ACTA IMEKO
ISSN: 2221-870X
December 2020, Volume 9, Number 4, 128 - 135

On pseudorandom number generators

Daniel Chicayban Bastos[1], Luis Antonio Brasil Kowada[1], Raphael C. S. Machado[1,2]

[1] *Instituto de Computação, Universidade Federal Fluminense, Brasil*
[2] *Inmetro - Instituto Nacional de Metrologia, Qualidade e Tecnologia, Brasil*

ABSTRACT
Statistical sampling and simulations produced by algorithms require fast random number generators; however, true random number generators are often too slow for the purpose, so pseudorandom number generators are usually more suitable. But choosing and using a pseudorandom number generator is no simple task; most pseudorandom number generators fail statistical tests. Default pseudorandom number generators offered by programming languages usually do not offer sufficient statistical properties. Testing random number generators so as to choose one for a project is essential to know its limitations and decide whether the choice fits the project's objectives. However, this study presents a reproducible experiment that demonstrates that, despite all the contributions it made when it was first published, the popular NIST SP 800-22 statistical test suite as implemented in the software package is inadequate for testing generators.

Section: RESEARCH PAPER

Keywords: randomness; random number generator; true random number generator; pseudorandom number generator; statistical tests; TestU01; NIST SP 800-22; random sequence; state-of-the-art; crush

Citation: Daniel Chicayban Bastos, Luis Antonio Brasil Kowada, Raphael C. S. Machado, On pseudorandom number generators, Acta IMEKO, vol. 9, no. 4, article 17, December 2020, identifier: IMEKO-ACTA-09 (2020)-04-17

Section Editor: Francesco Bonavolonta, University of Naples Federico II, Italy

Received October 30, 2019; **In final form** May 15, 2020; **Published** December 2020

Funding: This research was partially funded by the SHCDCiber project.

Corresponding author: Daniel Chicayban Bastos, e-mail: dbastos@id.uff.br

1. INTRODUCTION

Unfortunately, the list of past incidents involving bad random number generation is not too modest. Bad randomness has been with us for as long as random number generation has been in use. Perhaps the oldest catastrophe is RANDU, from IBM's System/370, used in the 60s. '[The] very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists! [...] [It] fails most three-dimensional criteria for randomness, and it should never have been used.' [1]

In 1996, Netscape Communications failed to properly seed their random number generator during SSL handshaking: they used the current timestamp and the browser's `PID` and `PPID`. In UNIX systems, `PID` means process identifier and `PPID` is the process parent's `PID`. In the 32-bit Linux kernel version 2.5.68, every `PID` is an integer between 1 and 32767. In 64-bit systems, the value can get up to 2^{22}, approximately 4.2 million [2].

The seed *per se* was computed by the MD5 hash function, but since an adversary could have a precise measurement of the current timestamp and the universe of possible PID numbers was not large, it was possible to considerably reduce the set of possible seeds available to the generator. While Netscape thought they had 128 bits of security, they in fact had 47 bits [3].

In 2003, Taiwan launched a project offering its citizens a smart card with which they could authenticate themselves with the government, file taxes, etc. RSA keys were generated by the cards using built-in hardware random number generators advertised as having passed FIPS 140-2 Level 2 certification [4]. 'On some of these smart cards, unfortunately, the random-number generators used for key generation are fatally flawed and have generated real certificates containing keys that provide no security whatsoever.' As a result, a total of 184 distinct secret keys were found out of more than two million 1024-bit RSA keys downloaded from Taiwan's national key repository [5].

In 2008, a vulnerability in OpenSSL on Debian-based operating systems was caused by 'a random number generator that [produced] predictable numbers, [making] it easier for remote attackers to conduct brute force guessing attacks against cryptographic keys' [6].

In 2012, a survey of TLS and SSH servers was performed [7]. The entire IPv4 space was scanned, providing a macroscopic view of the universe of keys on the Internet. Unfortunately, many servers were powered by malfunctioning random number

generators. About 5.8 million distinct TLS certificates and 6.2 million SSH distinct keys were analysed from about 10.2 million hosts. It was found that 5.57 % of the TLS servers and 9.60 % of the SSH servers shared keys with at least one other server. Among TLS servers, at least 5.23 % were using default keys generated by the manufacturer that had never been changed by the user. It seems some 0.34 % generated the same keys as one or more hosts due to malfunctioning random number generators. As a result, about 64,000 (0.50 %) TLS private RSA keys and about 108,000 (1.06 %) SSH private RSA keys were factored by exploiting the fact that some of these keys shared a common factor with at least one other host due to entropy problems in random number generation.

As technology adoption advances, incidents become more frequent. In 2013, a component of Android responsible for generating secure random numbers contained a weakness that rendered all Android wallets generated until then vulnerable to theft [8], [9], [10]. In 2015, a flaw in FreeBSD's kernel made SSH keys and keys generated by OpenSSL vulnerable due to the possible predictability of a random number generator [11].

It is not absurd to assume that, in the same way that smart phones use the same libraries as servers and desktop systems, embedded systems and others will use the same or similar versions of these software due to their often low resource demands, generating more security concerns as stable implementations of verified software might be changed to fit in with the requirements of more constrained systems.

