a reverse biased semiconductor junction to generate wide-band Gaussian white noise. This noise is amplified and digitized using an analog-to-digital converter. The raw output bits from the A/D converter are then further processed by an embedded microprocessor to combine the entropy from multiple samples into each final output bit, resulting in a random bit stream that is practically free from bias and correlation." No specific information about this transformation and no way to get the raw bits is mentioned in the documentation. On request, the company was ready to provide the description of the whitening procedure under a non-disclosure agreement (we did not ask for it).

Device has a green LED that (according to the documentation) indicates that the device is functioning correctly. The CD with documentation, drivers and information needed to use the device with Linux, Windows and NetBSD is provided.

Speed: about 100 kbits per second. Price: 109 euros.

Health check: nothing suspicious found in our (limited) testing using tests of Section 7.3 with several 1 Gb samples.

6.3 Gniibe

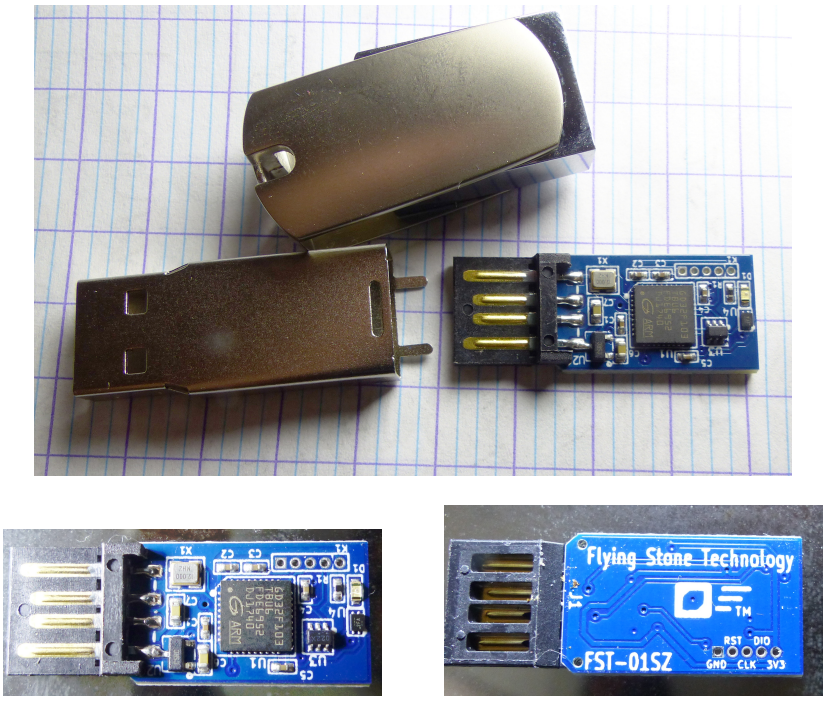


Figure 6.4: Gniibe random number generator, a firmware for FST-01SZ (both sides)

This generator (Figure 6.4), `https://www.gniibe.org/memo/development/gnuk/rng/neug.html`, is a firmware for the microcontroller usb stick, known as FST-01SZ, `https://www.gniibe.org/memo/development/fst-01/fst-01-revision-sz.html`. Both the firmware and the hardware are developed by Niibe Yutaka (see `https://www.gniibe.org/pdf/fosdem-2012/gnuk-fosdem-20120204.pdf`) as a tool for GNU project (Gnuk token and NeuG generator). The hardware does not include any special devices for generating random noise. Instead, it relies on the fluctuations in the voltages sent to the ADC convertors of the microcontroller. Then 128-bit string obtained from several measurement is converted to 32 bits by CRC32 function (intermediate source of entropy), and groups of 35 strings of length 32 obtained in this way (1120 bits in total) are sent to the final conditioning. At that stage, SHA-256 is used to produce 256 output bits out of the 1120 bits received. Half of these output bits are then concatenated with the input of the SHA-256 for the next step (so the total length of the input is $1120 + 128 = 1348$ bits).

The design of the entropy source makes it vulnerable to the environment, so we only could make some rough estimates. The device allows to get the primary data from ADC conversion, the result of CRC32, and the final result of the SHA-256 whitening. The Shannon entropy of the eight 16-bit primary sources in our experiment was about 6.8 bits for four of them and about 2.5 bits for four others, that gives the estimate 37.2 entropy bits (per 128 raw output bits), which is slightly more that required minimum of 32 bits for CRC32-output (but these sources may be dependent, and in other environment, say, with better power source, the entropy could be much less, so it is dangerously close to the minimum).

Tested with Linux ubuntu 18.04, commands are sent by `stty`, bits are obtained by `dd` from the device, no special software needed.

Speed: about 3 megabits per second (in all the regimes).

Price: around $50 at Free Software foundation shop, `https://shop.fsf.org/storage-devices/neug-usb-true-random-number-generator`.

Health check: no visible irregularities both in the cryptographically whitened and intermediate (only CRC32 used) streams in our limited testing (100 Mb samples, tests described in Section 7.3).

6.4 Tectrolabs SwiftRNG

Figure 6.5: Tectrolabs Swift RNG, LE version

This generator is developed by Tectrolabs company (`https://tectrolabs.com`) and exists in several versions; the cheapest one (LE, see Figure 6.5) is rather fast (20 mbits/sec) and costs $149. The documentation (`https://tectrolabs.com/swiftrng-le/`) says:

> Each of the two random noise sources of the SwiftRNG LE are analog electronic circuits that generate random electronic signals from reverse-biased Zener diodes using avalanche effect. The electrical noise generated by each random source is independently amplified and converted into digital values as bytes. The bytes produced are combined to produce a low bias random byte stream. To further reduce bias, the resulting byte stream is processed by a conditioning component with conditioning functions such as SHA256 or SHA512 (Marsaglia's XorShift64 is available when used with SwiftRNG LE device versions 1.2 and up)."

Here XorShift64 refers to a function that transforms 64-bit string in a reversible way: three linear transformations (over $\mathbb{Z}_2$) are made sequentially, and then the result is multiplied by some invertible constant modulo 2^{64}. This transformation was suggested by Marsaglia who used its iterations to generate a pseudorandom bit sequence, see `https://en.wikipedia.org/wiki/Xorshift` and `https://www.jstatsoft.org/article/view/v008i14` for details. The conditioning algorithm applies this transformation to 64-bit blocks of random bits (independently for each block).

The company documentation (`https://tectrolabs.com/2018/02/18/in-depth-how-swiftrng-generates-true-random-numbers/`) provides the following diagram explaining the details of random bits production (Figure 6.6). The CSD block in this figure is a "continuous self-diagnostics component ⟨...⟩ which validates the random values. The component performes frequency analysis, and checks for deteriorations or malfunctions in the noise source. The result of this analysis is reported in the overall generator status code."

Andrian Belinski (the founder of Tectrolabs) kindly provided more details about the internals of the device we bought. As he explained (personal communication), the SwiftRNG LE generator uses two Zener diodes that work in the avalanche mode (the voltage exceeds 5.1 V) in the following way. Each of the diodes produces a random noise that is (after DC correction) sent to the 12 bit ADC converter. Then the eight least significant bits of the ADC output are used and the four most

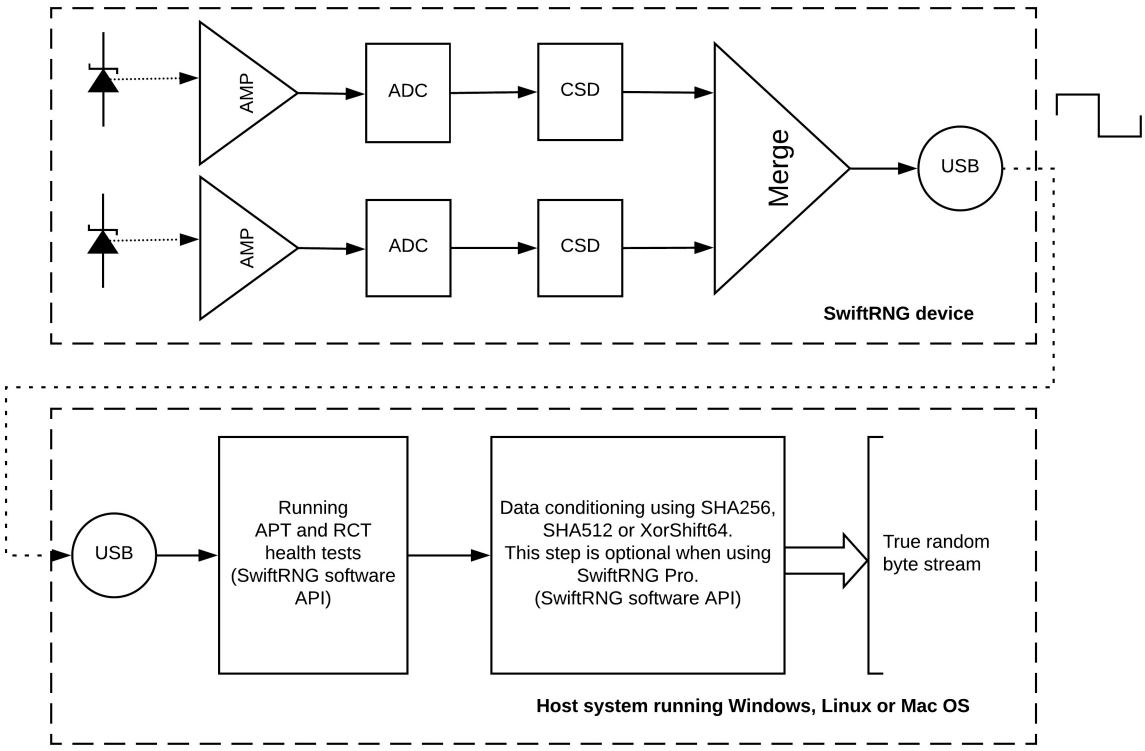


Figure 6.6: The structure of Tectrolabs Swift RNG devices. Upper block shows the device; the lower block represents processing in the host computer

significant bits are ignored[4]. Then the two bytes obtained in this way are `xor`ed to get an output byte of the device ("the low bias random byte stream" from the description above).

The provided software includes a kernel module (`swrandom`) that makes the output of the generator accesible to the Linux kernel, and a standalone utility (`swrng`) to get random bits. Both tools include the postprocessing algorithm (so the postprocessing is performed in the host computer, not in the device itself). There is also a utility (`swrawrandom`) that allows to collect raw noise bytes separately from both noise sources. The testing shows that the raw bits are not really random, but the non-randomness here is probably more subtle than in gaussian-like noise of other raw generators — the removal of the four most significant bits helps (see Figure 6.7). The `xor`ed bits do not show any problems in our testing (several 10 Gb samples).

The most advanced SwiftRNG Pro version of the device (`https://tectrolabs.com/swiftrng-pro/`, $449, Figure 6.8) is much faster, being the fastest device among the ones we tested. The documentation says that "SwiftRNG Pro is a general purpose USB device that generates true (hardware) random numbers at a rate of 200 Mbits per second without applying conditioning algorithms". Still "SHA-256, SHA-512, Marsaglia's XorShift64" are mentioned as "post processing algorithms available" (optional); the documentation contains test results for several tests but does not specify whether raw or processed bits were tested.

Andrian Belinski kindly provided additional information:

> SwiftRNG Pro processes data at faster speed and uses two noise sources (avalanche effect) and two High Speed ADC channels that sample noise sources at 40 Msps [millions of samples per second] rate. Similar to SwiftRNG and SwiftRNG LE, the Pro device also

[4]In other words, the ADC output is considered modulo 256. In this way the distribution of the noise, which should be close to a normal one, is split into 16 intervals that are then combined (the distribution functions added); this combination should have much more uniform distribution.

```
sts_serial:
rtest -x -f raw0 -e ../bitbabbler/raw_4_sources-1 -p 100 -q 100 -n 1200 -t 3 -d 36 -r 1
0.0782211579784 0.2111700862513 0.7020569828665 0.7020569828665 0.1111952605383
0.2111700862513 0.1548386665118 0.8154147124661 0.9084105017745 0.3681877860629
0.5830090612540 0.5830090612540 0.0782211579784 0.3681877860629 0.9684099261397
0.9996892272703 0.3681877860629 0.2819416298082 0.7020569828665 0.9684099261397
0.7020569828665 0.9684099261397 0.2819416298082 0.2111700862513 0.3681877860629
0.7020569828665 0.4695064485038 0.9084105017745 0.3681877860629 0.9942356257695
0.7020569828665 0.9084105017745 0.4695064485038 0.7020569828665 0.0240558028411
0.9942356257695

sts_serial:
rtest -x -f raw0 -e ../bitbabbler/raw_4_sources-1 -p 200 -q 200 -n 6000 -t 3 -d 45 -r 1
0.7125821300149 0.3281154440942 0.3281154440942 0.7125821300149 0.1779335278829
0.0878382826713 0.1122836028677 0.0085394839498 0.0060944182588 0.0006486080424
0.0002829222009 0.0085394839498 0.0395620258490 0.3281154440942 0.9646522162314
0.0000000000595 0.6284308022715 0.7125821300149 0.9877895282791 0.0521390871888
0.0680192046112 0.0395620258490 0.0060944182588 0.0060944182588 0.0000040248065
0.0297148244708 0.0162584592768 0.7933622419383 0.2704743832804 0.3935274335772
0.2704743832804 0.6284308022715 0.1420746514552 0.9973277498238 0.0395620258490
0.1779335278829 0.6284308022715 0.3935274335772 0.1122836028677 0.0118434497601
0.9238374197331 0.4662863907356 0.3281154440942 0.3281154440942 0.2205412175804

sts_serial:
rtest -x -f raw0 -e ../bitbabbler/raw_4_sources-1 -p 500 -q 500 -n 10000 -t 3 -d 48 -r 1
0.0000252154457 0.0007045191904 0.0007045191904 0.0000136717195 0.0000000565349
0.0000000179615 0.0000019751295 0.0000019751295 0.0000186051465 0.0001434246304
0.0018769751767 0.0018769751767 0.0163791281863 0.0023738215454 0.0198941100724
0.0240661470524 0.0000000000000 0.0240661470524 0.0134308881556 0.0000052932652
0.0000003518372 0.0000340349325 0.0058320825331 0.0240661470524 0.0007045191904
0.1725563396262 0.2263472905139 0.0954645323116 0.5600220048872 0.1980421265871
0.0586892094174 0.1293961699671 0.2263472905139 0.0693007656927 0.0586892094174
0.0000186051465 0.0058320825331 0.1980421265871 0.9351121874343 0.6659218712647
0.0586892094174 0.5089169658043 0.5089169658043 0.3293581065813 0.9603008958861
0.2919248807418 0.7704365372945 0.0815016732111
```

Figure 6.7: Testing Swift LE device. Bits from one of the generators look more or less OK for small samples but show more problems as the sample size grows. Here the results of `sts_serial` test (1 Mb, 10 Mb, 100 Mb samples) are shown.

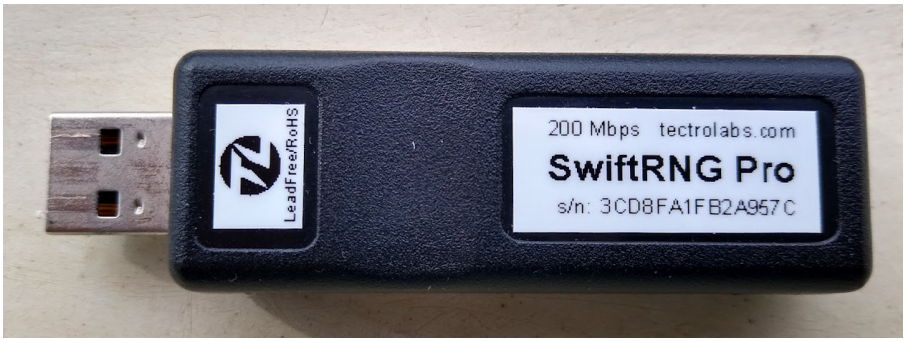


Figure 6.8: Tectrolabs Swift RNG, PRO version

```
ent_8_16:
rtest -x -f raw0 -e ../../bitbabbler/raw_4_sources-1 -p 1000 -q 1000 -n 10000 -t 9 -d 4 -r 1
0.0023934096489 0.0028179208243 0.4006338815833 0.2877976434847
Used 80016000 bytes from the test generator

ent_8_16:
rtest -x -f raw0 -e ../../bitbabbler/raw_4_sources-1 -p 500 -q 500 -n 200000 -t 9 -d 4 -r 1
0.0000000000000 0.0000000000000 0.0163791281863 0.0289959610611
Used 800008000 bytes from the test generator
```

Figure 6.9: Entropy tests for SwiftRNG pro bit generators (bits taken from one generator) with different parameters.

> reduces the ADC output bits to less significant 8 bits. Two bytes from one channel are combined with 2 bytes from the other channel using several XOR, OR and Shift logical operations, resulting in a final output stream of bytes. I was able to run multiple statistical tests on several terabytes retrieved from SwiftRNG Pro and obtained good results. That's all I can tell about SwiftRNG Pro. ⟨...⟩
>
> When in 'swrawrandom' mode, both noise sources are getting sampled with 40 Mbps each (simultaneously), however only one noise source is used and all the bytes collected from the other noise source are discarded/skipped. In that mode, each 16,000 byte block downloaded is guaranteed to contain a continuous sequence of random bytes collected from a single noise source.

The raw bits from one generator start to look suspicious for 100 Mb samples, and obviously fail the tests for 1 Gb sample (Figure 6.9). The bits obtained in the normal regime (as described in the quote) do not show anything suspicious in our limited testing (few samples of 10 Gb size were used). The same is true for bits obtained after post-processing (SHA256).

There is also a version of the device (SwiftRNG Z) that uses the breakdown mode Zener diodes (4.6 V voltage) and another postprocessing algorithm called "linear corrector" that is implemented in the device itself. This algorithm applies a non-invertible linear transformation that decreases the number of bits by factor 2 as suggested by P. Lacharne [27].[5] We did not test this model.

[5]In this paper the following observation is made. Consider a $n \times N$-matrix that transforms N-bit column vector to a n-bit one. Assume that we apply this matrix to N independent copies of a biased coin with bias ε (probabilities $1/2 \pm \varepsilon$). We get n bits; we want all linear combinations of those bits to have a small bias. Any linear combination of the output bits is a linear function of the input bits. To find out which input bits appear in this combination, we need to sum up some rows of the matrix. If the matrix corresponds to an error-correction code with distance d, the sum of its rows has

The documentation for SwiftRNG devices mentions "patented technology": Andrian Belinski has a patent [2] that describes a method of random bit generation. The general scheme described in the patent, in additional to Zener noise source and analog to digital conversion, includes high-pass filter (that is called "low-pass filter" in the patent, probably an error)[6]. The idea is that this high-pass filter decreases what the author calls "bias" in the random bytes (probably bias is understood here as the dependence between neighbor bytes). This is true to some extent, but there is no reason to expect to get the exactly uniform distribution after filtering. Moreover, if the original signal was perfectly random, the filtered one would be different.

6.5 Moonbase OneRNG

This random generator was developed in 2014–2017 by Paul Campbell and is available commercially from `https://moonbase-otago.myshopify.com/products/onerng-external` for $40. It is an open hardware/software device, see the description and links at `http://www.moonbaseotago.com/onerng`. The hardware contains two primary sources of randomness: the Zener diode and the radio frequency receiver (strictly speaking, it is not a source of randomness but the way to get randomness from the environment). The device is visible through standard USB interface `/dev/ttyACM`. It can be asked to produce raw bits from both sources, or whitened bits ("we have a CRC16 generator to use for whitening, when it's enabled we push each byte into the generator and extract the upper byte of the running CRC sum – this means that the bits of adjacent bytes are merged and xored together evening out the number of ones and zeros", `http://www.moonbaseotago.com/onerng/theory.html`). The processing is performed by Texas Instruments CC2531 system-on-chip.

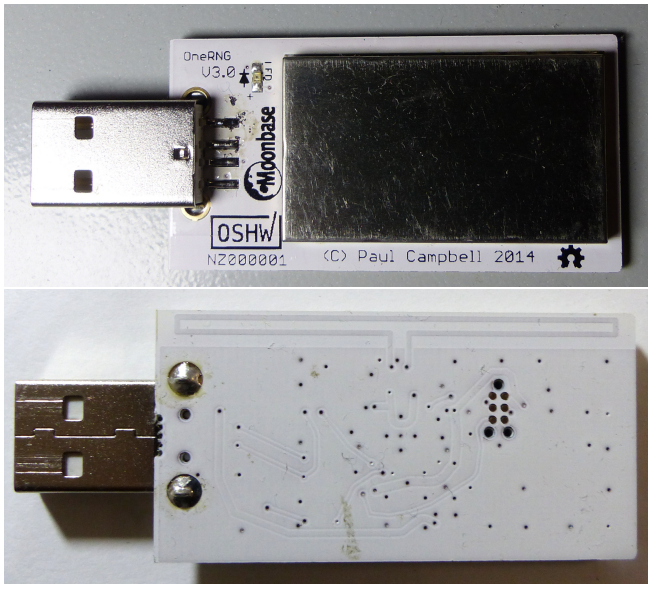


Figure 6.10: Moonbase OneRNG, version 3.0

The initial testing shows that the raw bits from the Zener noise source are visibly non-random (Figure 6.11, left), and the whitened bits are also definitely non-random (though in a bit different way,

at least d bits 1, so at least d input bits are `xor`ed to produce the resulting bit, and it is easy to see that the bias is $O(\varepsilon^d)$, and it is good to have large d. Still this argument is applicable only to independent input bits, and it is not clear whether it make sense for the output of an analog-to-digital conversion.

[6]The scheme in the patent contains four noise sources, but it is noted that one can use any other number of noise sources.

Figure 6.11, right). This is also obvious with tests[7] (first of all `opso` and `oqso`). The documentation says that the whitening procedure produces 8 output bits for each 8 input bits, so it cannot increase the entropy of the source, and this could be one of the possible reasons for the visible deficiencies in the whitened bits. (A similar picture for 8-bit bytes instead of 16-bit integers does not show clearly visible anomalies.)

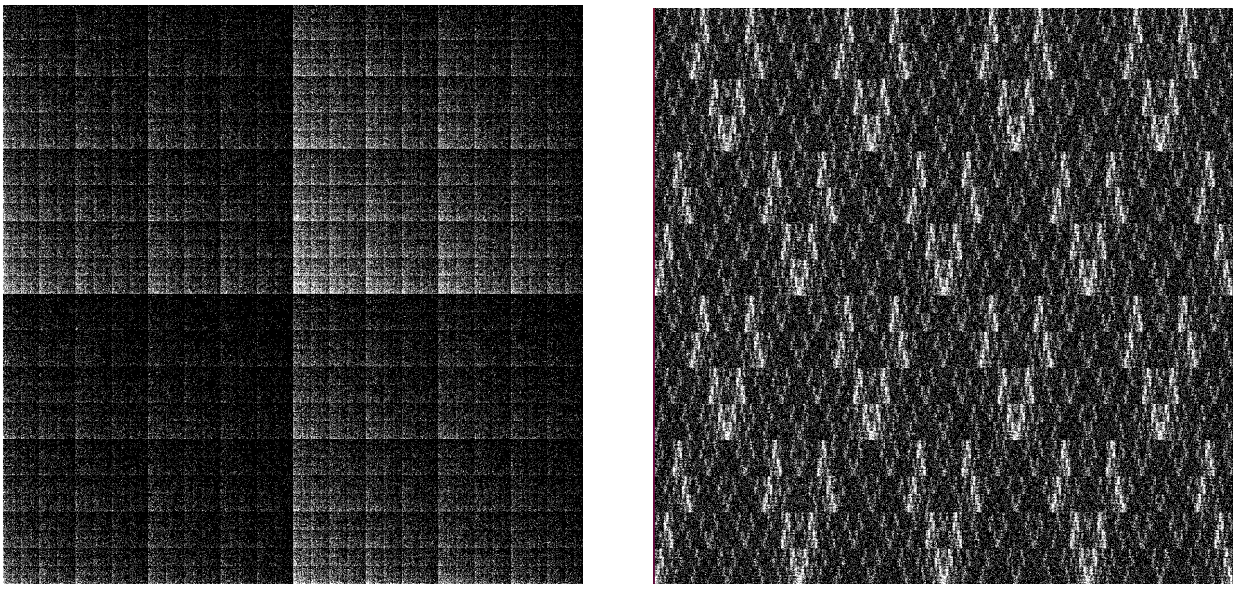


Figure 6.11: Moonbase OneRNG, raw bits from Zener source (left), processed version (right). Each 32 bits are converted to two 16-bit signed integers, and the corresponding point on a coordinate plane is displayed

The random bits extracted from the environmental RF noise (both raw and whitened) looked much better in our experiments; our limited testing did now show obvious problems. However, we did not try to isolate the circuit from the environmental noise and did not repeat the experiment in different environments, so one should not rely on these observations.

There are two more regimes where Zener diode noise and RF noise are interleaved; the result also is visibly non-random and does not pass the tests (neither in raw nor in whitened version).

For some results obtained by spectral tests see Figure 7.4 in the next chapter.

6.6 ID Quantique

The company ID Quantique (`www.idquantique.com`) produces several quite expensive random number generators. The least expensive one is a USB device that produces random bits with speed about 3.2 megabits per second (Figure 6.12).

The document *Redefining randomness. Random number generation. White paper. What is the Q in QRNG?* (ID QUANTIQUE SA, January 2019), available from the company, say that this device uses the "Quantum Random Number Generator, *Quantis*" that is "is available as a component, in the form of a compact metal package that can be mounted on plastic circuit boards". There are USB and PCI versions of the device using one *Quantis*, and more advanced PCI version that uses four of them (and is four times faster). The *Quantis* "contains the optical elements that are used to implement the random process and produce the random outcomes. It comprises a light emitting

[7]Let us mention that the graphs show obvious non-randomess for much smaller samples than p-value tests: 100 Kbytes are enough to see obvious non-randomness, while our tests for 1 Mb sample do not exhibit obvious problem and one should consider samples of 10–100 Mbytes.

Figure 6.12: Random number generator from IDQuantique, USB version

diode producing the photons, a transmission element, where the random process takes place, and two single-photon detectors — detectors with single-photon resolution — to record the outcomes." The transmission element is a mirror where (according to quantum physics) a photon either gets through or is reflected, with equal probabilities. This makes the device radically different from most of the devices that use some kind of random noise, since it is based on an individual and well defined quantum event.

Still, as the document explains, some whitening is needed: "As mentioned above, physical processes are difficult to precisely balance. It is thus difficult to guarantee that the probability of recording a 0, respectively a 1, is exactly equal to 50%. With Quantis, the difference between these two probabilities is smaller than 10% — or equivalently the probabilities are comprised between 45% and 55%. As this bias may not be acceptable in certain applications, the processing unit of Quantis

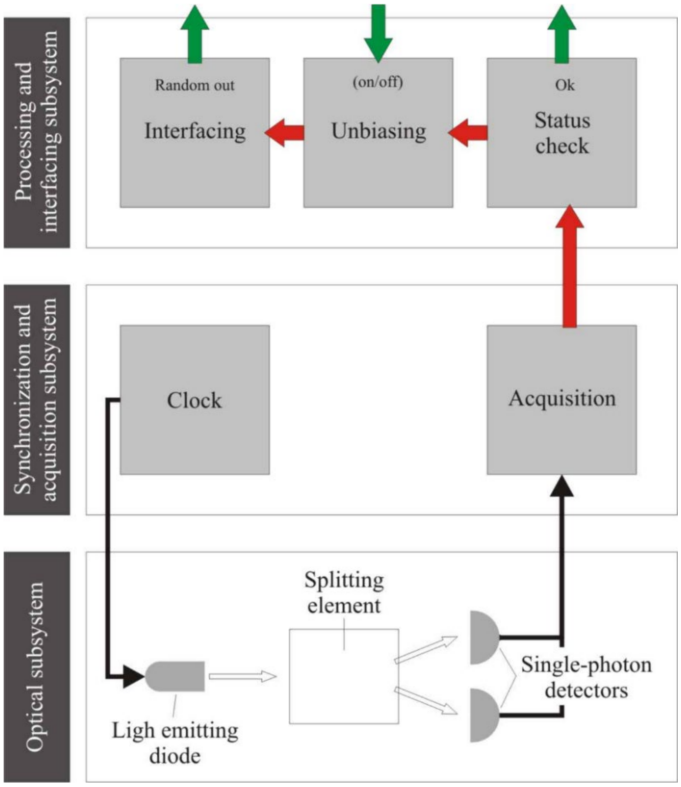


Figure 6.13: The structure of *Quantis* according to the company's description

performs unbiasing of the sequence." This "unbiasing" block can be seen in the device description (Figure 6.13, taken from the documentation provided by ID Quantique).

Unfortunately, the exact nature of this "unbiasing" is not specified. The section 3.6 of the white paper describes the von Neumann approach (splitting the sequence into 2-bit blocks, replacing 01 by 0 and 10 by 1 while ignoring 00 and 11 blocks) as one of the possible "unbiasing" procedures, and then says "Other unbiasing procedures exist. The one proposed by Peres[8], for example, is significantly more efficient than the Von Neumann procedure." The scheme also indicated that this unbiasing may be turned off, but the documentation does not explain how to do this. We have asked the IDQuantique support (for that you need to purchase a separate support contract for 99 euro) and they confirmed that this is not possible, via the following mail exchange:

> *Our questions, e-mail, January 27, 2020*:
>
> 1. The documentation mentions two kind of postprocessing. First is called unbiasing in "Redefining randomness. Random number generation. White paper. What is the Q in QRNG?", section 3.6. The second in called randomness extraction and is described in "Technical Paper on Randomness Extraction, version 1.0, September 2012". Do I understand correctly that these two are different procedures, and one is performed internally in Quantis all the time, while the second is optional and is implemented in the software library (using multiplication by a constant matrix provided)?
>
> 2. There are a lot of certificates for Quantis in Quantis Certification Collection. Do I understand correctly that these certificates are for the normal output of Quantis, not

[8]Probably this refers to [44]. As it is done in the von Neumann approach, Peres assumes the Bernoulli distribution with constant probability, a rather strong assumption that is hardly realistic for a physical device.

for the results of randomness extraction procedures? (There is one certificate for AIS.31 version of the device, that we do not have, so I am asking about other certificates.)

3. Is there any documentation describing the exact unbiasing procedure as implemented in Quantis? The section 3.6 mentions von Neumann algorithm and Peres' method, but does not say which method is used.

4. The diagram on p. 11 of the "Redefining randomness" white paper indicates that the unbiasing may be turned on and off (Figure 2, middle green arrow going down [Figure 6.13]). Is there some way for the user to switch it temporarily off (for testing purposes)?

The reply of Jean-Benoit Page (ID Quantique), e-mail, March 18, 2020:

Please find the answers to your questions:

1. Yes[,] your understanding is correct.

2. Yes[,] the certificates rely [refer] to the standard Quantis (non AIS-31) without extraction post processing.

3. Sorry, we don't share more information on this topic. All public information is available on our website.

4. No[,] there [is] no way to disable the unbiasing. This is an error on the diagram.

The nature of "unbiasing" is discussed in questions 3 and 4. The questions 1 and 2 refer to another whitening procedure, called "randomness extraction". It is explained in *ID Quantique White Paper, Randomness Extraction for the Quantis True Random Number Generator*, (version 1.0, September 2012) as follows:

> Quantis is a True Random Number Generator which exploits quantum physics to produce random bits. It is a perfect entropy source based on the reflection or transmission of a light particle — a photon — on a semi-transparent mirror. However, like any other physical randomness source, Quantis produces strings of bits which are slightly longer than their entropy. This excess length is minute and is in most cases not noticeable, as extremely long sequences must be considered. In addition, these extra bit symbols can be removed using a so called Randomness Extractor. This white paper describes the principle of randomness extraction and explains how it is implemented in the Quantis software package.

The text then explains again the bias problem, mentioning also correlations:

> No matter how elegant its principle is and how carefully designed a TRNG is, a particular physical realization will by definition have imperfections, which can in turn impact the quality of its output. The random sequence can for example exhibit a bias — unequal probability of generating a 0 and a 1 — or correlations — probability dependant on previous bit values. In spite of its qualities, the Quantis TRNG exhibits some — very low — residual bias and correlations. The bias is caused by differences in photon detection efficiency between the detectors and correlations arise from memory effects in these components. (Footnote: The Quantis TRNG uses avalanche photodiodes for photon detection. These components are optolectronic devices which can transform photons into macroscopic electronic pulses.)

Then the von Neumann unbiasing algorithm is explained again; the authors of the document claim that "the von Neumann unbiasing algorithm is an example of a so-called deterministic randomness extractor" (rather strange claim). However, as they explain, the software package provided by the

company uses "non-deterministic extractors", because "the prime advantage of using a randomness extractor is that it allows [us] to mathematically prove the quality of the output random sequence and base this proof on the well established information theory".

Actually this "non-deterministic extractor" is just multiplication by a 1024×768 bit matrix: in this way 1024 bits from *Quantis* (already "debiased" in some unknown way) are transformed into 768 output bits. The matrix is in fact fixed (provided by the software library); however, the paper is not very clear about it:

> The generation of the matrix M is crucial to guarantee that the properties of the output of the extractor meet the requirements in terms of randomness quality. Each position in the M matrix should consist of an unbiased random bit, completely independent of other bits. Note that the matrix can be the same for multiple Quantis TRNG and can be reused. It must thus be generated only once. ⟨...⟩ [The] compression factor has been selected to guarantee that the probability for the extracted sequence to deviate from a perfectly random sequence is less than 2^{-100}. This implies that even if he had millions of Quantis devices, a user will not see any deviation from perfect randomness in a time longer than the age of the universe.

This claim sounds rather confusing: the expression "the probability for the extracted sequence to deviate from a perfectly random sequence is less than 2^{-100}" has no meaning. In a more technical document, *ID Quantique. Technical Paper on Randomness Extraction, version 1.0, September 2012* (by M. Troyer and R. Renner), the authors explain that this bound is for the statistical distance between the uniform distribution and output distribution. However, the authors do not make a clear distinction between the (a) output distribution for two random inputs (the sequnce of "raw bits" and the uniformly distributed random matrix) and (b) the distribution for the outputs when the matrix is fixed. The bound they speak about is for (a). On the other hand, for the suggested use of the device (with fixed matrix) the distribution (b) matters, and the results about random extractors do *not* provide any guarantee for this distribution, as we have discussed above while speaking about Santha–Vazirani sources. We have also seen (while discussing strong extractors) that indeed there exists some justification for the use of fixed "randomly chosen" matrix. But this justification replaces the statistical distance $\varepsilon > 0$ provided by the strong extractor, by much bigger distance $\sqrt{\varepsilon}$ for the fixed second input (denoted by d in the discussion above), for most (up to error probability $\sqrt{\varepsilon}$) values of this second input, and there are nothing about that in the documentation.[9] This problem is not mentioned either in the extended version of this Technical Paper [13].

Another problem with this approach is that the strong extractors argument can be applied only to the distribution of 768 bits obtained by the linear transformation. However, in most applications we need more than 768 random bits, so we have to apply the same transformation to different 1024-bit strings from the generator. Even if each of them has distribution close to the uniform distribution on $\mathbb{B}^{768}$, the question about the possible dependence between them remains, and strong extractors argument does not help at all.

It is interesting to try different tests on ID Quantique product. The company provides a bunch of test reports, including tests from NIST [37], DIEHARD [29], CTL certification (that does not specify

[9] Unfortunately, the *Redefining randomness* paper contains some obviously wrong statements in its introduction part (though they are not directly related to the device). On p. 3, in a section "What is a random numbers", the authors wrote: "It is quite straightforward to define whether a sequence of infinite length is random or not. This sequence is random if the quantity of information it contains — in the sense of Shannon's information theory — is also infinite. In other words, it must not be possible for a computer program, whose length is finite, to produce this sequence. Interestingly, an infinite random sequence contains all possible finite sequences." There are several misunderstanding in these sentences. First, the Shannon information theory does not define "quantity of information" for individual binary sequences. The second sentence identifies randomness with non-computability, thus making the sequence where every second bit is zero and other bits form a non-computable sequence, random. Then the third statement becomes false, since this sequence does not contain the block 11.

which tests were used, and the link in the documentation does not work), iTech Labs (also DIEHARD tests), and AIS 20/AIS 31 certification according to [22]. The latter certification is for a special version of the device (that we had not bought), and classifies it as a PTG.3 class.[10]

Unfortunately, the documentation does not specifies whether these tests were performed for a bare Quantis device (after the internal unbiasing procedure) or on a sequence obtained by multiplication by a constant matrix ("randomness extraction"), and it is also not mentioned on the certificate sent with the device (Figure 6.14). The explanations given by the company (see the mail exchange above) clarify that the tests were performed for bare Quantis device. But in this case the test results say more about the quality of testing. Indeed, one can run standard tests (e.g., from `dieharder` suite) and immediately see the problems: several tests fail for the bare output of Quantis device (without randomness extraction stage).[11]

This observation was discussed by Darren Hurley-Smith and Julio Hernandez-Castro in [19, 20]. In the first paper the authors give the results of dieharder test for several samples obtained from different Quantis devices (PCI/USB, 4M/16M). However, as it is clear from this paper (and can be easily confirmed experimentally), different runs of dieharder tests (for different samples) may provide results that have not much in common (different number of WEAK/FAIL results, different tests that fail, etc.).[12] One can also recall that there are some problems with the tests and their implementation (as discussed above). They also provide the results of testing with `ent` tool (from ubuntu distribution) that show that χ^2-test gives bad results for Quantis. We have discussed these results and their rigorous version in Section 3.3. Figure 6.15 shows the output of the robust version of `sts_serial` test described in [37].

The second paper [20] discusses more extensive experiments (with more generators, and more tests, see Figure 6.16). It contains a lot of interesting observations, but also illustrate the methodological problems of randomness testing. Consider, for example, the last column. For all the generators more than 10% of sample fail the test. What does it mean? Is the test "oversensitive" in such a way that the truly random generator fails it with probability more than 10%? Or does it mean that all the generators are bad, just one needs a really good test like *Crush* to discover the problem?

The authors comment on this problem, but the comments seem to create more confusion: "...the common sense statement that good generators only fail overly complicated tests and bad ones only [sic!] fail tests that are simple enough to implement. This makes sense, as a generator will eventually fail any test if the test in question is sufficiently rigorous. As a result, this work focuses on generators that consistently fail so-called reasonable tests of randomness, rather than seeking specific failures that occur only in the most rigorous of tests" [20, section 3.2.4].

Another comment: "An important concept to bear in mind when considering these generators is the non-deterministic nature of the output. Solitary tests of individual sequences are not sufficient to identify issues; any non-deterministic bit generator (TRNG) will fail a given statistical test at some point due to the inability to guarantee that a given sequence will contain a uniform distribu-

[10]One of the requirements form PTG3 devices (3.6) says: "The algorithmic post-processing algorithm belongs to Class DRG.3 with cryptographic state transition function and cryptographic output function, and the output data rate of the post-processing algorithm shall not exceed its input data rate." It requires the use of one-way function, and none of the described operation for the standard version of Quantis and the accompanying software mention one-way functions. The special certified version is described in `https://www.idquantique.com/random-number-generation/products/quantis-ais-31/` where the company says: "The Quantis AIS31 RNG complies with all the requirements of the PTG.3 class. It provides cryptographic post-processing based on the Advanced Encryption Standard (AES) that offers an additional security anchor. The cryptographic post-processing required to achieve PTG.3 level compliance reduces the raw bit rate of 4Mbps to approximately 75Kbps". It is not clear why such a huge decrease in the rate is needed.

[11]This is acknowledged in the quote above from the company's documentation where the need for the randomness extraction stage is explaned.

[12]The same can be said about bias graphs on p. 14–16 of this paper. One should not assume that for perfect generator the graph should be flat with almost-zero bias; on the contrary, such a zero bias would be a clear indication of a bad generator. One should not make far-going conclusions from individual graphs, either: for different samples of the same device the graph could be significantly different, both for good and bad devices.

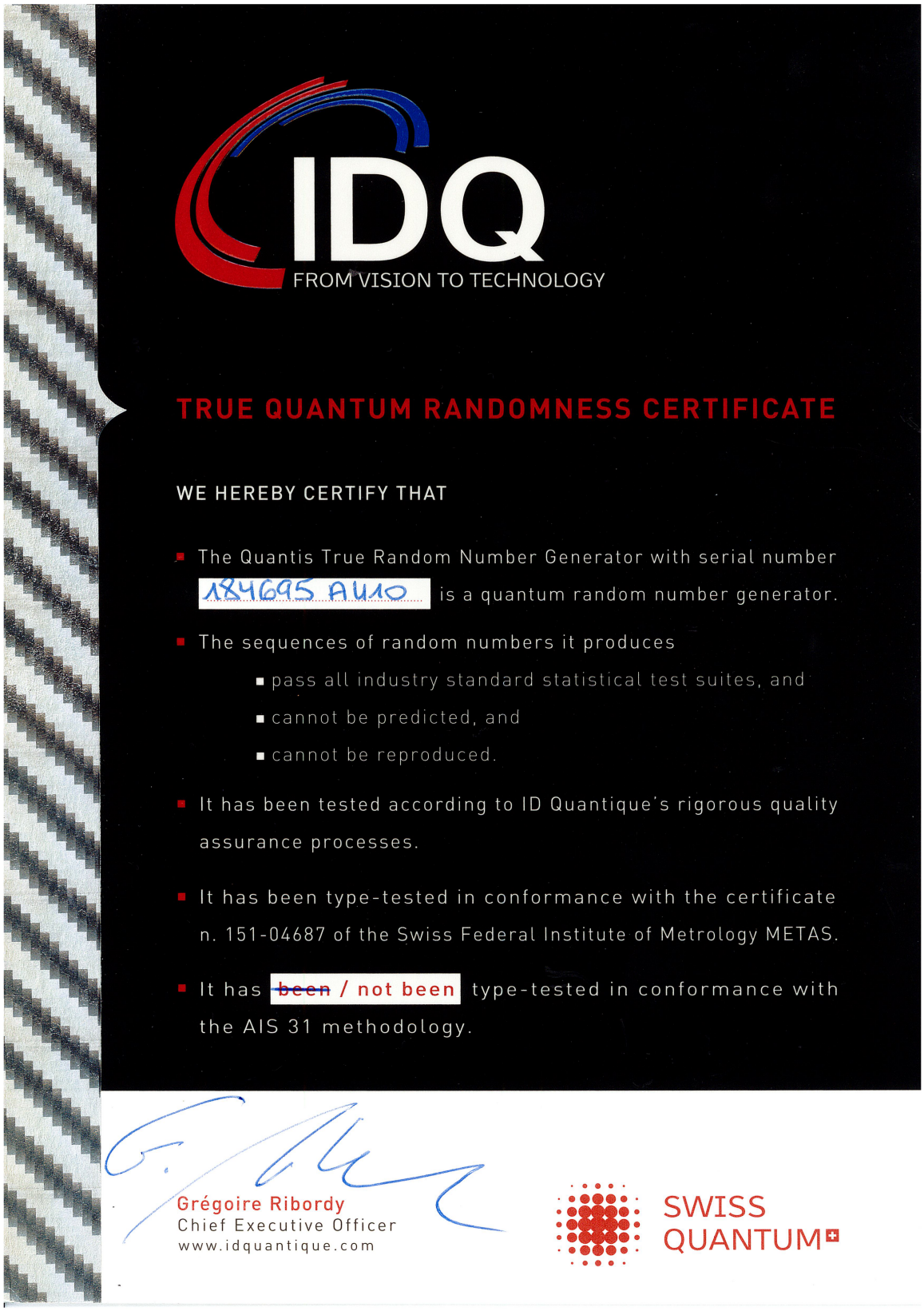


Figure 6.14: Certificate for ID Quantiue Quantis-USB-4M device

```
sts_serial:
rtest -x -f random11.dat -e ../bitbabbler/raw_4_sources-1 -p 1000 -q 1000
-n 100000 -t 3 -d 57 -r 1 -k
0.0691762539902 0.0000000001258 0.0000000000923 0.0000002461977 0.0000013787446
0.0028179208243 0.0071953614430 0.0129123522308 0.1641144775643 0.9690010280431
0.8282194040312 0.5362766985932 0.4326088695815 0.3136800387321 0.3701201760617
0.2634717271986 0.3411247511266 0.2406820248660 0.4326088695815 0.1641144775643
0.0000000000007 0.0000000018564 0.0000598456772 0.0971034965705 0.5005673707894
0.2406820248660 0.5005673707894 0.3136800387321 0.1483645207896 0.6854967337921
0.7593695685683 0.7946637387577 0.7593695685683 0.7228251828701 0.2406820248660
0.9136894237272 0.3701201760617 0.6854967337921 0.6101664688189 0.0000000000000
0.6478148720772 0.5005673707894 0.7593695685683 0.4659595288557 0.3136800387321
0.4659595288557 0.3136800387321 0.4326088695815 0.5005673707894 0.6478148720772
0.9542189106779 0.9883339509983 0.7593695685683 0.2406820248660 0.9690010280431
0.4659595288557 0.5362766985932
Used 800016000 bytes from the test generator [0]
Used 400008000 bytes from the etalon generator [1]
```

Figure 6.15: One of the robust tests (`sts_serial`) applied to the output of Quantis device without multiplication by 1024×768 matrix (file `random11.dat` mentioned in the command line). Other generator (`bitbabbler`, Section 6.8) is used as reference. (Recall that we do not need any assumptions about this reference generator to make these p-values valid.) The results of the postprocessing (matrix multiplication) did not exhibit any anomalies in our (limited) testing.

tion of values (be they bytes, integers or sub-sequences of arbitrary length). This is a key difference between uniform deterministic (PRNG) and non-deterministic bit generators (TRNG): the former ensure a uniform distribution, while the latter rely on their inherent entropy to provide truly random values. A uniform distribution is likely over a given set of sequences, but an individual sequence may exhibit deviations which will cause some statistical tests to fail. Rigorous testing of statistically significant data sets is required to reliably identify recurrent issues from probabilistic phenomena." [20, section 2.3]. It is hard to understand what the authors have in mind when speaking about uniform deterministic PRNG that ensure a uniform distribution.

Also recall that the `dieharder` test used in [20] performs an incorrect computation of Kolmogorov–Smirnov deviation and has other problems (as discussed in Section 2.3.2).[13]

Returning to the general general discussion of quantum bit generators, one may note (as the producer of Quantis do) that from the viewpoint of physics there is a difference between a "random noise" that comes from statistical mechanics (say, the thermal noise in a resistor, or the Brownian motion, or some chaotic dynamical systems with external noise, like Lorentz attractors) and more "refined" randomness that comes out of quantum mechanical systems where the events related to individual microscopic objects can be observed, for example, experiments with individual photons. However, one can argue that

- there is no clear distinction between two categories: how do we classify Geiger counter events for a macroscopic piece of slightly radioactive material? how do we classify the noise in a PN junction (definitely related to some quantum effects but in multi-particle systems)?
- from the practical viewpoint, it is doubtful that one can construct an experimental device that is "clean" enough to avoid the conditioning step. And if we use conditioning, do we

[13]There is no need to say that all these problems are mentioned not to blame the authors of [20] who did a lot of important work and highlighted some intrinsic problems with quantum randomness generation, but to illustrate the common need for robust testing methodology.

Table 3. Dieharder, NIST SP800-22 and TestU01 Results

Device	Samples #	Dieharder Passed	NIST Passed	Alphabits Passed	Rabbit Passed	Small Crush Passed	Crush Passed
16M PCI-E	100	100	100	54	60	93	47
Post 16M PCI-E	100	100	100	95	87	91	82
4M PCI-E	100	100	100	3	7	91	3
Post 4M PCI-E	100	100	100	91	82	93	86
4M USB	100	100	100	3	21	89	3
Post 4M USB	100	100	100	90	81	97	80
Comscire PQ32MU	100	100	100	91	86	93	84
ANU QRNG Server	10	10	10	9	8	-	-
Humboldt Physik	10	10	10	10	8	7	10

Figure 6.16: Table from [20]. As the authors say, "Table 3 outlines the results generated using three well-known statistical test batteries, intended for use in ascertaining whether data is sufficiently random. Each set of results represents the number of passing samples for each test battery, out of a total number of samples (2nd column)".

> really have an advantage using a delicate quantum-mechanical experiment instead of cheaper alternatives? One can argue that it is better to have "true randomness" instead of "mere chaos", or something like this. It definitely sounds good for philosophers or for a sales brochure, but are there more essential advantages?

6.7 Infinite Noise

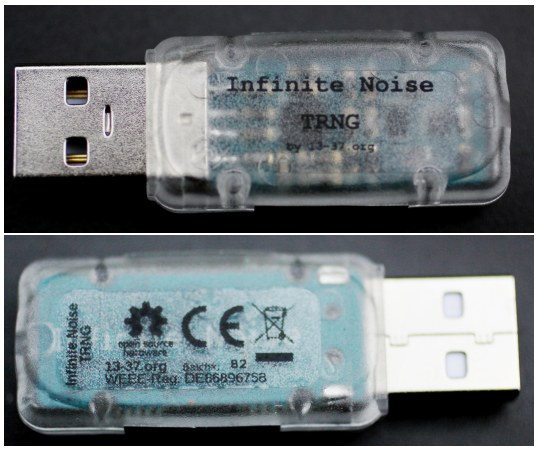


Figure 6.17: Infinite Noise generator

This generator (Figure 6.17) is available from `13-37.org` (`https://13-37.org/en/infinite-noise-trng/`) for 30 euro plus shipping. As the site says, "`13-37.org` electronics shop was founded in 2017 by Manuel J. Domke. We are primarily dedicated to manufacturing and direct sales of open-source hardware."

As Manuel writes in the blog notes, "The Infinite Noise TRNG is an open source random number generator. Check the GitHub repository from Bill Cox (waywardgeek) for detailed documentation of the hardware design. My contributions to the software side of this project are of course included in these builds. This means, `systemd`- and `udev` rules are included for automatic start of the driver

when connecting the device. Also its now possible to connect multiple devices to one computer (mainly for testing purposes)."

So both the hardware and software of this device are open-source (available from `github`), a big plus, especially for (rightfully) paranoid people.

Omitting technical details, its functioning can be described as follows. Consider the bit shift map $[0,1] \to [0,1]$ defined as follows:

$$t \mapsto 2t \bmod 1, \quad \text{i.e.,} \quad t \mapsto \texttt{ if } 2t > 1 \texttt{ then } 2t-1 \texttt{ else } 2t.$$

In terms of bit representation it shifts a binary number to the left, deleting the first bit (therefore the name). It can be easily implement if reals are interpreted as voltages, using operational amplifier and comparator. Using the clock signal and a sample and hold devices, we get a circuit where at every clock pulse the voltage (kept in a capacitor) is subjected to the bit shift transformation.

Applying a bit shift map to some real $x \in [0,1]$ iteratively, we get its bits from left to right (assuming that the computation is performed exactly). These bits can be read at the output of the comparator. However, there is always some noise in the circuit, so actually the output will be determined by this noise starting from some point, and not by the initial voltage. And for random bits generator it is a feature, not a bug. This method of capturing noise has some advantages over the digitizing the amplified noise (say, Zener diode noise). First, we do not need the analog-digital converter, the scheme uses only simpler devices. Second, the internal noise is combined with external interference in a better way. If, say, some traces of 50 Hz AC are added to the Zener diode signal (that is rather weak by itself), then the resulting output could lose almost all randomness, since the noise could be masked by the parasite signal. On the other hand, in this scheme the interfering signal will affect all the stages, and none of them involve amplification by a large factor.

An additional (and very important) bonus is the possibility of the verification of the signal, if we change the design a bit and use a smaller factor than 2 in the map. Then we cannot say anymore that the device reads the sequential bits, it does something more complicated. The resulting bits (more precisely, their groups, like bytes or integers of given length) have a kind of Cantor-set structure. It is shown in Figure 6.18 (left) that was obtained as follows: the raw bit stream from the device is split into 16-bit integers and the pairs of these integers are treated as point coordinates. For comparison we provide the simulation picture (right) obtained as described above (with coefficient about 1.9). In fact, the debug option of the provided software estimates this coefficient, so there is no need to tune it by hand trying to match the device behavior. (The closer is the factor to 2, the smaller are gaps in the simulation picture.)

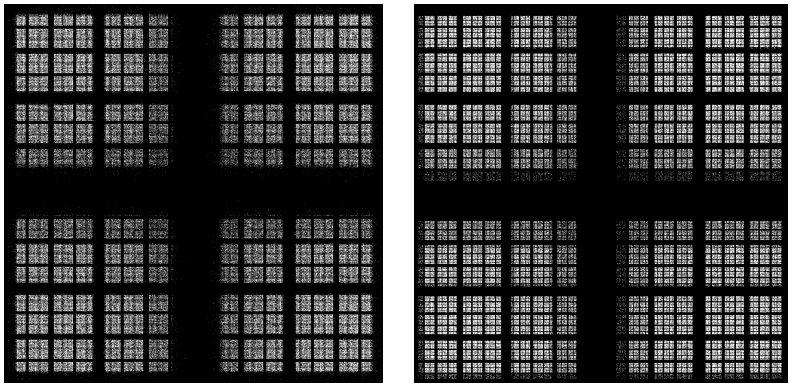


Figure 6.18: Pairs of consequtive 16-bit raw integers from Infinite Noise generator (left) and their simulation (right).

This approach (decreasing the amplification) makes easy to verify the design. However, the resulting bits are obviously non-random (and the entropy of the output distribution is slightly less

than 1, about $\log k$ where k is the amplification factor). Trying dieharder with some reasonable parameters, we see that most of the tests fail. So some whitening is definitely needed.

The software driver provided by the company performs whitening based on the SHA3 hash function. Though it is a standard approach (recommended by NIST), and could be considered as reasonably safe if health monitoring is used (it is provided by the drivers), still the paranoid people could say that there are no mathematical reasons behind this approach (and, may be, replace this approach by some other type of whitening, e.g., taking `xor` of several devices of this type, or doing something else).

The raw speed (for the device that we bought) is ≈350 kbits per second; after whitening it decreases (as it should) to ≈300 kbits per second; the documentation says 32 kbytes per second for the whitened stream.

Health check: for the whitened stream basic tests (on 1 Gb sample) do not show anything suspicious.

6.8 BitBabbler

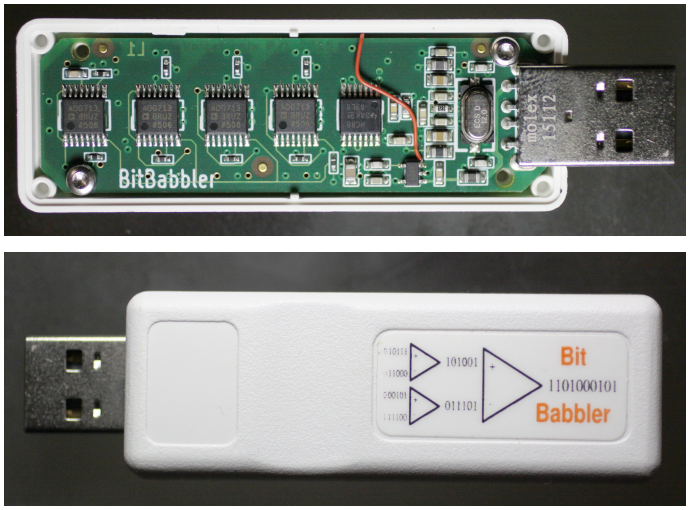


Figure 6.19: Bitbabbler generator (inside and outside)

This device (`http://www.bitbabbler.org/index.html`, Figure 6.19) is produced by a small Australian company *Voicetronix*. The Linux drivers, provided by the same company, are included in the standard distributions (we used ubuntu 18.04), so the software is open source (but hardware is not).

It uses the same basic principle as the InfiniteNoise device (see the previous section); however, one of the main goals of the designers was to avoid any cryptographical (and therefore based on unproven assumptions) whitening. Still some whitening is needed, and it uses `xor`-operation. Therefore:

- The multiplication factor closer to 2 is used (some variations in the electronic components are inavoidable; also the clock rate may be important, see below).
- The device included four separate bit generators, so their outputs can be `xor`-ed.
- The software driver performs "folding", i.e., two (or more) blocks of consecutive bits are `xor`-ed (in addition to the previous step); some self-testing is also included.

The software allows the user to change the defaults (use only some of the four generators or a xor of some set of them, switch off or increase the "foldering"). One can also change the clock rate (it

is 2.5MHz by default, so the device in rather fast, in the default configuration we got ≈1.1 Mbits per second). There is also an option to switch off the quality control, so in this way we can test the raw output of different devices with different clock rates.

The same Cantor-set-like structure is visible on the graphs for individual devices but it depends on the rate (Figure 6.20, page 58). Probably the visible uniformity is the best for the default rate 2.5 MHz. For the slower rates (100 kHz and 1 MHz) we get two rather similar pictures; it seems that there are some overlaps instead of gaps on all levels except the first one). For 5 MHz rate the gap becomes much larger (this corresponds to the decrease in the amplification factor in the model), and for 10 MHz the structure is almost destroyed: the values are much more concentrated, so we get less white points and much less entropy — so there are no advantages in using the rate that high).

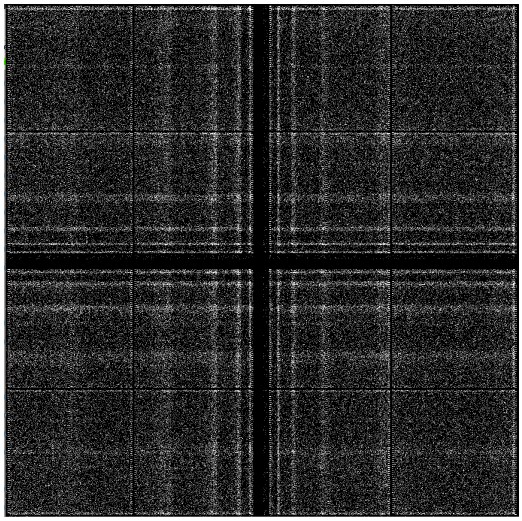
We may also compare graphs for the same rate and for different devices (recall that there are 4 in one white device), see Figure 6.21 (p. 59). Both devices have significant distortion but the graphs are visibly different.

The graph for `xor` of two devices (Figure 6.22) looks much more uniform, and it is hard to see the non-randomness (may be some hints of vertical and horizontal lines are visible). Still the tests (first of all `sts_serial` and `ent_8_16`) easily detect problems with the `xor` of two devices (Figure 6.24). For a single device the failures are even more obvious (though the folded version surprisingly did not exhibit any problems in our testing).

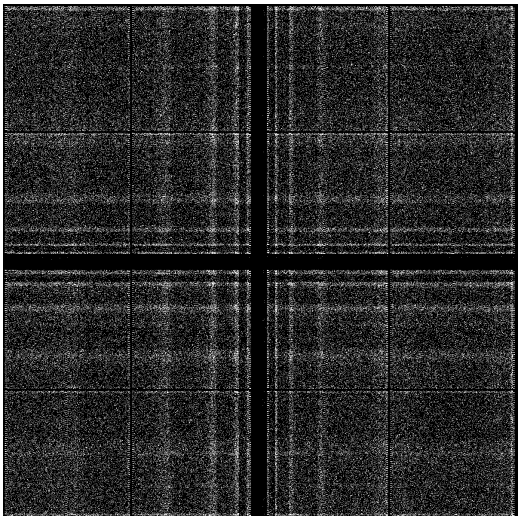
The graph for `xor` of three devices (Figure 6.23) looks even more uniform (though difference is hardly visible). Our testing did not show any suspicious behavior for samples of size up to 10 Gb both for the `xor` of three devices and the `xor` of all four devices.

Recall that (as we have said) the default output of the device is obtained by "folding" this 4-`xor`-stream; this decreases the speed of bit production by factor 2, but provides an additional security reserve.

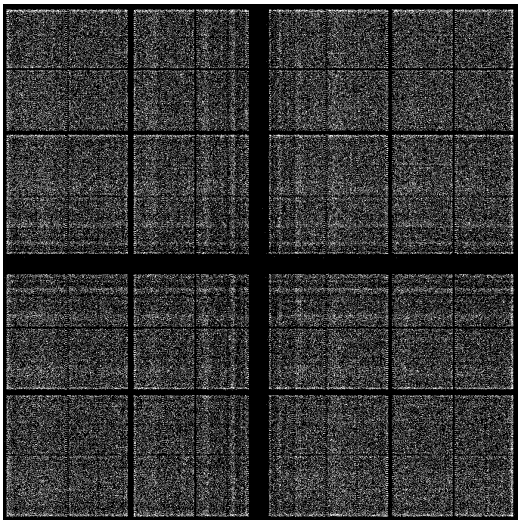
There exists a simpler (and cheaper) version of the device, called the "black" device (we did not test it). It includes only one bit generator, so whitening is based on foldering only.



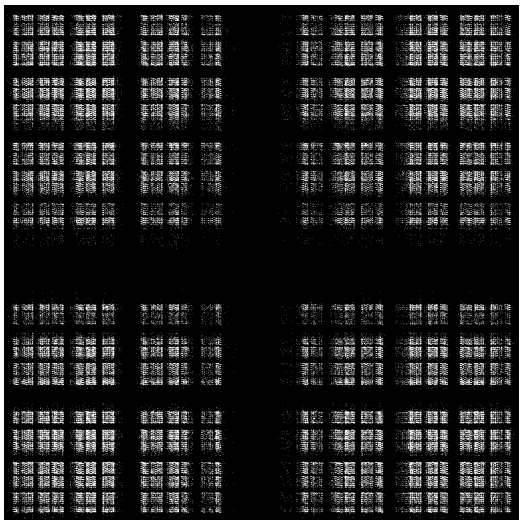
100 kHz



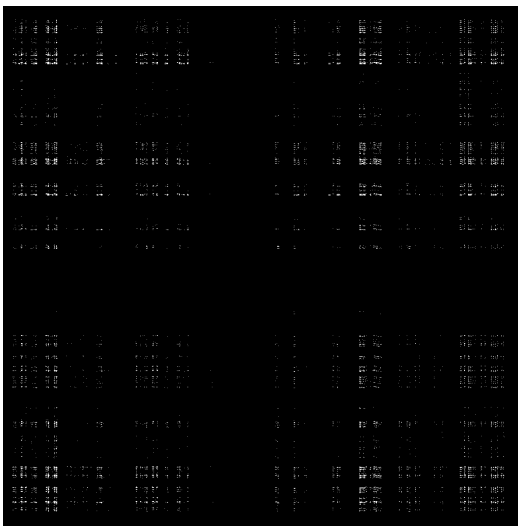
1 MHz



default rate 2.5 MHz

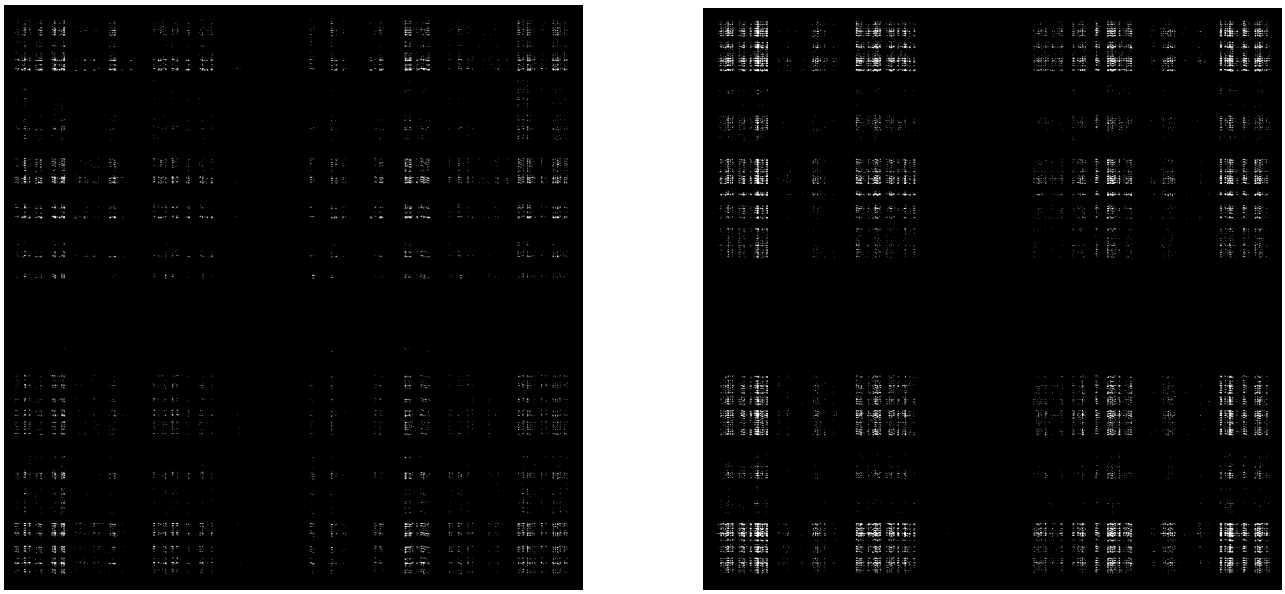


5 MHz



10 MHz

Figure 6.20: One of the four devices with different bit rates (no folding, no quality check).



Device 1 (10 MHz) Device 2 (10 MHz)

Figure 6.21: Two different devices, the same rate 10 MHz.

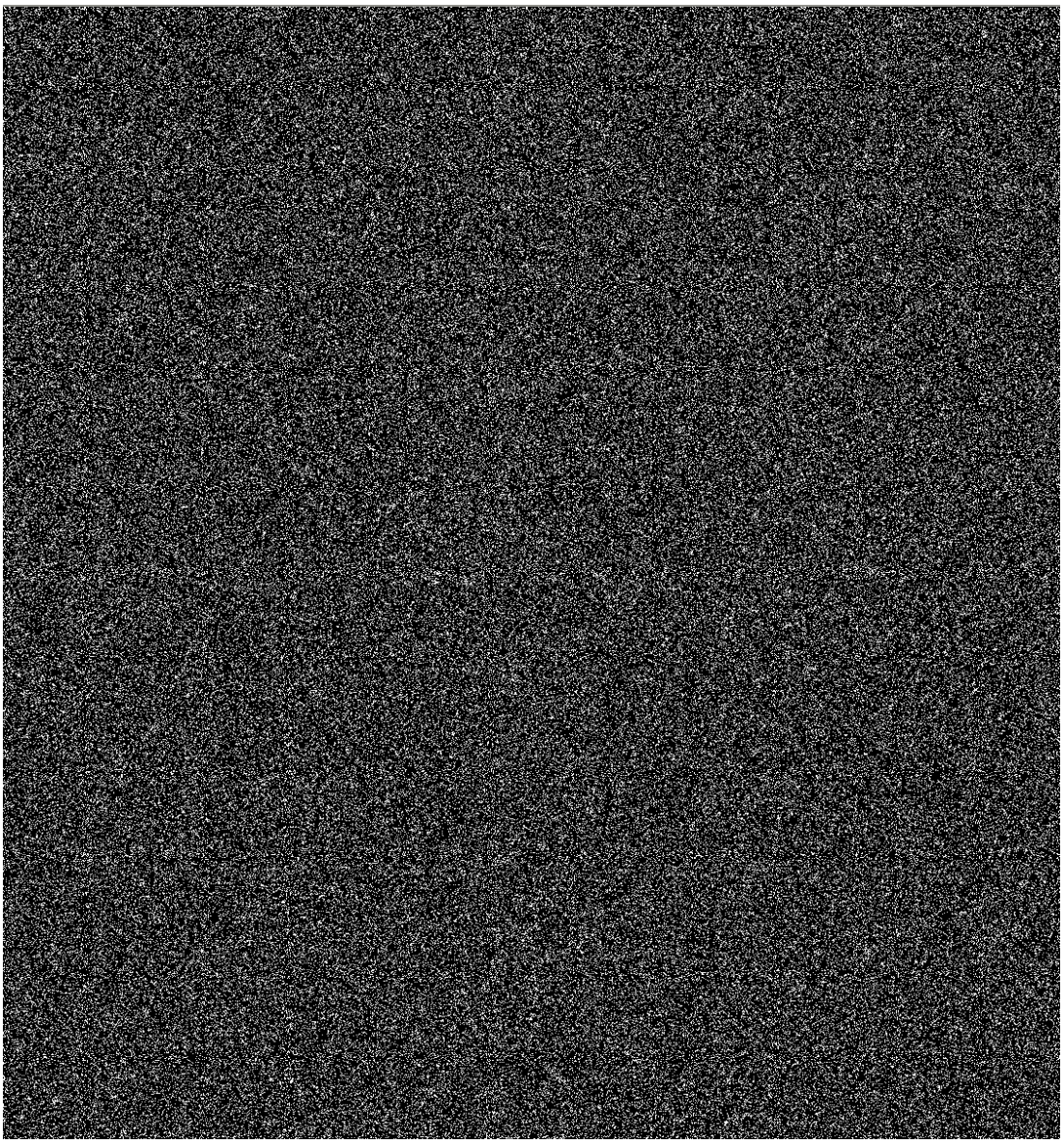


Figure 6.22: The `xor` for two different devices at default rate 2.5 MHz.

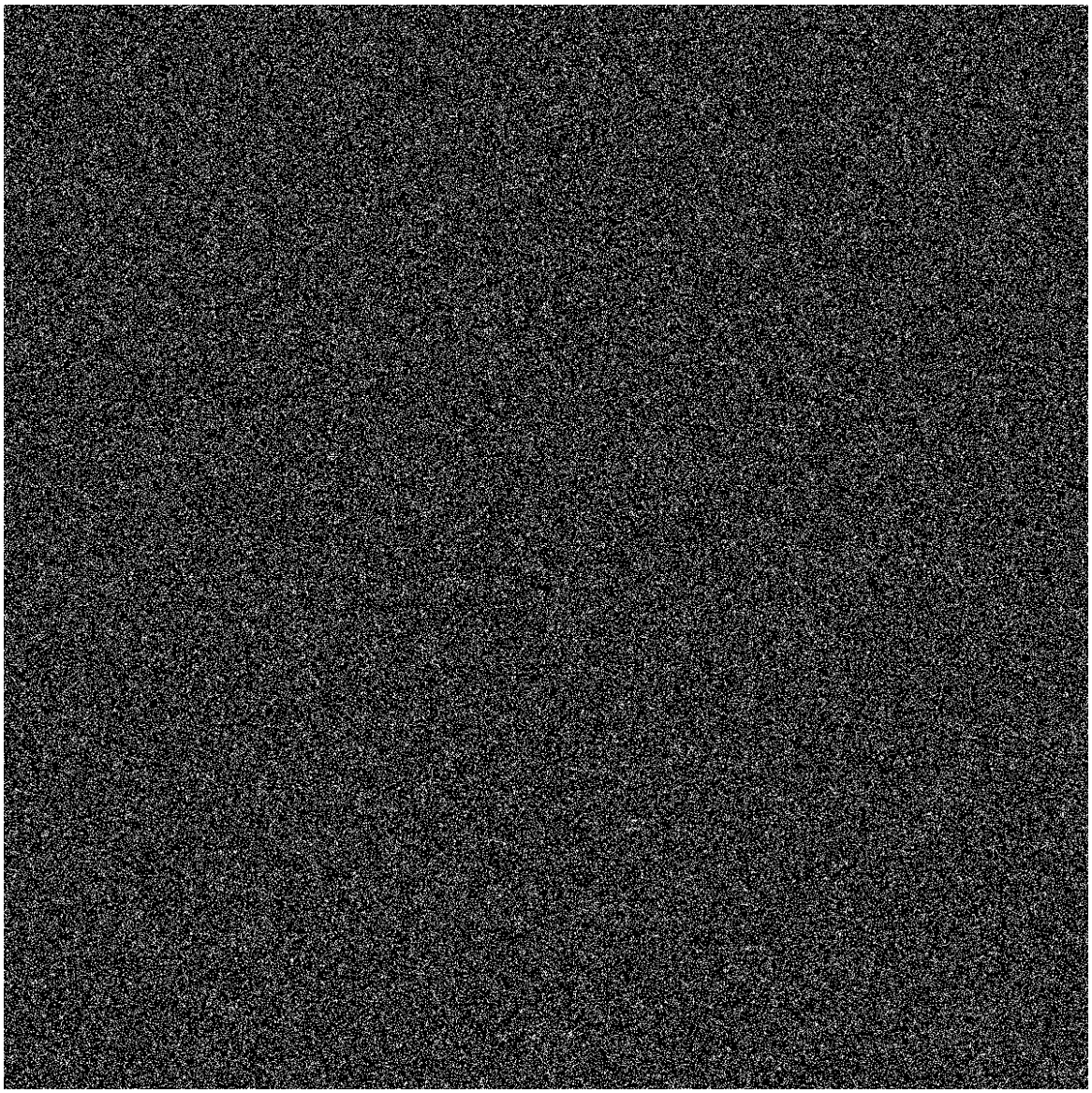


Figure 6.23: The xor for three different devices at default rate 2.5 MHz.

```
$ rtest -x -f raw_2_sources -e raw_4_sources-1 -p 100 -q 100 -n 10000 -t 9 -d 4 -r 1 -k
0.0240558028411 0.0240558028411 0.5830090612540 0.9084105017745
Used 8001600 bytes from the test generator [0]

$ rtest -x -f raw_2_sources -e raw_4_sources-1 -p 1000 -q 1000 -n 10000 -t 9 -d 4 -r 1 -k
0.0000000000000 0.0000000000000 0.1811645424830 0.3136800387321
Used 80016000 bytes from the test generator
```

Figure 6.24: Entropy tests for the xor of two generators in a Bitbabbler device.

6.9 Ubld.it

Figure 6.25: TrueRNG3 generator from `Ubld.it`

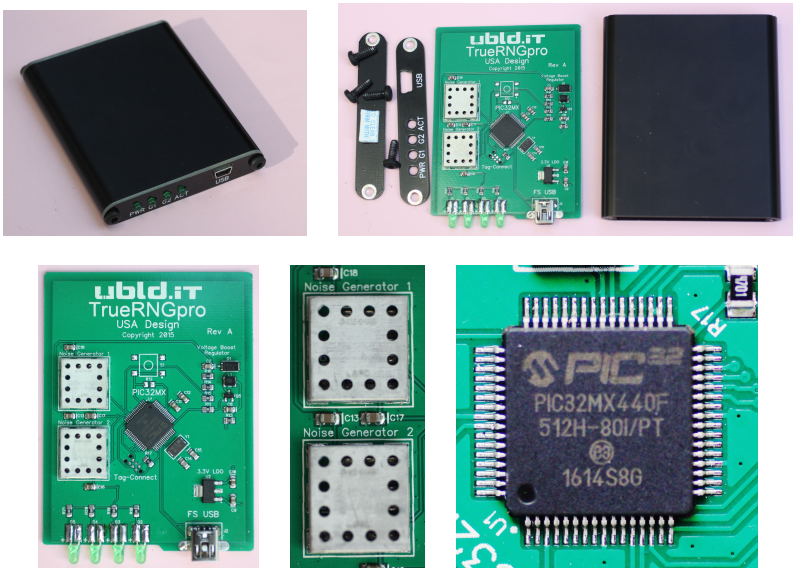


Figure 6.26: TrueRNGpro generator from `Ubld.it`; one can see two shielded generators and a processor chip.

The company `http://ubld.it/` produces devices of two types. The cheaper one, `TrueRNGv3`, is an USB stick (Figure 6.25). The more advanced one, `TrueRNGpro`, is a small box with USB connector and four LEDs (Figure 6.26).

According to the description, "The TrueRNG Hardware Random Number Generator uses the avalanche effect in a semiconductor junction to generate true random numbers. ⟨...⟩ The semiconductor junction is biased to 12 volts using a boost voltage regulator (since USB only supplies 5V), amplified, then digitized at high-speed. The digitized data is selected and whitened internal to the TrueRNG and sent over the USB port with more than 400 kilobits/second of throughput." As to whitening, they say that "entropy mixing algorithm takes in 20 bits of entropy and outputs 8 bit to ensure that maximum entropy is maintained. The algorithm uses multiplication in a Galois field similar to a cyclic redundancy check to mix the ADC inputs thoroughly while spreading the entropy evenly across all bits."

The advanced version has two shielded independent generators, metallic enclosure and LEDs (including two LEDs that show the results of the health checks for both generators). The software gives access to raw data from the generators, and to the whitened data stream from one or two generators (as requested by the user). Drivers for Linux (including `udev` rules) and Windows are provided (we tested only with Linux). Sample python program to get access to the random bits is provided. As to whitening algorithm, the FAQ on the company site says "We split the data into multiple streams

and use the XOR method to reduce bias (whiten) the output data. ⟨...⟩ A significant amount of time was spent getting the whitening correct without reducing the throughput too much. Currently, the random data is XORed/downselected at about a 20 : 1 rate to whiten while keeping maximum entropy." No detailed information about the whitening algorithm is provided. The raw data show a reasonable distribution (Figure 6.27) that is consistent with the description in the documentation.

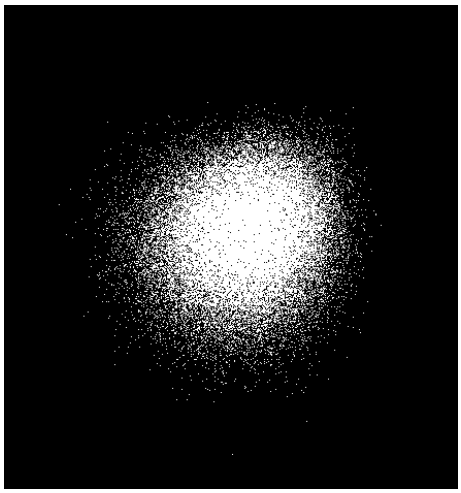


Figure 6.27: Raw data for TrueRNGpro generator. Each white point corresponds to raw readings from two generators (for two coordinates). Each generator produces a 10-bit value; however, most of the values are close to the middle of the $[0, 1023]$ interval (the graph shows about 600 middle values for both coordinates).

Speed: 400 kbits/sec for TrueRNGv3, 3200 kbits per second for TrueRNGpro. Price: around $50 for TrueRNGv3, around $100 for TrueRNGpro.

Both devices does not show any suspicious behavior in our limited testing (samples of size up to 1 Gb).

6.10 Homemade random bits generator

It may make sense to make a hardware random generator in a do-it-yourself way. First, in this way one can be sure that there are no Troyan horses (after minimal precautions); second, this helps to understand better the problems and possible solutions. So we performed a (completely amateur) experiment of this type, just to see what can be done without serious tools or preparations (Figure 6.28).

We decide to use Zener diodes as noise sources, use the standard entry-level audio mixing tables to digitize the signal, and get the signal from the basic circuit for Zener diodes: the resistor and the diode are connected to the power source, and the voltage from the diode is sent to the microphone XLR input of the mixer. Since standard recording programs normally work with two channels (for stereo sound), we made two circuits like this.

Some things that we have discovered while trying different settings.

- The noise from the power supply is comparable with what the Zener diode produces; not a surprise since the linear power supplies often use p-n-junctions for reference voltage. One may consider the power supply as just an additional source of noise, but we tried to filter it a bit (using electrolytic capacitor after a resistor).

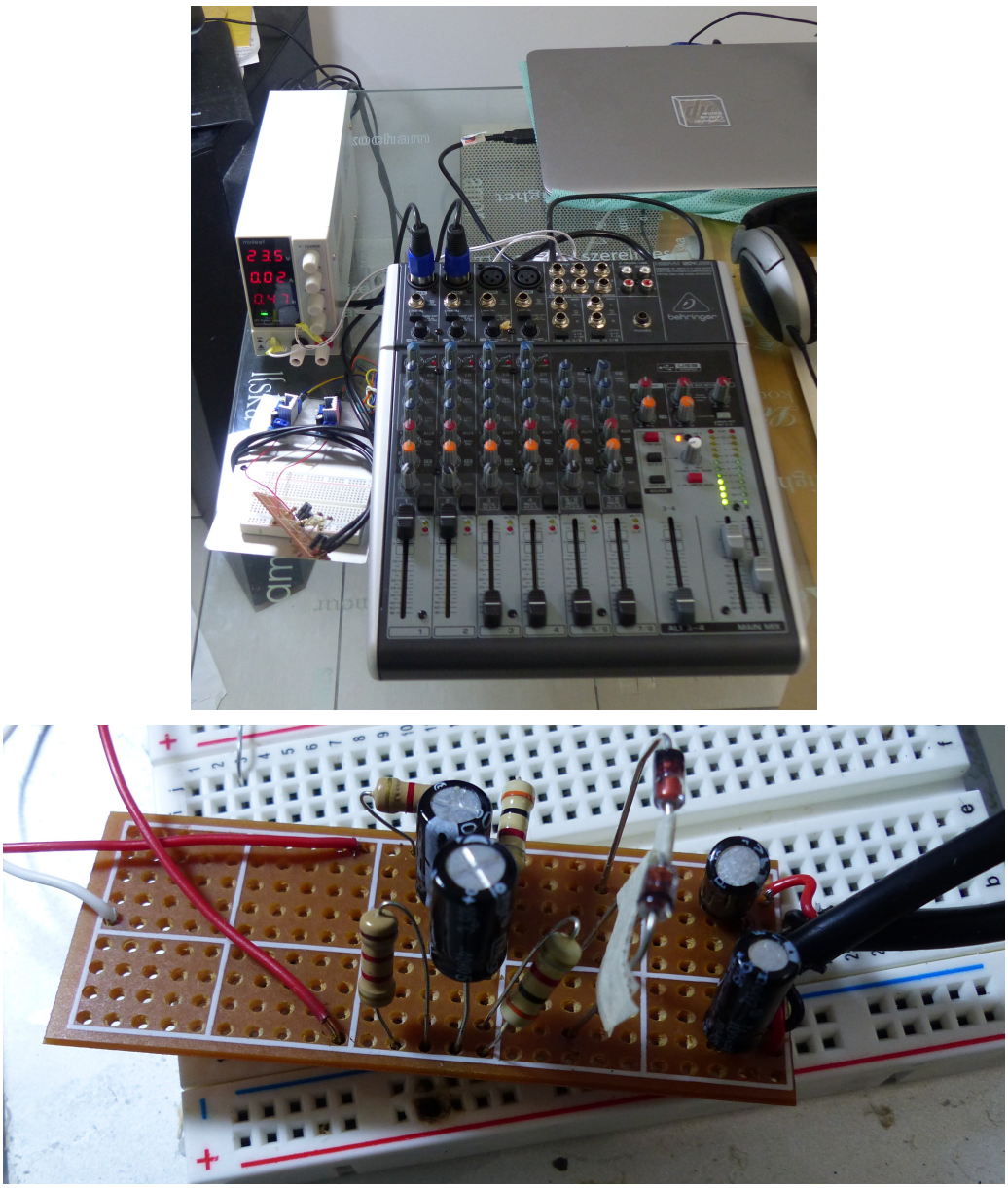


Figure 6.28: Recording noise from two Zener 12V diodes using Behringer XENYX 1204USB mixing table. The voltage (now 23.5 V) from the power source is filtered by two linear LM317 filters, giving two 20 V outputs; they are additionally filtered by RC circuit and then applied to the resistor — Zener diode circuit; the signal is sent to microphone XLR inputs through a capacitor.

- Since we wanted to have independent noise for both channels, two standard LM317 linear power supply circuits (producing 20V out of 24 volts from power adapter) for both channels were used.
- We ordered several batches of cheap Zener diodes from `aliexpress.com`, and made some preliminary tests with dozens of them. Different diodes (even from the same batch) can behave quite differently. In particular, there are two effects that we tried to avoid. First, for some diodes the spectrum of the recorded noise (tested with standard `audacity` software) is obviously non-uniform, going down almost linearly (in the logarithmic scales). Probably this means that the underlying physics mechanism generating noise is different, and the decline in the high frequency range is obviously bad for random bits generation (making the neighbor readings dependent). Second, for some diodes the graph has visible and audible spikes

("static"), and this also is bad for us: the digitized values for some periods have different behavior. Another observation was that maximal positive and negative (compared to the average: we connected the input through a capacitor to get DC isolation, so the average value was zero) peaks often have different amplitudes; we try to select diodes for which this difference was not very high.

- The noise also depends on the current, and it should not be too low to avoid unstable effects.
- Trying to record mono signals, we discovered that the combination of the mixing tables we used (Behringer 1204USB) and the recording software (`audacity` for Mac OS X and Linux) is non-symmetric in the sense that one channel is one reading behind: ith reading in the left channel corresponds to $i + 1$-th reading in the right one; this is clearly visible in `audacity` (Figure 6.29). (Strange, but easy to correct. The same effect was observed with Yamaha MW10 mixing table, but not with Yamaha AG06, so it is probably caused by the hardware, not by software.)

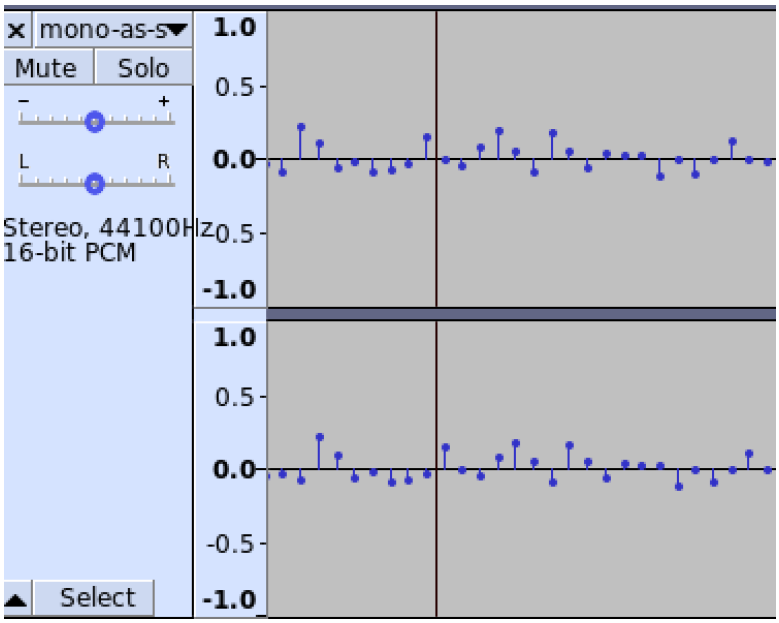


Figure 6.29: A mono signal recorded by Yamaha MW10 mixer and `audacity`; left channel (top row) is one sample ahead of the right channel (bottom row).

To diminish the external interference (that was audible as humming and visible as peaks, including 50 Hz harmonics on the frequency spectrum) we made a noise source with balanced output (as required by the XLR microphone inputs of the mixing tables). The spectrum of the output signal of this noise source (Figure 6.31) still has some peaks (as the picture shows, the biggest one is at 30Hz and we do not know its origin) and is not flat (has more low frequencies). Also one can see that the spectrum is well below the Nyquist frequency (half of the sampling rate), 91 kHz for our case, and one can see (Figure 6.32) that the consequent samples are dependent (e.g., often have the same sign). Another way to visualize this dependence is to draw the pairs of consecutive samples as points on the plane. If the signal changes slowly, then these points are close to the diagonal. On the other hand, if samples are independent, the distribution is the product of two one-dimensional distributions. Figure 6.33 shows a significant dependence between neighbor samples. The shape of the data curve and the spectrum depend on the Zener diodes used in the noise source (as we have mentioned earlier, the diodes from the same batch may have different properties) and the current used for noise generation. For comparison we made a simple noise generator (Figure 6.34) that uses the phantom

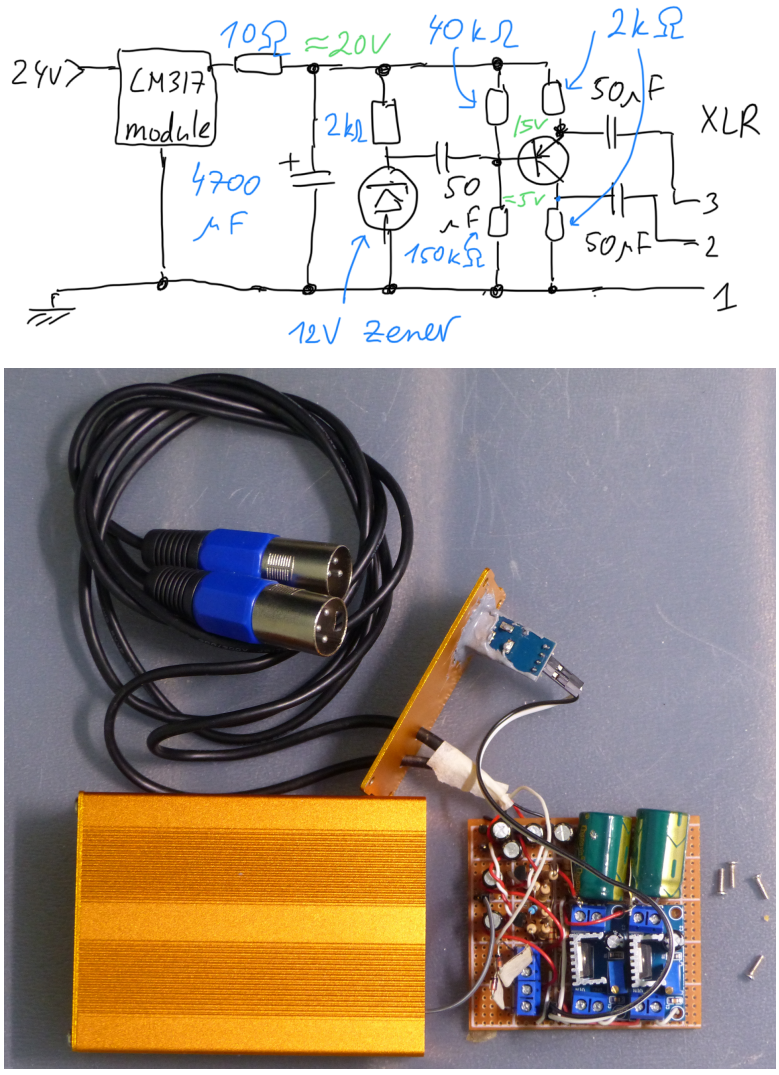


Figure 6.30: A noise source with balanced output (two identical channels).

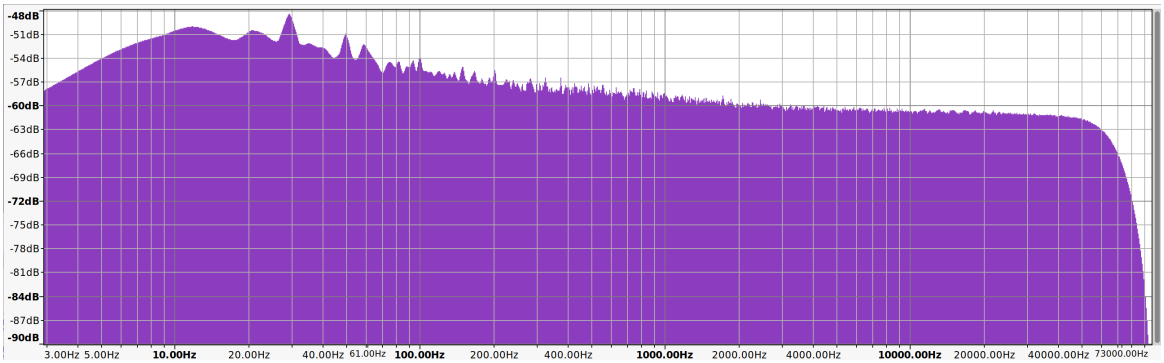


Figure 6.31: The spectrum of the noise source with balanced output, digitized by Yamaha AG06, 24 bits, 192 kHz, shown by `audacity` program.

power provides by some microphone inputs (all in Behringer XENYX 1204 usb and Yamaha MW10 and one of two inputs in Yamaha AG06). The signal is now the difference between noise signals from two Zener diodes. Looking at the data, we see that the curves are now significantly different and less symmetric (Figure 6.35), though the spectrum looks more flat. The advantage (except for

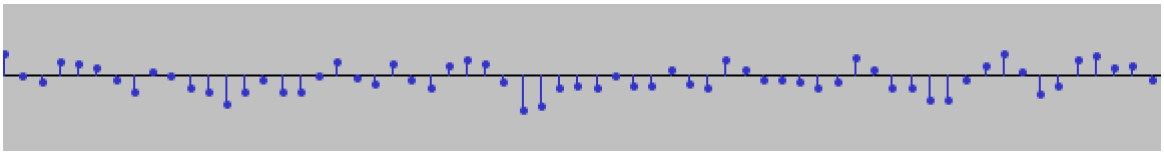


Figure 6.32: A sequence of noise samples (digitized by Yamaha AG06, 24 bits, 192 kHz) as shown by the same program.

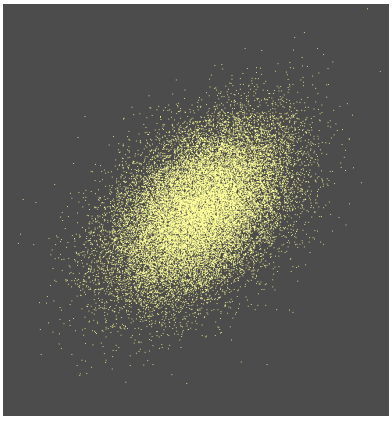


Figure 6.33: Sample pairs (digitized by Yamaha AG06, 24 bits, 192 kHz).

simplicity) is that no power adapter is needed.[14] The difference is also clearly visible on 2D plot (Figure 6.36).

One can probably get some random bit generators that pass the tests by using `xor` of several noise streams (obtained by different generators), as it is done in many previous devices. Still we get much smaller bit rates (assuming 1 byte for sample and 44100 Hz sampling, we get about 300 kbits per second, not much compared to the devices considered) and much bigger and more expensive device (as usual, paranoia has its costs).

We used the audiofiles obtained from 44.1 kHz 16-bit stereo `.wav` file from device shown in Figure 6.30 to try the two-source extractors described in Section 5.5 (see Section 7.6 for the implementation details). Our tests did not exhibit a suspicious behavior (for several 1 Gb samples).

Let us mention also that other "homemade solutions" are possible (see Figure 6.37) for more or less serious examples.

[14] A suitable power adapter could be a problem. Many cheap switching power adapters create a lot of parasite signals, probably due to bad isolation between the primary and secondary circuits or bad filtering of the high frequency oscillations.

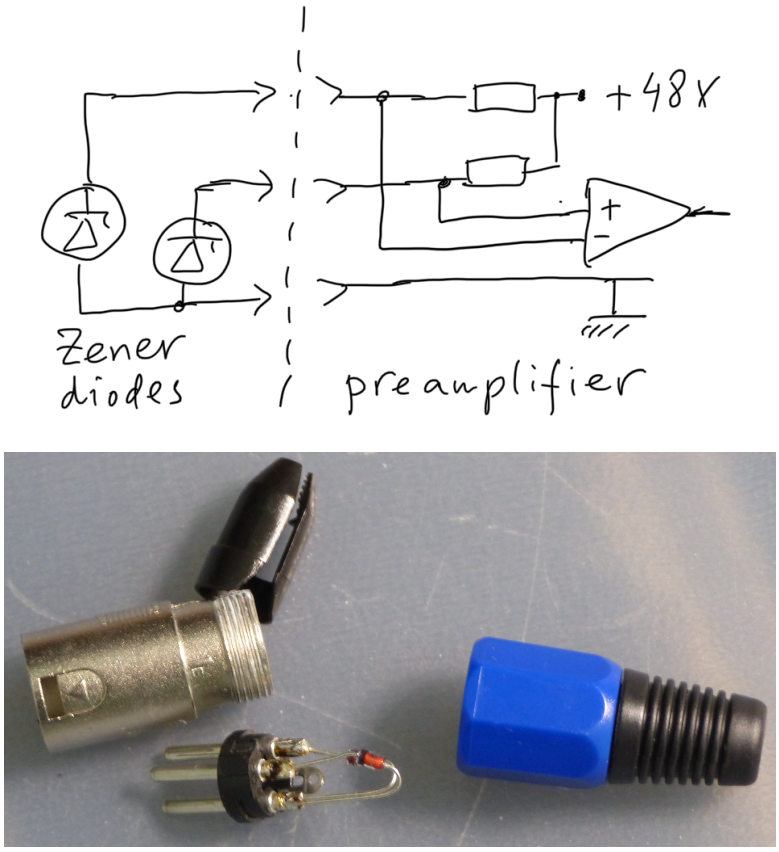


Figure 6.34: Two Zener diodes on an XLR connector can be used as a noise source if connected to a microphone preamplifier with balanced inputs and phantom power. The resulting signal is the difference between two noise signals from both diodes.

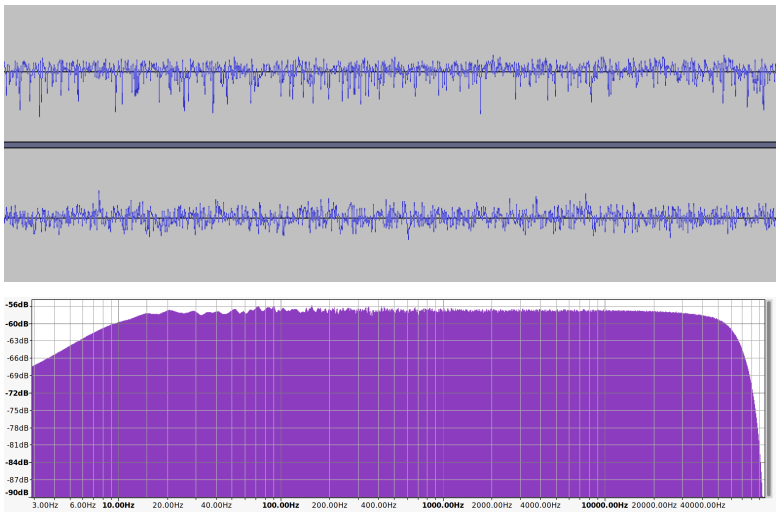


Figure 6.35: The signal from device with 2 Zener diodes (upper row), digitized by Yamaha AG06 (24 bits, 192 kHz), and the frequency spectrum of 2 Zener diodes device.

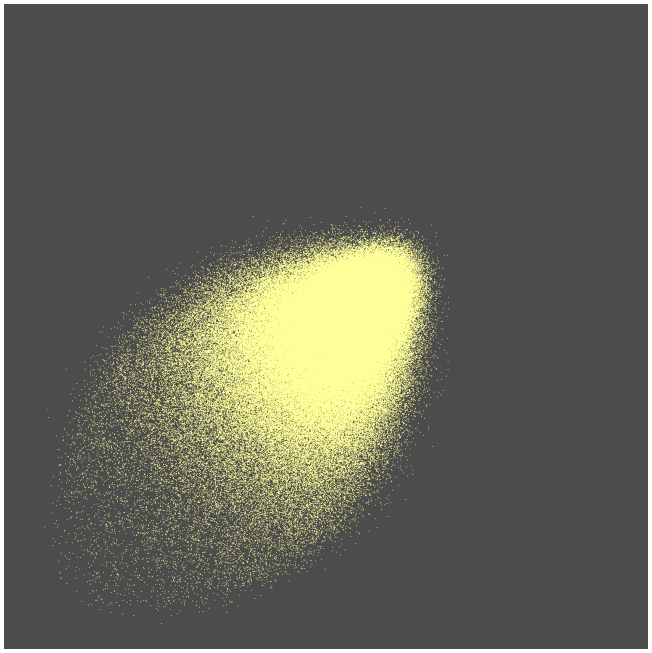


Figure 6.36: Pairs of consecutive samples from 2-diodes noise source (AG06, 24 bits, 192kHz).

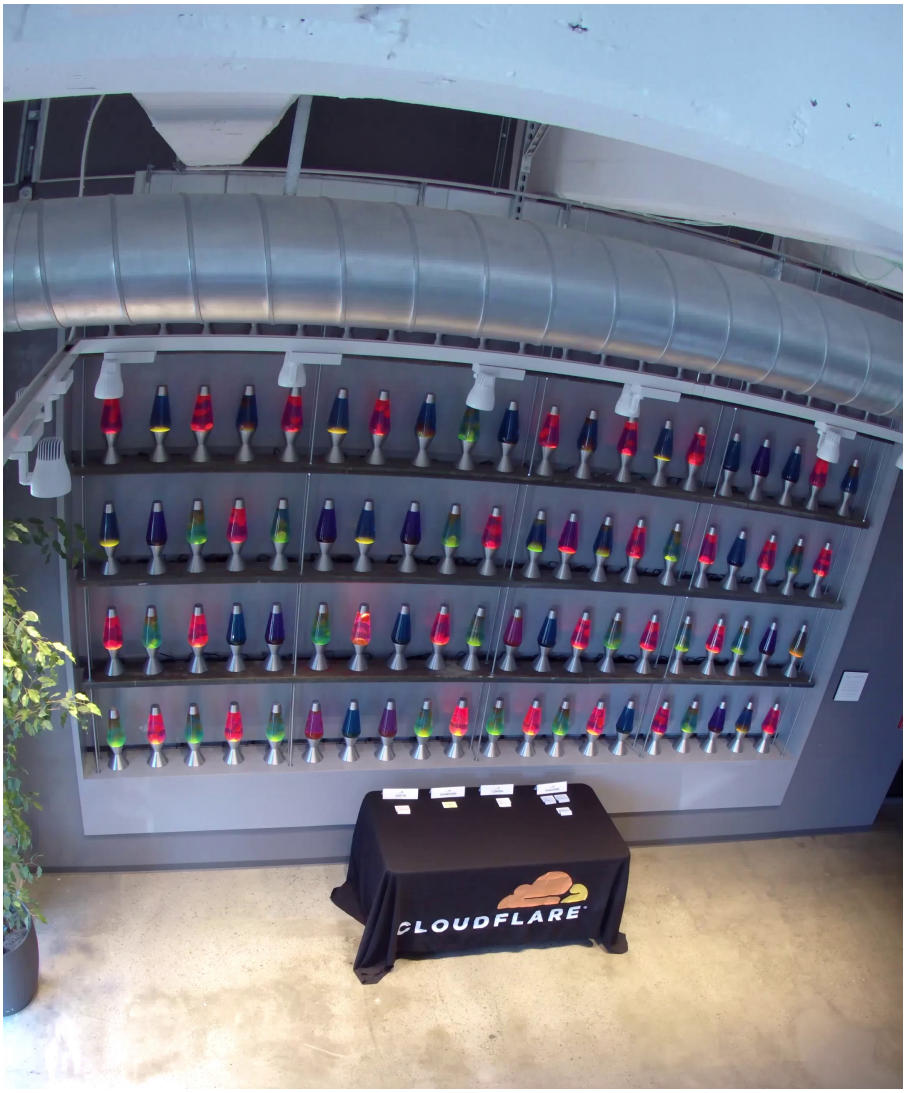


Figure 6.37: Randomness made visible: roulette wheel, lottery machine and lava lamps at `cloudflare.com` (pictures from `wikipedia` and `https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/`).

6.11 Testing synthetic pseudo-random generators

In many applications (e.g., randomized algorithms, communication protocols) people use pseudo-random generators instead of physical sources of "true randomness." Such a generator is typically initialized by a short seed (usually a few bytes), and then produces a very long sequence of bits that are computed from the seed deterministically. Such a sequence of bits has obviously very low entropy (bounded by the size of the seed). However, a pseudo-random sequences can have good statistical properties expected from a truly random sequence. In particular, such pseudo-random sequences can pass standard batteries of tests like Dieharder. The observed statistical "regularity" of pseudo-random sequences can be justify their usage in some applications.

We tested with Dieharder several (pretty standard) pseudo-random generators listed below.

Generator 1: A version of a pseudo-random generator by Blum, Blum, and Shub [5]. We choose a number M that is a product of two prime numbers (in our experiments $M = 100000004483 \times 100000004987$), a seed x_0 that is co-prime with M, and compute a sequence of values $x_{i+1} = x_i^2 \mod M$ for $i = 0, 1, 2, ...$ The the least significant bit of each x_i is taken as the outcome of the generator.

Generator 2: A very simple version of a linear congruential generator. We let

$$M = 100000004483 \times 100000004987$$

(this choice of the parameter makes it easier to compare this generator with the BBS Generator 1), and two numbers modulo M; in our experiments these are $A = 444444$ and $B = 55555555555$. Then we choose a seed x_0 co-prime with M, and define $x_{i+1} = Ax_i + B$ for $i = 0, 1, 2, ...$ The the 32 least significant bits of each x_i a taken as the outcome of the generator (this is about a half of the size of the internal state of the generator, which is an integer number modulo M).

Generator 3: Another (more conventional) version of a linear congruential generator used in the the standard POSIX. It is similar to Generator 2, the only difference is the choice of parameters: we let

$$M = 2^{48},\ A = 25214903917,\ B = 11.$$

The output of the generator are the 4 less significant bytes of the current x_i. This scheme is used, for example, in the function `mrand48()` of the GNU C Library and in Java's class `java.util.Random`.

Generator 4: A simple and very fast generator based on *linear feedback shift register* proposed by G. Marsaglia, [32]. Here is the implementation in C borrowed as given at [33]:

```
struct xorshift64s_state {
    uint64_t a ;
};
uint64_t xorshift64s(struct xorshift64s_state *state)
{
    uint64_t x = state->a;
    if (! x) {
        x = (uint64_t) SEED; // initialize
    }
    x ^= x >> 12; // a
    x ^= x << 25; // b
    x ^= x >> 27; // c
    state->a = x;
    return x * UINT64_C(0x2545F4914F6CDD1D);
}
```

In our experiments we take the 32 least significant bits returned by `xorshift64s()`.

Generator 5: We use the same generator as in Generator 5, but we take only one bit from each value retuned by `xorshift64s()` (in our experiments we took the 15th least significant bit of the outcome).

Generator 6: In this example we use random bits produced by the well known generator called Mersenne Twister [35]. We used the standard version implemented in 2002 by Takuji Nishimura and Makoto Matsumoto, [46] (we used all 32 bits from each value returned by the generator).

Generator 7: In this example we use a permuted congruential generator (PCG), which is described as having good statistical performance despite small and fast code [45]. We used the following code borrowed from the PCG library on `https://www.pcg-random.org`.

```
typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;
uint32_t pcg32_random_r(pcg32_random_t* rng)
{
    uint64_t oldstate = rng->state;
    // Advance internal state
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc|1);
    // Calculate output function (XSH RR), uses old state for max ILP
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}
```

Observed results : In our experiments, generators 1, 2, 5, 6, 7 pass the standard tests of Dieharder; also the spectral tests do not reveal for them any abnormality. Generator 3 and 4 fail the battery of tests of Dieharder. It is interesting that Generator 3 produces random graphs with a spectral radius pretty similar to the properties of "truly random" graphs (we perfrmed experiments, in particular, for graphs with 256 vertices and average degree 16). On the other hand, the random graphs (with the same number of vertices and average degrees) obtained with Generator 4 have dramatically different spectral properties.

In our experiments, each graph was represented by a rather short sequence of bits (typically by a block of 4096 bytes). We believe that spectral properties of much larger graphs deserve more careful research in future.

6.12 Concluding remarks

Our analysis of the devices is quite superficial: for a more serious one we would need to analyze the PCB, the firmware, make extensive testing in different environmental conditions, etc. However, some observations could be made.

- There are significant improvements since the time when Marsaglia developed `dieharder` for randomness testing (1990s): only one of of the devices tested by us failed immediately our health tests (while all three hardware devices tested by Marsaglia did fail miserably, even after correction for the dos-unix conversion problem).
- There are several kinds of threats. First, one may be afraid of hardware instability in different conditions. For that, one should avoid devices that use the environment as a source of randomness (as it is done by gniibe/neug device) since they require extensive testing in different environmental conditions to be sure that enough randomness is available. The devices that uses electromagnetic noise are also unreliable in this respect.

- One can be afraid of a trojan horse installed by the device maker or on the way. Then one should avoid device with microprocessors and prefer to use device with easy to check or open architecture (like bitbabbler, or, better, the infinite noise device). The ultimate solution would be to make the device in-house using one of the standard approaches or the open-source hardware data; the bit shift devices or the noise sources with some off-the-shelf ADC or even a commercial audio card could be a reasonable solution.

- One can use inadvertent deficiencies in the device due to bad or instable components. Then the ability to get the raw data for systematic check on the go is important (so you can see if, say, a Zener diode changes its parameters suddenly). For example, from this viewpoint TrueRNGpro defice is much better than TrueRNGv3 one.

- The most paranoid people may be afraid of security holes in widely used cryptographical primitives like hash functions and other tools prescribed by the standard (recall that even accepting unproven assumptions from complexity theory we have no mathematical reasons to trust these standard tools). Then simple and well-understood tools like `xor` of several sources are preferable even if they require more raw random bits.

- On the other hand, if we care about small statistical regularities and do not worry much about malicious adversaries, the cryptographic primitive are usually more or less enough, and they still provide minimal defense against most intruders even if the statistical properties of the primary source degrade. They also allow a trade-off between speed and security since we may get much more than N pseudo-random bits from N random bits and still be practically indistinguishable from true random source. The resulting generator can be viewed as pseudo-random generator with random seed and continuous reseeding from hardware random source.

- Note that it is easy to combine protective measures against different kind of threats just by `xor`-ing the outputs of several generators that are resistant to different attacks: it is enough to have one correctly working generator to guarantee the correct distribution at the output. This assumes that we do not doubt the independence of the sources — but for the most practical cases having several usb devices bought independently from different manufacturers is probably enough (assuming that our application is able to `xor` the output sequences correctly).

- There are more exotic approaches; for example, some people use lava lamps as sources of randomness: `https://qz.com/1642628/cloudflare-uses-lava-lamps-to-generate-a-crucial-resource/` (Figure 6.37) — this is definitely good for advertisements and provides some level of "security through obscurity". One can also note that a cheap noisy video camera (preferably with minimal post-processing) has a lot of internal noise that is available via a high bandwith channel.

- Of course, one can also use existing high-end solutions to get a high-speed stream of random bits, or some certification, or some additional feeling of security, or a chance to use it for advertisements. Still it is questionable whether this is really much more than a "security theater".

Warning: In this analysis we did not consider an important aspect: how much additional programming / system administration / hardware installation work is needed for this or that generator. This depends on the specific nature of the application (e.g., the operating system in use, where the generators should be installed, etc.), and can make some approaches practically unusable.

Chapter 7

Comments on the software used

We start (Section 7.1) with observations made when analyzing the existing code (mainly from `dieharder` suite since it is widely used). Then we describe some software tools that we developed for our experiments. Section 7.2 describes the tools related to the Kolmogorov–Smirnov two-sample test. Section 7.3 describes the utility that includes robust version of more than 15 tests; Section 7.5 describes some graphic tools (very primitive, but useful for quick assessment). Finally, Section 7.6 describes the implementation of a simple two-source extractor with provably good properties based on finite fields (discussed in Section 5.5).

7.1 Dieharder: some notes and comments

Since Dieharder [9] utility is often used for practical testing of hardware random generators, it makes sense to look at it more closely and note some pecularities. Our remarks refer to the version 3.31.1 (which seems to be the current one in March 2021).

- Flag `-T` is mentioned in the code (`dieharder/parsecl.c`, lines 97–100)

  ```
  if(i == TCNT) {
         fprintf(stderr,"Invalid -T %s option.\n",optarg);
         exit(1);
       }
  ```

  It seems that `-T` options is not mentioned in the man page. This message appears when the name given for the -D option does not exist in the table of names (so `-T` should be probably replaced by `-D`). The same `-T` is mentioned in the *comments* in `dieharder/dieharder.h`, lines 18–19:

  ```
  * is set in set_globals to a default to be used if -T 0 is selected.
  * tflag is otherwise accumulated from a series of -T FLAG calls, where
  ```

- Too many random bits read in the bitstream test for overlapping (main) mode, `libdieharder/diehard_bitstream.c`, lines 73–78:

  ```
    * for non-overlapping samples we need (2^21)*5/8 = 1310720 uints, but
    * for luck we add one as we'd hate to run out.  For overlapping samples,
    * we need 2^21 BITS or 2^18 = 262144 uints, again plus one to be sure
    * we don't run out.
    */
  #define BS_OVERLAP 262146
  ```

In fact, for overlapping samples we need $2^{21} + 19$ bits, i.e., about $2^2 1/32$ 32-bits integers (`uint`). This is $2^{21-5} = 2^{16} = 65536$ (+1 for the +19 additional bits). So the last line should be

```
#define BS_OVERLAP 65537
```

(Of course, a good generator should pass the test even if some bits are ignored, but in this way the test uses four times more bits than needed, and this requires more time for hardware generator testing.)

- In the same test, there is an error in counting the strings that do not appear as 20-bit substrings of the generator. Namely, line 172 is

  ```
  w[w20]++;
  ```

  it increments the counter of the number of appearances of bit string `w20`. There is a small problem: `w` is defined as `*char` in line 113:

  ```
  w = (char *)malloc(M*sizeof(char));
  ```

  so the counting happens modulo 256. In the rare case where some string appears 256, 512, etc. times (it should be very rare, since the expected number of occurrences is only 2, still theoretically possible), it will be counted as missing. The obvious correction is to make the counter `bool` (we do not need the count value, only the yes/no information), replacing the line 172 by `w[w20]= 1;` this solves the problem.

- The dieharder `bitstream` test ignores 8 first bits of the stream both in overlap and non-overlap modes (for different reasons). For the non-overlap mode the bitstream is different, since the memory is considered as a sequence of bytes, not 32-integers, as it is done in the overlap mode, and byte ordering is different. The lines 120–12 in `diehard_bitstream.c` are actually needed only for the overlap case, but increase `i` by 1 anyway, so in the non-overlap mode the first byte is ignored.

- The dieharder `bitstream` test does not react properly to the `-L` option (always uses overlapped sampling), because line 52 of `diehard_bitstream.c` says so (and local variable is used instead of the global one).

- With small number of samples, p-value is sometimes reported as `1.00000` and test fails (this should not happen).

  ```
  $ dieharder -g 1 -p 2 -d 1
  #=============================================================================#
  #            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
  #=============================================================================#
     rng_name    |rands/second|   Seed   |
             cmrg|  4.63e+07  |3941174114|
  #=============================================================================#
          test_name   |ntup| tsamples |psamples|  p-value |Assessment
  #=============================================================================#
        diehard_operm5|   0|   1000000|       2|1.00000000|  FAILED
  ```

  The source of the error is that (unfortunately!) the computation of Kolmogorov–Smirnov statistics in `kstest.c` is incorrect (e.g., one should never divide by `count+1`), so for small sample length n it produces impossible values (like $p = 0.05$ for $n = 2$) and therefore the `ks_test` function returns 1, interpreted as a failure.

- The call `dieharder -g 1 -S 1 -d 1` produces different results when run several times, though `-S 1` should fix the seed. The reason is that in `choose_rng.c` the seed is given to the generator before measuring the speed (line 246), and reseeding happens when measuring the speed (lines 251-253) if `-D 8192` (speed measuring) is on, so the result depends on the the display options (a rather confusing behavior);

  ```
  \texttt{dieharder -d 1 -g 1 -p 10 -S 1 -s 0 -D pvalues -D 8192
  ```

  produces different values too (though it seems that strategy prevents reseeding and the seed is fixed). This can be corrected if we reseed again after timing tests in `choose_rng.c` However, `rgb_timing()` changes the global variable `seed`. To avoid this, one could make variable `seed` local (as `unsigned long int`) in the `rgb_timing.c`, function `rgb_timing()`.

- The implementation of Marsaglia `opso` test [31] the code (`diehard_opso.c`) does not correspond the description both in the paper and in the code itself. The description suggests to consider a sequence of $n = 2^{21} + 1$ integers, consider them as 32-bit words, and keep only 10 bits in some fixed position. In this way we get a word in 1024-letter alphabet having length $2^{21} + 1$. Then we count how many of the 1024×1024 two-letter words in this alphabet do *not* appear among 2^{21} factors of length 2 of this n-bit strings. (This is a random variable whose distribution is assumed to be very close to a normal one, and this is checked.)

  At the same time, code says (here t is an integer varible that is incread by 1 in the `for`-loop):

  ```
  if(t%2 == 0) {
    j0 = gsl_rng_get(rng);
    k0 = gsl_rng_get(rng);
    j = j0 & 0x03ff;
    k = k0 & 0x03ff;
  } else {
     j = (j0 >> 10) & 0x03ff;
     k = (k0 >> 10) & 0x03ff;
  }
  w[j][k] = 1;
  ```

  We see that instead of overlapping pairs (as the name of the test says) here two pairs of 10-bit strings are taken from different (disjoint) positions in a pair of two 32-bit integers obtained from a random number generator (`j0`,`k0`). These two pairs are registered in the bit array `w`, and then a new pair of integers is obtained. So the code makes something completely different. (In fact, the `dieharder` man page characterizes this test as "suspicious".)

- A similar problem exists with another test from Marsaglia, namely, the `oqso` test [31]. According to the description (reproduced in the dieharder source), this test uses 5-bit groups (instead of 10-bit groups), gets a word of length $2^{21} + 3$ in 32-letter alphabet, and then considers 2^{21} factors of this word of length 4, and counts the 4-letter words that do not appear as factors. This is done for every of $28 = 32 - 5 + 1$ positions, so we get 28 counters.

  The code in `diehard_oqso.c` does something completely different:

  ```
  for(t=0;t<test[0]->tsamples;t++){
    if(t%6 == 0) {
      i0 = gsl_rng_get(rng);
      j0 = gsl_rng_get(rng);
  ```

```
    k0 = gsl_rng_get(rng);
    l0 = gsl_rng_get(rng);
    boffset = 0;
  }
  /*
   * Get four "letters" (indices into w)
   */
  i = (i0 >> boffset) & 0x01f;
  j = (j0 >> boffset) & 0x01f;
  k = (k0 >> boffset) & 0x01f;
  l = (l0 >> boffset) & 0x01f;

  w[i][j][k][l]=1;
  boffset+=5;
}
```

Instead of taking (overlapping) factors, it makes 6 quadruples from disjoint bits in four 32-bit integers. (There is no reason to think that the resulting test is worse than the original `oqso` test, but it does not match the description in the same file.)

- In the runs test (that goes back to Knuth), the code in `dieharder` (file `diehard_runs.c`, lines 132–142) the first and the last elements of the tested block of integers are compared (with no apparent reason, Knuth's description and `runtest.c` did not do anything like that).

- The overlapping sum test is described in [29] as follows. We take the sum of m independent random variables uniformly distributed in $[0,1]$ (obtained from m unsigned integers). Compute their sum; its distribution is approximately normal. Then replace one of the number by a fresh random real in $[0,1]$; we get another (heavily correlated) random variable whose distribution is close to normal. Doing this $m-1$ times, we use $2m-1$ random reals and produce m overlapping sums. Each of the sums is approximately normal, and they are correlated. Marsaglia suggests to apply a linear transformation that will convert these (approximately) normally distributed variables to (approximately) independent ones, convert them to (approximately) uniformly distributed independent variables and to use Kolmogorov–Smirnov one-sample test to check this hypothesis (of uniformly null distribution). The original test used $m = 100$, so from every 199 unsigned integer we got one p-value. Then the distribution of 100 these p-values was assumed to be uniform and again Kolmogorov–Smirnov test was applied to them to get a secondary p-value. Finally, ten secondary p-values were submitted to one more ("tertiary") Kolmogorov–Smirnov test.

  Obviously this procedure is based on several layers of approximations, and no attempts to provide an error bound were made. (Also the derivation of the suggested linear transformation was not given, just some formulas in file `die.c/cdosum.c` from [29].)

  The `dieharder` code (file `diehard_sum.c`) complains about these problems saying that "unfortunately, Marsaglia's description is pretty opaque". Still the code from `cdosum.c` for the linear transformation is reproduced, but with a change. Here is a fragment from `diehard_sum.c`:

```
x[0] = y[0]/sqrt(1.0*m);
x[1] = -x[0]*(m-1)/sqrt(2.0*m - 1.0) + y[1]*sqrt(m/(2.0*m - 1.0));
 [...]

for(t=2;t<m;t++){
```

```
[...]
a = 2.0*m + 1.0 - t;
b = 2.0*a - 2.0;
/* x[t] = y[0]/sqrt(a*b) - y[t-1]*sqrt((a-1.0)/(b+2.0)) + y[t]*sqrt(a/b);*/
x[t] = y[t-2]/sqrt(a*b) - y[t-1]*sqrt((a-1.0)/(b+2.0)) + y[t]*sqrt(a/b);
```

The commented line is taken from Marsaglia's code; however, the first term is changed (for no apparent reason and with no explanation).

It is noted in the beginning of `diehard_sum.c` that "At this point I think there is rock solid evidence that this test is completely useless in every sense of the word. It is broken, and it is so broken that there is no point in trying to fix it. If I crank up tsamples to where one can actually see the asymptotic distribution (which should be uniform) it is visibly non-uniform and indeed the final kstest *converges* to a non-zero pvalue of 0.09702690 for *all* rngs [random number generators] tested, which hardly seems useful. If one runs the test on only 100 samples (as Marsaglia's code did in both the original fortran or the newer C version) but then increases the number of runs of the test from the default 100, it is easy to make it fail for gold standard generators. The test just doesn't work. It cannot be used to identify a generator that fails the null hypothesis. Don't use it."

We tried Marsaglia's recipe to get (presumably) independent uniformly distributed variables and its version from `dieharder` by computing 20000 overlapping sums from three generators. (Figure 7.1). As we see, these six curves are quite close to each other and differ significantly from the diagonal (which would appear if the distribution is truly uniform). So it seems that indeed this test uses approximation assumptions that are too rough (both in the original form and in the changed one). Its two-sample version, of course, is still valid (as it happens for every test function), only the sensitivity could be a problem.

- File `dieharder_rank_32x32.c` starts with comments

```
* This is the Diehard BINARY RANK 31x31 test, rewritten from the
* description in tests.txt on George Marsaglia's diehard site.
*
* This is the BINARY RANK TEST for 31x31 matrices. The leftmost ::
* 31 bits of 31 random integers from the test sequence are used ::
* to form a 31x31 binary matrix over the field {0,1}. The rank  ::
* is determined. That rank can be from 0 to 31, but ranks< 28   ::
* are rare, and their counts are pooled with those for rank 28. ::
* Ranks are found for 40,000 such random matrices and a chisqua-::
* re test is performed on counts for ranks 31,30,29 and <=28.   ::
```

  Still it tests ranks of 32×32 matrices (according to the file name and `dieharder` messages).

- For the 6×8 rank test the `dieharder` comments look even more confusing. The description says

```
* This is the Diehard BINARY RANK 6x8 test, rewritten from the
* description in tests.txt on George Marsaglia's diehard site.
*
* This is the BINARY RANK TEST for 6x8 matrices.  From each of  ::
* six random 32-bit integers from the generator under test, a   ::
* specified byte is chosen, and the resulting six bytes form a  ::
```

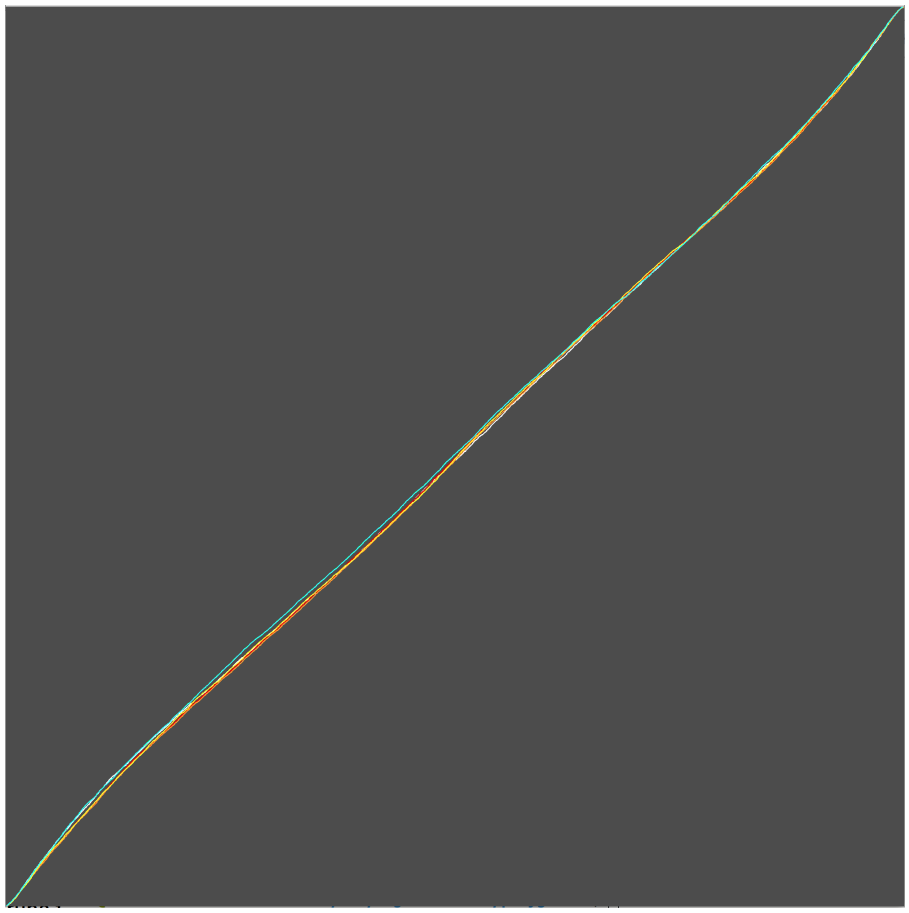


Figure 7.1: Six empirical distributions of presumably uniform samples obtaind using three different generators. Three use the original Marsaglia's transformation recipe, and the other three incorporate `dieharder`'s changes).

```
* 6x8 binary matrix whose rank is determined.  That rank can be ::
* from 0 to 6, but ranks 0,1,2,3 are rare; their counts are     ::
* pooled with those for rank 4. Ranks are found for 100,000     ::
* random matrices, and a chi-square test is performed on        ::
* counts for ranks 6,5 and <=4.                                 ::
```

It is not clear how the words "specified byte" should be understood[1] The comments in the code say

```
/*
 * We generate 6 random rmax_bits-bit integers and put a
 * randomly chosen byte into the LEFTMOST byte position
 * of the row/slot of mtx.
 */
```

But there are traces of the "random choice" of a byte, it seems that just the 8 first bits are selected from each 32-bit integer. (There are also strange things in the code. For example, six rows and eight columns with integer values are reserved for the matrix:

```
mtx = (uint **)malloc(6*sizeof(uint *));
for(i=0;i<6;i++){
  mtx[i] = (uint *)malloc(8*sizeof(uint));
}
```

which is strange since the rank computations in `rank.c` use individual bits in unsigned integers (and only `mtx[i][0]` are used later in the code).

- In the birthday tests (`cdbday.c` in `diehard` and `diehard_birthdays.c`) there are several peculiarities in the code. The description (given by Marsaglia are reproduced by Brown) says that m days are randomly chosen in a year of n days, then the intervals between these "birthdays" are considered and the "number of values that occur more than once in that list" is computed.[2] This distribution is claimed to be asymptotically close to Poisson distribution with parameter $m^3/4n$, and a χ^2-tests is used to measure this closeness. According to Marsaglia's description, "A sample of 200 j's is taken and a chi-square goodness of fit test provides a p-values". (In fact, Marsaglia's code uses 500 values of j, not 200. Also, Marsaglia uses values $m = 2^{10}$ and $n = 2^{24}$, both in the comments and the code in `cdbday.c`, but Brown, reproducing Marsaglia's comments, cites another value $m = 2^9$, along with $n = 2^{24}$. So the Poisson parameter becomes 2 instead of 16.

  Marsaglia gets random 24-bit words (birthdays) from 32-bit random integers (that are tested) using bits 1–24, then using bits 2–25 (from the same integers), etc., thus getting 9 different p-values from the chi^2-test. Then "the nine p-values provide a sample for a KSTEST". This process is based on several unfounded assumptions. First, the approximation by Poisson distribution is only an approximation (no derivation is given, not to speak about error bounds). Then χ^2-test needs to be applied, and the standard practice is to combine the values with small probabilities (rule of thumb says the we should combine tail bin in such a way that the

[1] It seems that in the original `diehard` code the χ^2-test was applied for each of 25 position separately (which is reasonable), but then Kolmogorov–Smirnov test was applied to the 25 results given by χ^2-tests, and this is a bad idea, since they are not independent.

[2] Strangely, both Marsaglia and Brown consider the interval that crosses the year boundary in a special way: only the part in the new year is counted.

expected value for a bin is at least 5). Marsaglia does this according to the rule of thumb, while Brown considers only the right tail and just ignores that bins with expectation less than 5 instead of combining them). As Brown writes in `diehard_birthdays.c`, if the number of repetitions (j) is bigger that the allowed maximal value (called `kmax`), "we simply ignore — it is a **bad idea** to bundle all the points from the tail into the last bin, as a Poisson distribution can have a lot of points out in that tail!". (This argument looks very strange, and we cannot explain the reasoning.)

There are two other points to be mentioned. One of them is the interpreration of the words "Let j be the number of values that occur more than once in that list" (of intervals between birthdays). Should we literally count just the values or, if there are three intervals of the same length, we could count them as two repetitions, so j is the difference between the cardinalities of a multiset of intervals and set of intervals? Marsaglia uses the second interpretation, and Brown uses the first one ("We count the number of interval values that occur more than once in the list. Presumably that means that even if an interval occurs 3 or 4 times, it counts only once!").

The second point is the overlapping sampling done by Marsaglia (bits 1–24, then 2–25, etc.). Brown writes in the comments that "I'm running it on all 32 24-bit strings implicit in the variables. We'll do this by rotating each variable by one bit position in between a simple run of the test. A full run will therefore be 32 simple (rotated) runs on bits 1–24". But this is not what happens in the code, it seems [3] that the 24-bit portions are taken from the input bit stream without any overlap (which is perfectly OK and makes the different p-values independent, but is not what is said in the description).

To illustrate how these different assumption are reflected in the data, we show two empirical distributions graphs. They are done for $n = 2^{24}$, $m = 2^{10}$ (as in Marsaglia's code). We considered all the intervals (including the one that crosses the year boundary) in the same way, used the Poisson distribution with recommended parameter $m^3/(4n)$ and chi^2-test with combined bins in the both tails (until the expectation is at least 5). The size of sample that was compared with Poisson distribution was 500 (as in Marsaglia's code); for this value bins 8 ... 26 are counted (all numbers smaller than 8 are combined with 8, and all numbers greater than 26 are combined with 26). As one can see from the graphs (Figure 7.2), the assumption of uniform distribution of the χ^2-test p-value seems very improbable (for both ways of counting the repetitions), so the validity of the test in the original form is questionable.

7.2 Kolmogorov-Smirnov 2-sample tools

As we have mentioned in Section 3.1, one can use Kolmogorov–Smirnov criterion for two samples to get a robust randomness test. Assume (null hypothesis) that $a_1, \dots, a_m$ and $b_1, \dots, b_n$ are real numbers independently sampled from some distribution (in our example: real functions of blocks of random bits). Let us sort the array $a_1, \dots, a_m, b_1, \dots, b_n$ and list all the values in the increasing order. Then we forget the values, and remember only where they come from (writing letter `a` or `b`). We get a sequence of $n+m$ letters that consists of m letters `a` and n letters `b`. The assumption (null hypothesis) implies that all strings of this type could appear with equal probabilities.

Therefore, if we have some subset of the set $S(m, n)$ of all strings of this type that has small cardinality (compared with the total number of all strings of this type, i.e., $\binom{n+m}{n} = (n+m)!/n!m!$), this set can be used as a test set.

[3] It is hard to say it for sure: the code for cutting the random stream in the blocks of requested size, `bit.c`, has 1441 lines. Still this is what is claimed in the comments.

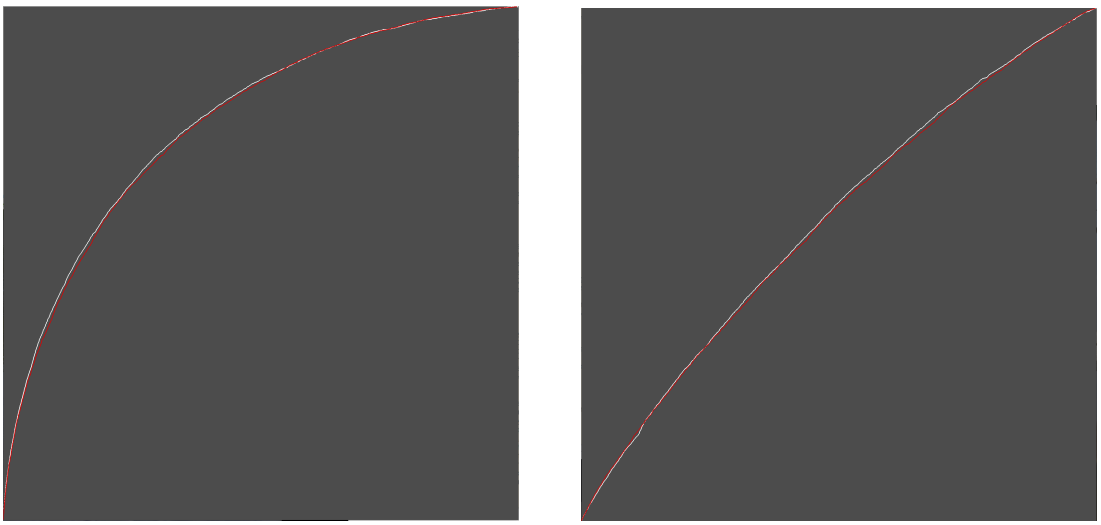


Figure 7.2: Empirical distributions for birthday test: 2^{10} birthdays, year length 2^{24}, distribution of the number of repeated intervals is compared with Poisson distribution ($\lambda = 16$) using samples of size 500. The p-values obtained by χ^2 test are accumulated from several thousands of trials and their empirical distribution function is shown. Two ways of counting repetitions are used: according to `dieharder` approach (interval length that repeated $k > 2$ times is counted as one repetition, *left*) and to the `diehard` approach (length that repeated $k > 2$ times is counted as $k - 1$ repetitions, *right*).

Kolmogorov–Smirnov test considers a function on $S(m, n)$ called *maximal deviation*. It is defined as follows: for each prefix of a string that consists of m letters `a` and n letters `b` we count the numbers m' and n' of letters `a` and `b` in this prefix, and then compute the *deviation* as

$$\left|\frac{m'}{m} - \frac{n'}{n}\right|.$$

One can represent our string as a path from $(0, 0)$ to $(1, 1)$ where each letter a means going right by adding $(1/m, 0)$, and each letter b corresponds to going up by adding $(0, 1)$. Then the deviation is proportional to the maximal distance between the path and the diagonal (Figure 7.3).

For a given deviation value d, one can compute Kolmogorov–Smirnov p-value, i.e., the fraction of strings from $S(m, n)$ that have the same or bigger deviation. The file `ks2.mp` does this using GNU multi-precision library using standard dynamic programming. The computation is exact (and later the answer is converted to double floats). The function

```
double ks2bar(int n0, int n1, long deviation_times_n0_n1)
```

gets two integers (number of both letters), and the maximal deviation multiplied by the product of these two integers (to make it integer) and returns the p-value as described above.

The code for computing the same value used in R statistical package (`http://ftp.stjude.org/pub/software/JUMPm/labeled/R-3.1.0/src/library/stats/src/ks.c`; the adapted version is in `ksmirnov.c`) uses the same technique but works with floating point numbers instead of multi-precision library, and includes also an asymptotic formula. One can compare the answer using the utility `ks2compare`:

```
$ ks2compare 1000 700 50000
Two samples two-side exact Kolmogorov-Smirnov p-value:
p-value for two samples (1000+700) and deviation 0.071429 [=50000/1000*700] is
0.028360425691996
For reference:
```

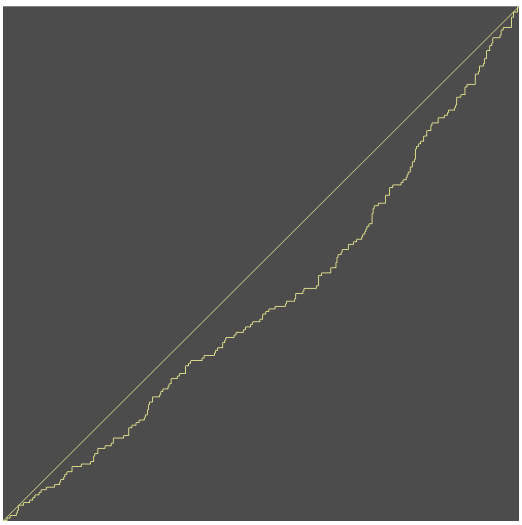


Figure 7.3: A path obtained while comparing two samples (produced by `ks2graph` utility). The deviation from the diagonal is quite visible, and this is confirmed by the numbers: characters appear 500 and 199 times, the deviation is 0.156, and the Kolmogorov–Smirnov p-value is 0.00173.

```
  asymptotic formula: 0.029940690
  floating point iterations: 0.028360426
```

The words "two-side" indicate that we consider the deviation in both directions (with absolute values).

The experiments show that for reasonable lengths (several thousands) all the approaches give the answer fast (so it is not a bottleneck) and the floating point iterative computation gives small error. So the asymptotic formula is not really needed (this is good news since there is no easy error estimate) and one may use the computation with floating point numbers and check the answer with an exact computation if needed.

The utility `ks2val` takes a string (made of two different characters) from the standard input and computes the deviation and Kolmogorov–Smirnov p-value (using the exact computation):

```
$ echo 0100010111011 | ks2val
Profile:
  length: 13
  character [0] appears [6] times
  character [1] appears [7] times
Maximal deviation: 0.547619
KSvalue:    0.212121212121212
```

The utility `ks2vala` does the same thing as `ks2val` using floating point (long double) computations.

The utility `ks2draw` does the same thing as `ks2val` and also draws the path corresponding to the input string (see Figure 7.3 for example).

Note that for Kolmogorov–Smirnov comparison of two samples only the relative ordering between their elements is important. However, to understand better the nature of the difference between distributions it could be useful to draw the actual distribution functions for several samples. The utility `distribs` takes several files (with the names indicated in the command line); each file contains lines with real numbers considered as a sample. The utility draws (on the same graph) all the distribution functions for given samples (see Figure 7.4).

To obtain data for Kolmogorov–Smirnov analysis, one can use python script `ks2files.py`. It gets two file names from the command line. Each line in both files should start with a floating point

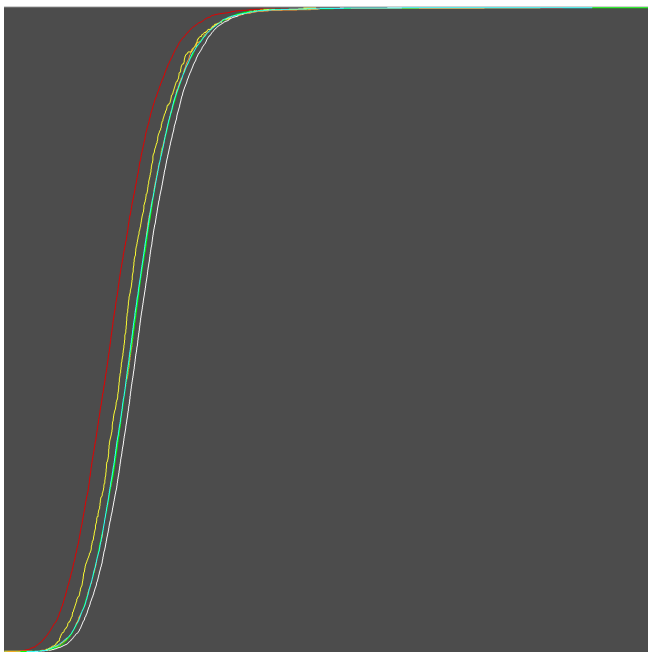


Figure 7.4: Utility `distribs` shows distribution functions for several samples obtained by computing spectral radii for random graphs obtained from Moonbase OneRNG generator in different regimes. The difference is clearly visible (and can be confirmed by computing p-values).

real (that is used as an element of the sample); the rest of the line is used for tie breaking (no two lines should be exactly the same). The program sends to the standard output the sequence of characters `1` and `2` that indicate the origin of the data point after merging and sorting (whether it comes from the first or the second file).

Finally, Kolmogorov–Smirnov test can be used to test independence (as described in Section 3.5). This is done by utility `ks2indep`. This utility is applied to two files (the names are given in the command line). Each file is considered as a sequence of bytes, divided into groups of fixed length (in bytes, determined by option `-b`)., Then each group of bytes is converted into a signed integer, interpreting them in the LSB (least significant byte first, option `-l`) or MSB (most significant byte first, option `-m`) way. These two sequence of integers ($n_0, n_1, \ldots$ and $m_0, m_1, \ldots$) are used in the following way: if n_i satisfies [resp. violates] some condition (compiled-in list of condition is used; one of them is chosen by `-c` option; now two conditions are implemented: being positive [0] or being even [1]), the corresponding m_i is sent into the first or the second subsequence). This is done until a given number of pairs (option `-n`, default 1000) are classified; then two subsequences are subjected to Kolmogorov–Smirnov test, the p-value is printed and the procedure is again applied to the rest of the files.

This procedure is a valid statistical test for the null hypothesis: both sequences are obtained by independent sampling from two independent sources. The distributions of the sources may be different, but both independence between different integers from the same source and independence between sources are important. Also we assume that there are no ties (if they happen, a warning is printed using `stderr`).

To reduce the independence between integers in one source one could skip several reading after each pair of integers used as described above. The skipped integers in the second source are also used for resolving ties. The number of skipped integers is given as `-s` option parameter (default 0).

A special regime is allowed with only one file (option `-i`). Then both integers are taken from the same file (first n_0, then m_0, then n_1, then m_1, etc.) and integers are skipped only after $m_0, m_1, \ldots$ (skipped integers again are used for tie resolving).

Finally, the numbers of p-values in ten buckets $[0, 0.1), \ldots, [0.9, 1)$ (ten integers) are printed (if `-v` verbose option is used).

7.3 Robust tests utility

We implemented several tests using the general scheme described in Chapter 3. There is a general driver (`rtest.c`, utility `rtest`) that calls different test functions depending on the options. It uses the following options:

- `-t test_num` the number of test function used (see the list of all test functions below);
- `-f test_file` specifies the name of the file that is tested. The file is understood as a sequence of 32-bit unsigned integers (four consecutive bytes form an integer) or a sequence of bytes (the interpretation is determined by a test function used);
- `-e etalon_file` specifies the name of the file that is used for comparison. If no `-x` option is used, then the test function is applied to data from `test_file` and `etalon_file`; in the robust mode (with `-x`) the test function is applied to data from `test_file` and to the `xor` of (new) data from `test_file` and data from `etalon_file`;
- `-x` requests the robust mode (see above for `-e` option);
- `-p num_samples` specifies the number of times the test function is applied to (fresh) data from tested file;
- `-q num_samples` specifies the number of times the test function is applied to etalon data;
- `-d dimension` indicates the number of values returned by each call to the test function (different test function have different dimensions); this number is the number of p-values returned by each Kolmogorov–Smirnov two-sample test;
- `-n number` determines how many bytes are consumed by each call of the test function; the exact interpretation of this number depends on the test function; some tests ignore this value and decide themselves how many bytes they need; for some other tests only some values of this number are allowed;
- `-o directory_name` asks to create files with the list of values that were submitted to the Kolmogorov–Smirnov two-sample test, and tells the name of the directory where these files should be created; the directory should not exist before the call. Every Kolmogorov–Smirnov value obtained by the test is accompanied by two files in this directory (that show both samples); one can understand the behavior of the test better when looking at these files.
- `-r` requires the test to be repeated; if a positive number is provided as an argument, it is interpreted as the number of repetitions; zero argument means that the test should be repeated as long as there are enough data in the input files.
- `-k` requires the exact computation with multi-precision integers for Kolmogorov–Smirnov two-sample p-value (instead of long double computation performed by default); can be useful for very large `-p` or `-q` options (exceeding 5000).

The program runs as long as there are data in the input files. After one test is performed and several Kolmogorov–Smirnov values are produced (the number of these values is the dimension of the test), the testing continues with the rest of the bits in both files.

Implementation remarks: the main driver file is `rtest.c`; the definitions for the test functions are given in `test_func.c` and additional files (see the macro `TESTS_C` in the `Makefile`). To add a new test function, one should put its code in a `.c` file, put its forward definition in `test_func.h` and add it to the list of all test functions in `test_func.c`.

We used hashes in a different way to simplify the code structure: the function returns not only some values (the number of returned values is the dimension), but also returns 64 bits (two $32-bit$ integers) that are used to resolve the ties: if the test function returns two identical values, these hash values are compared for sorting. Normally 64 bits should be enough to resolve the ties (if this does not happen, this means that the file we are testing is not random enough, so it creates a failure by itself).

The tools to compute the Kolmogorov–Smirnov p-values (for two samples; some tests also use calls to Kolmogorov–Smirnov distribution function for one sample) are taken from another directory (see `KS_C` macro definition in `Makefile`).

The following tests are currently implemented (listed according to their test numbers used in `-t` option; the C name of the corresponding function is shown after the number).

0 `all_bytes`

Dimension 1, ignores `-n`, counts the number of bytes that one needs to read to see all 256 possible bytes (coupon collector problem) if this is not too large (does not exceed some maximal number that is returned otherwise). Uses about 1.5 Kbyte per test value (this number should be multiplied by `-p` plus `-q` option parameters for the full test).

1 `all_16`

A similar test to the previous one but for 16-bit integers (pairs of bytes), so it requires much more data (1.5–2 Mbytes per test value)

3 `sts_serial`

The test depends on two parameters: the number of bits (n) analyzed and the maximal length of substrings analyzed (m). It produces an array of $3m$ values, i.e., 3 values for each integer from 1 to m. For each integer in this range there are two values called p_1 and p_2 in the NIST document; we report also one auxiliary "raw" value. The value of n is 32 times `-n` option; the value of m is taken as `-d` option divided by 3 (integer part). Requires n bits ($n/8$ bytes, 4 times `-n` option), the reasonable application is possible when $n \gg 2^m$ (otherwise the frequencies of factors do not make much sense).

Let X be the sequence of n bits to be analyzed. For each word w of length at most m we count the number $count[w]$ of occurrences of w in X (considered as a cyclic word); the sum of all $count[w]$ for all words w of given length equals n. The we compute $\psi^2[k] = (2^k/n)\sum_w(count[w]^2) - n$ where the sum is taken over all strings w of length k.

For our purposes the values $\psi^2[k]$ (raw values mentioned above) can be used directly, but NIST recommends to consider the first and second differences

$$\Delta\psi[k] = \psi^2[k] - \psi^2[k-1],$$
$$\Delta_2\psi[k] = \psi^2[k] - 2\psi^2[k-1] + \psi^2[k-2].$$

The `dieharder` code suggests to let $\psi^2[0] = 0$, so $\delta\psi[k]$ is defined for $k \geqslant 1$ and $\delta\psi_2[k]$ is defined for $k \geqslant 2$. (Non-existing values are filled by zeros in the test output).

These numbers are converted to p-values according to approximate distributions,

$$p_1 = igamc(2^{k-2}, \Delta\psi[k]/2),$$
$$p_2 = igamc(2^{k-3}, \Delta_2\psi[k]/2).$$

4 `opso`

Original description: [31]. (The `dieharder` implemenation deviates from the original description in a rather strange ways, see Section 7.1; we followed the original description). Test consumes $2^{23}+1$ unsigned 32-bit integers and generates $32-10+1=23$ values according to 23 possible positions of 10-bit substrings in a 32-bit string. For each of 23 positions we get a word of length $n=2^21+1$ in 2^{10}-letter alphabet. We count 2-letter words that are *not* factors of this word, and convert this count to a (presumably uniform) p-value assuming normal distribution with some parameters (mean about 141909.33 and σ about 290.46). The `-n` parameter should be 2097153, the `-d` parameter should be 23. Uses about 8 Mbytes per test value.

5 `oqso`

The test is similar to `opso` and is described in the same paper. (Again the `dieharder` implementation does something rather different.) The difference is that the alphabet contains 2^5 letters (factors of length 5 at different positions in the input integers), and we count factors of length 4 that are missing. Consumes $2^{23}+3$ integers and produces 28 values. Approximation mean and variance are slightly different (141909.60 and 294.656). The `-n` parameter should be 2097155, the `-d` parameter should be 28. Uses about 8 Mbytes per test value.

6 `bytedistribs`

A reimplementation of a test from `dieharder`, file `dab_bytedistribs.c`. From every three 32-bit numbers from the generator (`dieharder` allows also generators of different word size, but we assume the size is 32) one extracts 9 bytes (three from each integer: 8 least significant bits, 8 most significant bits and 8 bits in the middle). This is repeated n times (specified by `-n` parameter), so we get 9 distributions on $\{0,1\}^8$ corresponding to 9 different byte positions. Then we apply χ^2-square test to the combined distribution on $9\cdot256$ objects with $ndf=9\cdot255$ degrees of freedom (since we have 255 degrees of freedom for each of 9 positions). One resulting value is returned (test dimension is 1). More precisely, each of $256\cdot9$ counters has expected value $n/256$ and actual value x; we compute the sum S of $(x-e)^2/e$ for all counters and then compute the approximate p-value as $gsl\_sf\_gamma\_inc\_Q\,(ndf/2, S/2)$. Uses 12 times `[-n]` bytes; the `-n` parameter should be much bigger than 256 (otherwise the distributions do not make much sense).

7 `knuth_runs`

Test described in [25, p. 65] (the `dieharder` deviates from it since it considers the first element in a special way, but probably the results do not differ much). For a sequence of (arbitrary) length n we count the number of parts in its minimal splitting into non-decreasing subsequences (runs). Then we consider 6-vector consisting of numbers $x_1,\dots,x_6$ of runs of lengths 1, 2, 3, 4, 5, 6-∞. Then (as explained by Knuth [25]) one computes

$$V=\frac{1}{n}\sum_{1\leqslant i,j\leqslant 6} a_{ij}(x_i-nb_i)(x_j-nb_j),$$

where coefficients a_{ij} and b_i are given by tables provided by Knuth. Test dimension: 2 (for non-increasing and non-decreasing runs). Uses [`-n` option] integers ($4\times$ `[-n]` bytes).

8 `osums`

This is the "overlapping sum" test described in Marsaglia [29] and then included in `dieharder` [9] with come changes. Still after some investigation Robert Brown, the author of `dieharder`, came to the conclusion that this test should not be used (see Section 7.1). Still (like any other test function) it provides a correct test if used in two-sample scheme, so we included it. The test takes any number $m>1$ as size (`-n` parameter) and uses $2m-1$ input 32-bit unsigned

integers (converted to $[0,1]$ double reals) to produce m overlapping sums (using m integers in each sum). Then some linear transformation (suggested by Marsaglia) is applied to get m presumably independent standard normal variables, they are converted to presumably uniform random variables in $[0,1]$ and Kolmogorov–Smirnov one-sample test is used. The resulting p-value is the only test output (dimension is 1). As we have discussed, it seems that for large m there is a significant deviation from the assumed distributions, and Kolmogorov–Smirnov one-sample tests gives small values. So it may be wise to stick to the value $m = 100$ as used in original Marsaglia `diehard` test. Uses about $8 \times$ `[-n]` bytes

9 `ent_8_16`

This simple test of dimension 4 takes arbitrary number n of 32-bit integer (specified in `-n` option), and returns four numbers (has dimension 4). It considers distributions on bytes and 16-bit integers. For each of these distributions the entropy is computed in a standard way, and also χ^2-test is applied (with 255 and 65535 degrees of freedom for 8 and 16 bit versions). The four numbers returned are (entropy for 8 bits, χ^2-test p-value for 8 bits, entropy for 16 bits, χ^2-test p-value for 16 bits). Note that the values of entropy are less useful for the subsequent analysis since they are usually very close to 8 and 16, and measure the deviation from the uniform distribution in a similar way. Uses about $8 \times$ `[-n]` bytes; so `-n` parameter should be much bigger that 64 (for bytes) and 32768 (for 16-bit integers).

10 `fftest`

This is a Fourier transform (spectral) test from NIST, as described in [37]. This description includes corrections from citekim-umeno-hasegawa. In this test n integers from the generator are converted into $32n$ bit sequence; this sequence is interpreted as array of ± 1, and discrete Fourier transform is applied. We assume that n is a power of 2, so the simple algorithm for Fourier transform can be used (taken from `gsl`, GNU scientific library). In this way we get n complex numbers (since we start with real numbers, only $n/2$ first number are used). We see how many of then have absolute value that exceeds some threshold (namely, $\sqrt{\ln(1/0.05)n}$), the description in NIST document claims that expected value of this number is $0.05n/2$, and the distribution is close to binomial distribution for $n/2$ independent experiments with 0.05 probability. No derivation for this approximation is given (not to speak about error bounds). It seems that this approximation is quite rough. Figure 7.5 shows the experimental distributions for comparison of some random number generator and its xor with reference generator. One can see that both curves are quite close to each other and quite far from the diagonal (that should appear for uniform distributions). The jumps happen since there are only a few values of the test statistics (the number of values exceeding the thresholds). Let us stress again that the approximation errors do not destroy the validity of test results when using two-sample robust tests.

Test dimension is 1, uses $4 \times$ `[-n]` bytes; the `-n` parameter should be a power of 2.

11 `rank32x32`

This and the following tests (both are implemented in `test_rank.c`) check ranks of some matrices . This test has dimension 1. It forms [`-n` option] matrices 32×32 (each uses 32 integers). Ranks of these matrices are computed and we count the frequencies of ranks 32, 31, 30, [29 and smaller]. (This grouping is reasonable since matrices with ranks less than 29 are rare.) Then the χ^2-statistic is computed and converted to a (presumably) uniform distribution. The same computations are done in `dieharder` (while `diehard` used 31×31 matrices). Uses $128 \times$ `[-n]` bytes.

12 `rank6x8`

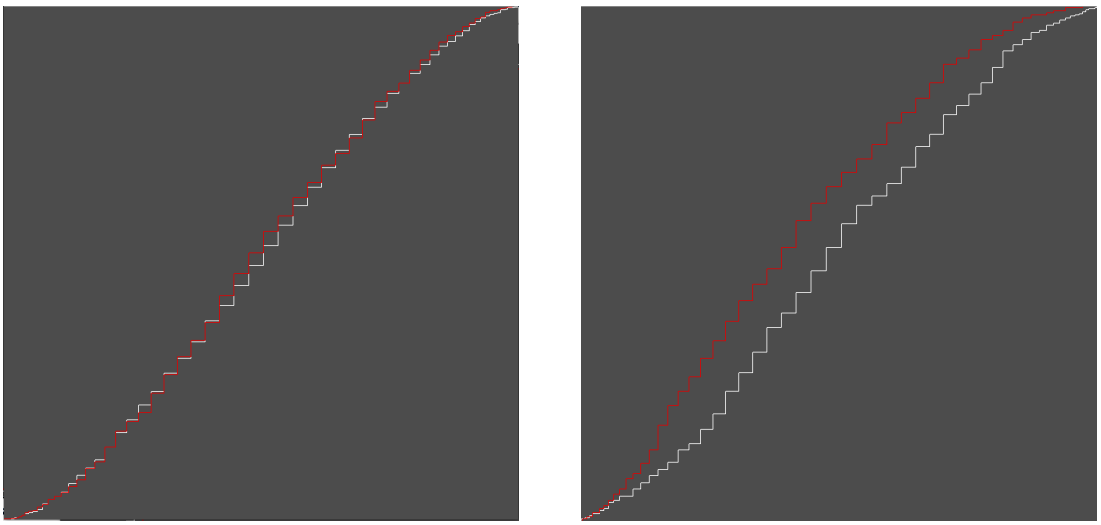


Figure 7.5: *Left*: the distributions of presumably uniform random variables obtained by Fourier transform test, command `rtest -x -f a10bis.raw -e ../bitbabbler/raw_4_sources -p 1000 -q 1000 -n 256 -t 10 -d 1 -o outfft`. *Right*: a visible difference between test and etalon samples for some bad generator.

This test has dimension 25 (since there are $32 - 8 + 1 = 25$ possible position of a 8-bit factor in a 32-bit word). It takes $6 \times$ `[-n]` integers; for each group of 6 integers it form 25 matrices 6×8 corresponding to 25 positions. Ranks are computed, and for each position a distribution of matrices according to ranks (6, 5, [4 or less]) is analyzed via χ^2 statistics and transformed into a presumably uniform variable (separately for each position). This follows `diehard` and `dieharder` with some corrections, see the discussion in Section 7.1. Uses $24 \times$ `[-n]` bytes.

13 `bitstream_o`

This and the next tests (overlapped and non-overlapped version of a bitstream test from `diehard` and `dieharder`) are implemened in file `test_bitstream.c`. This test considers the input stream of bits (32 integers are concatenated as written from left to right; the bits are also written from left to right). The test uses $2^{21}+19$ bits (and discard unused bits in the last integer). It consideres all 20-bit factors in this long string and counts the missing ones, i.e., 20-bit strings that do not appear as a factor. This number is transformed into a presumably uniformly distributed variable (following `dieharder`) using empirically observed normal distribution with mean 141909 and $\sigma = 428$. The dimension of the test in 1, the `-n` option parameter should be 2097171 (number of bits used), or about 256 Kbytes per test value.

14 `bitstream_n`

Similar to the preceding test, but non-overlapping 20-factors are used (the input bit stream is cut into 20-bit pieces). Uses $2^{21} * 20 = 41943040$ bits, and this number should be the value of `-n` parameter; dimension is 1. Since the words are now independent, the distribution is different, and the `dieharder` uses the same mean 141909 with smaller $\sigma = 290$. About 5 Mbytes per test value

15 `lz_split`

Lempel-Ziv test as it was described in the original version of [37]. The sequence of bits is split into words according to the following Lemper–Ziv rule: the next word is chosen as *the minimal prefix of the non-processed yet part that is not a prefix of already chosen word.* The algorithm counts the number of full words produced when analyzing $32 \times$ [`-n` parameter]

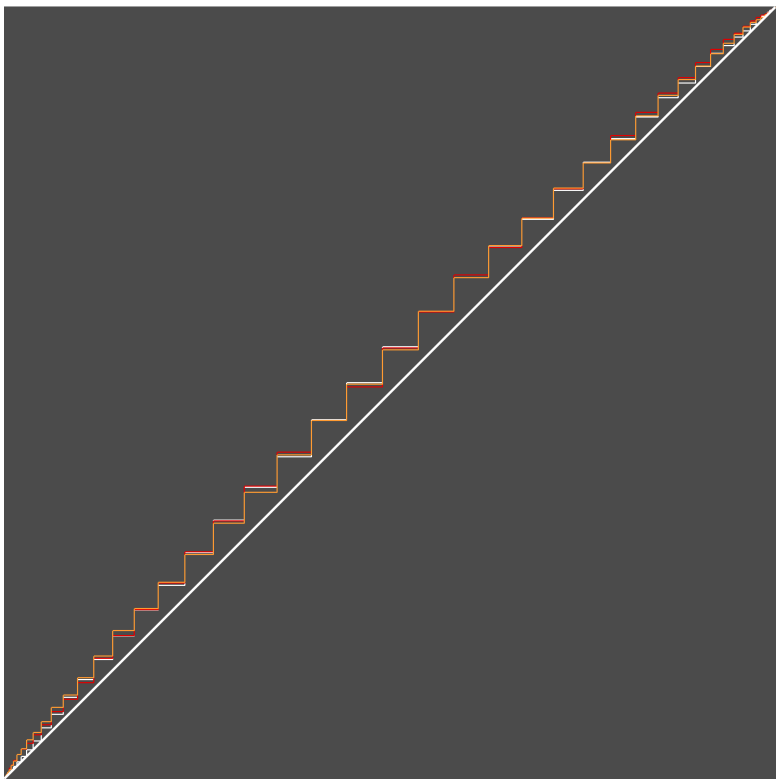


Figure 7.6: Empirical distributions. Each experiment involved 3000 Lempel–Ziv decompositions of a bit sequence of length 10^6; several experiments are shown with different colors. According to the original version of the test, the empirical distribution functions should be close to a diagonal line. One can see both effects mentioned in [23]: first, the number of different values is small, and this creates the "ladder" structure; second, one can also suspect systematic deviations from the straight line (added for comparison).

input bits. The NIST standard suggest some approximation for the resulting distribution in case of 10^6 bits (`-n 31250`), so in this case we convert the values to presumably uniform distribution using this approximation. As we have mentioned in Section 2.3.3, the problems with this approximation were found in [23]. They can be illustrated by a distribution obtained when applying this test (with NIST parameters) may times to several hardware generators (Figure 7.6). Still it can be safely used in the two-sample robust version.

16 `birthdays`

A "year" consists of 2^{24} days; we choose 1024 random days ("birthdays") and compute 1024 intervals between birthdays. Some intervals may coincide, and we count the number of coincidences (i.e., the difference between 1024 and the number of different intervals). We repeat this experiment several times (`-n` option) and compare the resulting distribution with presumed approximation (Poisson distribution with $\lambda = 16$), getting one p-value that is considered as the output of the test. More precisely, we use first 24 bits from the 32-bit integers from the input sequence; this gives us one p-value, than next 24 bits from the same 32-bit integers, and so on (32 cyclic shifts, as described — but not implemented — in `dieharder`). The test dimension is 32, and it uses 1024 times [`-n` option] integers ($4096 \times$ [`-n`] bytes) for each set of 32 values. Note that the test is robust (as any other two-sample test) but its usefulness is not clear (due to the systematic deviation of the observed distribution from the presumed one, see Section 7.1, Figure 7.2).

There are also some shell scripts that run these tests for suitable parameters (adjusted for 1 Mb,

10 Mb, 100 Mb, 1 Gb, 10 Gb files). Some tests take a long time, so it may have sense to delete some tests from these files if they take too much time. Two arguments are required: the name of the tested file and the name of the reference file.

A short reminder to be taken into account when interpreting the test values:

- The p-values produced by these files are reliable (not based on any unproven assumptions about the tests and/or approximations[4]). For any given test this means that for every $p > 0$ and for every reference file the probability of the event "the test with random bits in the tested file will produce value at most p" is at most p. Note also that the tests never "rewind" the file (using the same bits several times): if there is not enough bits in the file, no value is produced.

- There is no claim of uniform distributions for the test output (for a random bits input), so *output values close to* 1 *do not indicate any problems.*

- The tests (some individual tests and the shell scripts) produce many numbers at once. These number are not independent. So one should *not* multiply them to get a more impressive number. Moreover, one should apply the Bonferroni correction (that makes the conclusion less impressive): If for a chosen family of tests *one* of m number (where m is the total number of outputs, depending on the chosen family) is p, the resulting p-value is mp. For example, if one of the 40 test values is less than 0.01, one should not become suspicious about the generator, since $40 \cdot 0.01$ is 0.4, definitely not enough to start worrying.

- On the other hand, if you repeat some test several times (say, using `-r` option, or just applying it to different runs of the same random bit generator), the output values *are* independent. This can be used to amplify the non-randomness conclusions. For example, if a test (or a group of tests) fixed before the experiment produces 37 values, we made 15 runs with the same generator (and the same or different reference files, fixed in advance), and it turned out that one of 37 values is less than 0.001 for all the experiments, the correctly computed p-values is $37 \cdot (0.001)^{15}$. One should distinguish this from the other case: for each of 15 experiments there is some output value less than 0.001, but this value can be produced by different tests in different experiments. In this case the correct computation of p-value gives $(37 \cdot 0.001)^{15}$.

- The recommended approach when discrediting some generator: make some preliminary experiments to determine which tests look more suspicious. Then try the selected test on the fresh input from the generator and report the p-value obtained for this test. But be careful if a small p-value appears after several attempts to discredit the generator: a honest researcher should multiply the resulting p-value by the number of attempts.

7.4 Spectral tests

The spectral tests currently involve a lot of parameters, so they are not yet included in the robust testing suite. Currently we do *not* have examples where they beat all the other tests in the sense that they find some deficiency in a generator that cannot be found by other tests. Still it would be desirable to have more robust tests especially if they use different approaches.

The call

```
example -g gap -e err -s size -d deg -o iter -n number -l -u -h] filename
```

[4]The only compromise here happens when we use floating point numbers for computing the Kolmogorov–Smirnov value; looking at the output of `ks2compare` utility, we may hope that this does not create any problems for samples of size up to 5000, but one should be careful before pronouncing the final verdict and use option `-k` that asks for an exact computation.

The utility `example` analyses one file with name `filename` (so this utility should be combined with the `xoring` made separately to get a robust test). It uses bytes from this file to construct a sequence of graphs. The size of the graph is determined by option parameter `-s size`; it should be a power of 2 not exceeding 65536. Each (undirected) graph is constructed by emitting `deg` random edges from each vertex. (Note that the resulting undirected graph may not have constant degree, but the average degree is close to $2 \cdot$ `deg`.) Then the algorithm from [43] is used to compute the second eigenvalue for the resulting graph. This is an iterative algorithm that has some parameters. It constructs some lower and upper bounds for the second eigenvalue and stops when the difference between them becomes smaller than `gap` (`-g` parameter). Also it uses some internal randomness (now crudely replaced by some fixed pseudo-random generator[5]. Also the algorithm uses some upper bound for the number of iterations that can be specified by option `-o`. The option `-n` parameter specifies the maximal number of graphs that should be constructed (if there is enough bits in the input file `filename`).

The output of the utility is a sequence of lines. Each of them corresponds to one graph, and the contents of the lines is determined by the options `-l,-u,-h` (at least one of them should be present). The output line contains

- the lower bound for the second eigenvalue, if `-l` is present;
- the upper bound for the second eigenvalue, if `-l` is present;
- some hash value (useful for tie breaking), if `-h` is present.

Also an option `-v` (verbose) can be used to provide more information.

7.5 Other tools

There are few simple tools that visualize data from `.wav` and raw data files and perform conversion between formats.

- `readwav [-m|-s|-y] filename.wav` shows data (signal readings interpreted as signed integers) from a `.wav` file. For a stereo file `[-s]` it takes values from both channels and draws a pair. The option `[-y]` does the same but corrects the 1-sample lag in the right channel (that happens in Yamaha MW10 and Behringer 1204USB mixers we used); mono signal has identical readings in both channels after the correction. The `[-m]` option splits the movo `.wav` file into pairs of sample values (signed integers) and draws the corresponding points. The number of bits per value should be 8, 16 or 24.
- `getdatawav [-m|-s|-y] source.wav dest1 [dest2]` converts standard `.wav` file into raw format, deleting the header and splitting stereo data into two separate files. Options `-s` and `-y` require two destination files (`dest1` is for the left channel, `dest2` is for the right channel). Option `-m` is used for mono files, and just deletes the header. The input file should have resolution 8, 16, or 24. The order of bytes remains unchanged.
- `drawpair -b num_bytes -l|-m -c` draws pairs obtained from two files (in parallel) or from one file (`-c`; two consecutive data form a pair). The `num_bytes` parameter gives the number of bytes interpreted as a signed integer; `-l`/`-m` option (one of two should be present) says whether the integers are assumed to appear in `l`east or `m`ost significant byte first.

[5]Since our `xor`-trick guaranteed correctness of p-values independently of the function used for testing, we decided to use some simple pseudo-random sequence with fixed seed.

7.6 Two-source extractor tools

We implemented the two-source extractor described in Section 5.5. We used the finite field of size 2^{1024} whose elements are represented as polynomials in $\mathbb{F}_2[x]$ considered modulo

$$p(x) = x^{1024} + x^{19} + x^6 + x + 1.$$

This polynomial is suggested as low-weight irreducible polynomial in [51]. "Low weight" means that only a few coefficients are non-zero; this allows us to simplify the division modulo this polynomial. We used the natural basis

$$1, x, x^2, \dots, x^{1023}$$

in the field (the quotient of $\mathbb{F}_2[x]$ modulo $p(x)$); the multiplication by the basis elements can be obtained by repeated multiplications by x. And a multiplication by x is just a shift in the array of coefficients, followed by a reduction modulo $p(x)$. This reduction is applied to a polynomial of degree 1024; we have to change coefficients in the degrees involved (20, 7, 2, 1). This simple operations are implemented in file `extract.c`; we produce a bit string of length 128 from two strings of length 1024 (the field elements).

The file `extractor.c` calls this extraction function. Since in our experiments we used `.wav` files where most significant bits (of a 16-bit words from analog-digital converter are often zeros), we decided to use `xor` of two halves of an 16-bit word to get 8 bits that have (hopefully) reasonably high min-entropy. The command

```
extractor file1 file 2
```

applies this operation (`xor`ing of two bytes in a 16-bit integer) to each of files `file1`, `file2`, applies the extractor function as described, and send the resulting bits to standard output.

In this way from 2048 + 2048 bits from both files we generate 1024 + 1024 bits (by `xor`ing), and then generate 128 output bits, so the production rate is 1 : 32 (one output bit for 32 input bits).

Chapter 8

Conclusions

Let us summarize some observations made in this report.

- The main goal of this project was to review the current situation in hardware random number generation (devices, tests, standards) taking into account the corresponding theory (algorithmic information theory, pseudorandom generators, extractors). It turns out to be quite unsatisfactory: the basic principles are not always understood by the authors of the standards and developers of tests, as well as the device makers. As a result, tests are unreliable in many ways and testing results and claims made by the manufacturers are often meaningless.
- The reliable test methodology can be developed based on two-sample Kolmogorov – Smirnov tests. The general scheme is easy to implement. We adapted a dozen of often used tests into this framework and showed that this methodology works. Still one should add more tests to it, so that existing tests (and new ones that are developed by different authors) could be used in a reliable way to produce real p-values, not just some numbers with no guarantees.
- There are several physical processes that can be used for random number generators. The devices we tested used Zener diodes (most of them), analog multipliers (Infinite Noise, Bitbabblers) and photon reflection (Quantis). None of them give a bit sequence that can pass the basic tests without some "conditioning", but all three approaches seem to be suitable for high quality generators after some post-processing, and there are no obvious reasons to prefer expensive "quantum randomness" (whatever it is).
- One can use postprocessing based on functions used in cryptography. There are no theoretical reasons or assumptions that guarantee anything about this type of post-processing, but the currently known tests do not show obvious weaknesses in this approach. Still the possibility of some hidden trapdoor remains.
- Instead, one can use much simpler methods based on `xor`ing several random streams (generated by different sources or in different time) or using other linear tranformations of bits. Known theoretical results about that are not really applicable to practical situations, but this does not prevent these methods to work quite well in practice. Simple postprocessing of this type often gives sequences that do not exhibit suspicious behavior with known tests.
- There are also more complicated (and not yet used in practical generators) tools for randomness extraction based on two-source extractors. They are significantly slower than `xor`ing, but still are quite practical and can be easily implements in software.
- All these approaches can easily provide streams of random bits with rates 100–1000 kilobits per second using easily available devices (few hundred euros) and some additional postprocessing. The choice of the device and postprocessing methods depends on the speed requirements and

the paranoia level. Most paranoid people can achieve rates about 10–100 kilobits per second without using any commercial random bits generator, just by combining zener diodes and audio digitizing equipment.

- One can obtain rates 10–100 megabits per second or higher using several devices in parallel or more expensive (and sometimes less transparent) devices. Faster random bit production would require serious parallelism or much more expensive devices.

Acknowledgments

The authors are grateful to their colleagues who worked on the RaCAF project (Laurent Bienvenu, Bruno Durand, Fabien Givors, Ruslan Ishkuvatov, Gregory Lafitte) and to the members of the ESCAPE team (LIRMM, Montpellier), Kolmogorov Seminar (Moscow), Theoretical Computer Science Lab (Moscow, HSE), and all others from whom we learned about randomness, including Vladimir Uspensky, Leonid Levin, Alexander Zvonkin, Nikolay Vereshchagin, Vladimir Vovk, Vladimir Vyugin, Péter Gács, Wolfgang Merkle, Paul Vitányi, Daniil Musatov, Andrei Rumyantsev, Mikhail Andreev, Gleb Novikov, Bruno Bauwens, Konstantin Makarychev, Yury Makarychev, Ilya Razensteyn, Gleb Posobin, Alexey Vinogradov. The work was supported by ANR RaCAF grant ANR-15-CE40-0016-0. We are also grateful to the device makers who kindly agreed to answer some of our questions.

Bibliography

[1] ASA Publication: Ronald L. Wasserstein, Nicole A. Lazar, Editorial: The ASA's Statement on p-Values: Context, Process, and Purpose, *The American Statistician*, **70**(2), 129-133 (2016), http://dx.doi.org/10.1080/00031305.2016.1154108

[2] Andrian Belinski, *Method and apparatus of entropy source with multiple hardware random noise sources and continuous self-diagnostic logic*, US9477443B1 patent, https://patents.google.com/patent/US9477443.

[3] Daniel J. Benjamin et al. (72 authors), Redefine statistical significance, *Nature human behaviour*, **2**, 6–10 (2018)

[4] Laurent Bienvenu, Péter Gács, Mathieu Hoyrup, Cristobal Rojas, Alexander Shen, Algorithmic tests and randomness with respect to a class of measures, *Proceedings of the Steklov Institute of Mathematics*, **274**, 34–89 (2011). See also http://arxiv.org/abs/1103.1529.

[5] Lenore Blum, Manuel Blum, Mike Shub, A Simple Unpredictable Pseudo-Random Number Generator, *SIAM Journal on Computing*, **15**(2), 364–383 (1986), https://doi.org/10.1137/0215025

[6] Manuel Blum, Silvio Micali, How to Generate Cryptographically Strong Sequences of Random Bits, *SIAM Journal on Computing*, **13**(4), 850–864 (1984), https://doi.org/10.1137/0213053 (preliminary version was presented at FOCS 1982 conference).

[7] Émile Borel, *Le Hasard*, Paris, Librarire Félix Alcan, 1920. 312 pp.

[8] Jan Bouda, Matej Pivoluska, Martin Plesch, Improving the Hadamard extractor, *Theoretical Computer Science*, **459** (2012), 69–76, http://dx.doi.org/10.1016/j.tcs.2012.07.030.

[9] Robert G. Brown, Die*Harder:* A Gnu Public License Random Number Generator, version 3.31.1. http://www.phy.duke.edu/~rgb/General/dieharder.php (2006–2018).

[10] A. Philip David, Steven de Rooij, Glenn Shafer, Alexander Shen, Nikolai Vereshchagin, Vladimir Vovk, Insuring against loss of evidence in game-theoretic probability, *Statistics and Probability Letters*, **81**, 157–162 (2011), https://doi.org/10.1016/j.spl.2010.10.013

[11] Robert Davies, *Hardware random number generators*. Presented at *15th Australian Statistics Conference*, July 2000, and *51st conference of New Zealand Statistical Association*, September 2000, http://robertnz.net/hwrng.htm.

[12] Yevgeniy Dodis, Ariel Elbaz, Roberto Oliveira, Ran Raz, Improved Randomness Extraction from Two Independent Sources. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques. RANDOM 2004, APPROX 2004*. Lecture Notes in Computer Science, **3122**, Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-27821-4_30.

[13] Daniela Frauchiger, Renato Renner, Matthias Troyer, *True randomness from realistic quantum devices*, arxiv preprint, https://arxiv.org/pdf/1311.4547.pdf

[14] Yuri Gurevich, Grant Olney Passmore, Impugning Randomness, Convincingly, *Studia Logica*, **100**(1–2), 193–222 (April 2012), https://link.springer.com/article/10.1007/s11225-012-9375-1, see also https://arxiv.org/pdf/1601.00665.pdf, https://www.cl.cam.ac.uk/~gp351/Gurevich-Passmore-IRC.pdf.

[15] Yuri Gurevich, Vladimir Vovk. *Test statistics and p-values*. Working paper #16, On-line compression modelling project (new series), http://www.alrw.net/articles/16.pdf. See also https://arxiv.org/pdf/1702.02590.pdf.

[16] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, Michael Luby, A Pseudorandom Generator from any One-way Function, *SIAM Journal on Computing*, **28**(4), 1364–1396 (1999, original conference papers from 1989 and 1990), https://doi.org/10.1137/S0097539793244708.

[17] Miguel Herrero-Collantes, Juan Carlos Garcia-Escartin, Quantum Random Number Generators, *Review of Modern Physics*, **89**, 015004 – Published 22 February 2017, https://journals.aps.org/rmp/pdf/10.1103/RevModPhys.89.015004. See also https://arxiv.org/abs/1604.03304v2.

[18] S. Hoory, N. Linial, A. Wigderson, Expander graphs and their applications, *Bulletin of the American Mathematical Society*, **43**(4), 439–561 (2006)

[19] Darren Hurley-Smith, Julio Hernandez-Castro, *Quam Bene Non Quantum: Bias in a Family of Quantum Random Number Generators*, https://eprint.iacr.org/2017/842.

[20] Darren Hurley-Smith, Julio Hernandez-Castro, Quantum Leap and Crash: Searching and Finding Bias in Quantum Random Number Generators, *ACM Transactions on Privacy and Security*, **23**(3). 1–25 (2020), ISSN 2471-2566, see also https://kar.kent.ac.uk/81957/11/Quantum_Leap_TOPS_Submission_FINAL.pdf

[21] David Johnston, *Random Number Generators — Principles and Practices. A Guide for Engineers and Programmers*, Walter de Gruyter, Berlin/Boston, ISBN 978-1-5015-1513-2, https://doi.org/10.1515/9781501506062-201, 436 pp.

[22] Wolfgang Killmann, Werner Schindler, *A proposal for: Functionality classes for random number generators, Version 2.0, 18 September 2011*, AIS20/AIS31, https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_20_Functionality_classes_for_random_number_generators_e.pdf

[23] Song-Yu Kim, Ken Umeno, Akio Hasegava, *Corrections of the NIST Statistical Test Suite for Randomness*, https://eprint.iacr.org/2004/018.pdf.

[24] Alexander Kireev, *On the falsified results of the "referendum" in Sevastopol* [In Russian], blog post in LiveJournal, https://kireev.livejournal.com/1095568.html.

[25] Donald Knuth, *The Art of Computer Programming. Volume 2. Seminumerical Algorithms*. Second edition, Reading et al., Addison–Wesley, 1981. ISBN 0-201-03822-6.

[26] Alexey Kupriyanov, Gauss against Churov: preliminary conclusions [In Russian], *Troitskii variant* newspaper, 08.05.2019, https://trv-science.ru/2018/05/08/gauss-protiv-churova-promezhutochnyj-itog/.

[27] Patrick Lacharme, Post-Processing Functions for a Biased Physical Random Number Generator. In: *Fast Software Encryption. FSE 2008*. Lecture Notes in Computer Science, **5086**. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-71039-4_21

[28] George Marsaglia, A Current View of Random Number Generators, *Computer Science and Statistics, Sixteenth Symposium on the Interface*, Elsevier, North-Holland (1985), 3–10.

[29] George Marsaglia, *Random Numbers CDROM including the Diehard Battery of Tests of Randomness*, 1995, was available at `http://stat.fsu.edu/pub/diehard/`; now (2019) still avaliable as snapshots from `https://web.archive.org`. Contains the preprint version of [31, 28]

[30] George Marsaglia, Wai Wan Tsang, Some difficult-to-pass tests of randomness, *Journal of Statistical Software*, 7(3), 2002, `https://www.jstatsoft.org/article/view/v007i03`.

[31] George Marsaglia, Arif Zaman, Monkey Tests for Random Number Generators, *Computers and Mathematics with Applications*, **26**(9), 1–10 (November 1993).

[32] George Marsaglia, Xorshift RNGs, *Journal of Statistical Software*, **8**, issue 14 (July 2003), `http://dx.doi.org/10.18637/jss.v008.i14`

[33] George Marsaglia, `https://github.com/omec-project/il_trafficgen/blob/master/pktgen/app/xorshift64star.h`

[34] Ueli M. Maurer, A Universal Statistical Test for Random Bit Generators, *Journal of Cryptology*, **5**(2), 89–105 (1992), `https://link.springer.com/article/10.1007/BF00193563`.

[35] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, **8** (1): 3–30 (1998)

[36] FIPS PUB 140-2. *Security requirements for cryptographic modules*. Federal information processing standards publication (Supercedes FIPS PUB 140-1, 1994 January 11). Issued May 25, 2001. Information Technology Laboratory, National Institute of Standards and Technology, `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`

[37] NIST Special Publication 800-22: Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, Revised: April 2010, Lawrence E. Bassham III, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce (NIST), Revision 1a (2010), 131 pp., `https://www.nist.gov/publications/statistical-test-suite-random-and-pseudorandom-number-generators-cryptographic`. Previous version seems to be unavailable at this site, but the review of Elaine B. Barker, *ITL Bulletin, December 2000, 3 pp.* is available at `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=151231`. The Lempel–Ziv test, criticised in [23], was there (#10) according to the review; it is missing in the updated version.

[38] NIST Special Publication 800-90A: Elaine Barker, John Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce (NIST), June 2015, `https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final` (previous version: January 2012).

[39] NIST Special Publication 800-90B: Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry McKay, Mary Baish, Michael Boyle, *Recommendation for the Entropy Sources Used for Random Bit Generation*, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce (NIST), January 2018, `https://csrc.nist.gov/publications/detail/sp/800-90b/final`.

[40] NIST Special Publication 800-90C (Second Draft): Elaine Barker, John Kelsey, *Recommendation for Random Bit Generator (RBG) Constructions*, April 2016, https://csrc.nist.gov/CSRC/media/Publications/sp/800-90c/draft/documents/sp800_90c_second_draft.pdf.

[41] NIST Special Publication 800-107: Quynh Dang, Recommendation for Applications Using Approved Hash Algorithms Revision 1 (August 2012), https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf.

[42] Gleb Novikov, Randomness deficiencies, *Computability in Europe, 2017: Unveiling Dynamics and Complexity*, Lecture Notes in Computer Science, **10307**, Springer, 338-350, https://doi.org/10.1007/978-3-319-58741-7_32.

[43] Yann Ollivier, *Spectral gap of a graph*, a public domain code for finding the spectral gap of a graph, http://www.yann-ollivier.org/specgraph/specgraph (2004)

[44] Yuval Peres. Iterative von Neumann's procedure for extracting random bits, *The Annals of Statistics*, **1**(1), 590–597 (March 1992), https://www.jstor.org/stable/2242181

[45] Melissa E. O'Neill, Melissa,(5 September 2014). *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation* (Technical report). Harvey Mudd College. HMC-CS-2014-0905. See also https://www.pcg-random.org/download.html

[46] Takuji Nishimura and Makoto Matsumoto. A C-program for MT19937, with initialization improved 2002/1/26. http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/MT2002/CODES/mt19937ar.c

[47] R, a freely available language and environment for statistical computing, see *The Comprehensive R Archive Network*, cran-r.project.prg (January 18, 2020, version R-4.0.3), file R-4.0.3/src/library/stats/src/ks.c, functions psmirnov2x and pkstwo.

[48] RAND corporation, *A Million Random Digits with 100,000 Normal Deviates*, Free press, 1955. (Reissued in 2001 as ISBN 0-8330-3047-7.)

[49] Omer Reingold, Salil Vadhan, Avi Wigderson, *A Note on Extracting Randomness from Santha–Vazirani Sources*, report available from Reingold, https://omereingold.files.wordpress.com/2014/10/svsources.pdf.

[50] Miklos Santha, Umesh V. Vazirani, Generating Quasi-random Sequences from Semi-random sources, *Journal of Computer and System Sciences*, **33**, 75–87 (1986), https://doi.org/10.1016/0022-0000(86)90044-9.

[51] Gadiel Seroussi, *Table of Low-Weight Binary Irreducible Polynomials*, Hewlett–Packard Computer Science Laboratory technical report, HPL-98-135 (August 1998), https://www.hpl.hp.com/techreports/98/HPL-98-135.pdf

[52] Alexander Shen, Around Kolmogorov complexity: Basic Notions and Results, *Measures of Complexity. Festschrift for Alexey Chervonenkis*, Springer (2015), 75–116, https://link.springer.com/chapter/10.1007/978-3-319-21852-6_7, see also https://arxiv.org/abs/1504.04955v1.

[53] Alexander Shen, *Election and statistics: the case of "United Russia", 2009–2018*, preprint, https://arxiv.org/abs/1204.0307.

[54] Alexander Shen, *Making randomness tests more robust*, HAL archive, 2018, https://hal.archives-ouvertes.fr/hal-01707610.

[55] Alexander Shen, *Randomness Tests: Theory and Practice*. In: Fields of Logic and Computation III, Lecture Notes in Computer Science, 12180, Springer-Verlag, 258–290 https://link.springer.com/chapter/10.1007%2F978-3-030-48006-6_18 (2020). See also: https://hal-lirmm.ccsd.cnrs.fr/lirmm-03065320

[56] Alexander Shen, Vladimir A. Uspensky, Nikolai K. Vereshchagin, *Kolmogorov Complexity and Algorithmic Randomness*, American Mathematical Society (2017), http://www.lirmm.fr/~ashen/kolmbook-eng-scan.pdf.

[57] Richard Simard, Pierre L'Ecuyer, Computing the Two-Sided Kolmogorov–Smirnov Distribution, *Journal of Statistical Software*, **39**(11), 18 pp. (2011), available at https://www.jstatsoft.org/index.

[58] Tom Stoppard, *Rosencrantz and Guildenstern Are Dead*, a play (1966), ISBN978-0-8021-3275-8 (1971 edition).

[59] TrueRNG v3 documentation from Ubld.it, http://ubld.it/truerng_v3.

[60] Nikolay Vereshchagin, Alexander Shen, Algorithmic Statistics Revisited, *Measures of Complexity. Festschrift for Alexey Chervonenkis*, Springer (2015), 235–252, https://link.springer.com/chapter/10.1007/978-3-319-21852-6_17, see also https://arxiv.org/abs/1504.04950v2.

[61] Nikolay Vereshchagin, Alexander Shen, Algorithmic statistics: forty years later, *Computability and Complexity. Essays Dedicated to Rodney G. Downey on the Occasion of His 60th Birthday*. Springer (2017), Lecture Notes in Computer Science, **10010**, 669–737, see also https://arxiv.org/abs/1607.08077.

[62] Andrew C. Yao, Theory and application of trapdoor functions, *23rd Annual Symposium on Foundations of Computer Science* (FOCS), 80–91 (1982), http://ieeexplore.ieee.org/document/4568378/.