2. TERMINOLOGY

There are at least two types of random number generators, those called true random number generators (TRNGs) and those called pseudorandom number generators (PRNGs). The former is usually associated with a physical mechanism that produces randomness by way of a physical process 'such as the timing between successive events in atomic decay' [12]. A pseudorandom number generator is often an arithmetical procedure performed by a machine based on initial, hopefully random, information called a seed. If a pseudorandom number generator has enough desirable properties that it could be recommended for cryptographic applications, then the acronym CSPRNG is often used, meaning computationally secure pseudorandom number generator.

3. WHAT IS A RANDOM SEQUENCE?

Looking at probability theory textbooks, one sees they require the concept of randomness, but most expositions carefully dodge the difficulty of precisely defining what a random sequence is, which is required for the definition of the term 'probability'. Instead of making absolute assertions, the theory concerns itself with telling how much probability should be attached to statements involving events. In other words, the objective is to quantify, measure and compute, not to give meaning [1]. From the perspective of a formalist, this is not unusual, for pure mathematics is mostly concerned with the form of statements, not with their content, a view of pure mathematics that has been remarkably described by Bertrand Russell [13].

> Pure mathematics consists entirely of assertions to the effect that, if such and such a proposition is true of anything, then such and such another proposition is true of that thing. It is essential not to discuss whether the first proposition is really true, and not to mention what the anything is, of which it is supposed to be true. [...] Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.

One should recognize, however, that mathematical logic is not able to capture the whole of mathematics, as has been clear since the advent of Gödel's theorems. (For a precise definition of 'capture', see section 4.6, page 35 of Peter Smith's 'An Introduction to Gödel's Theorems', Cambridge University Press, 2007, ISBN: 978-0-521-85784-0.)

In the context of probability theory, *if* one has a random sequence, it can be used to draw samples from a population. Given these random samples, then 'such and such' deductions can be made. 'It is essential' not to discuss whether the sequence with which one began is really random. It is by hypothesis. And, finally, it is essential not to discuss what probability really is, since that would prompt us to discuss what randomness is. (The issue is discussed at length by von Mises in 'Probability, Truth and Statistics.' Richard von Mises, 1957. Dover Publications, Inc., 2nd edition, 1981. ISBN: 0-486-24214-5. On page 24, von Mises writes that '[t]he term "probability" will be reserved for the limiting value of the relative frequency in a true collective which satisfies the condition of randomness. The only question is how to describe this condition exactly enough to be able to give a sufficiently precise definition of a collective.' On page 12, he defines collective as 'a sequence of uniform events or processes which differ by certain observable attributes'. For example, 'all the throws of dice made in the course of a game form a collective wherein the attribute of the single event is the number of points thrown.')

However, if a probability is measured as a number, it can then be compared. For example, one can assert that a probability x is greater than a probability y, which is astoundingly useful.

Sequences that are ∞-distributed have been given serious consideration as candidates for a definition of a random sequence. To explain what ∞-distributivity is, it will help us to consider the particular case of binary sequences. A binary sequence is considered ∞-distributed if it is k-distributed for all natural numbers k. Intuitively, a k-distributed binary sequence is one in which the probability of a certain k-digit binary string appearing in the sequence is the same as any other. In other words, the sequence's probability distribution is uniform for k-digit binary strings.

In more precise terms, a binary sequence X_n is k-distributed for a certain k if

$$\Pr(X_n X_{n+1} \dots X_{n+k-1} = x_1 x_2 \dots x_k) = 1/2^k$$

for all binary k-digit numbers $x_1 x_2 \dots x_k$. For example, a binary 1-distributed sequence must satisfy $\Pr(X_n = 0) = 1/2$ as well as $\Pr(X_1 = 1) = 1/2$. One such sequence would be $0, 1, 0, 1, \dots$, since $\Pr(X_n = 0)$ is the limit of the sequence $1, 1/2, 2/3, 2/4, \dots$, which converges to $1/2$ [1]. Another example is $0, 0, 1, 1, 0, 0, 1, 1, \dots$ For a binary sequence to be 2-distributed, it would have to satisfy

$$\begin{aligned}
&\Pr(X_n X_{n+1} = 00) = 1/4,\\
&\Pr(X_n X_{n+1} = 01) = 1/4,\\
&\Pr(X_n X_{n+1} = 10) = 1/4,\\
&\Pr(X_n X_{n+1} = 11) = 1/4.
\end{aligned}$$

One can check that the sequence $0, 0, 1, 1, 0, 0, 1, 1, \ldots$ is also 2-distributed, but it is not 3-distributed. It is not 3-distributed because $\Pr(X_n X_{n+1} X_{n+2} = 000) = 0$ when it should be $1/8$. This suggests that for every periodic sequence, there is a natural number k such that the sequence is not k-distributed. Indeed, every periodic sequence of period p is not p-distributed [14]. A periodic 3-distributed binary sequence is not easily guessed, but one can check that the sequence

$$0,0,0,1,\ \ 0,0,0,1,\ \ 1,1,0,1,\ \ 1,1,0,1,\ \ 0,0,0,1, \ldots$$

is indeed 3-distributed [1].

An algorithm is not limited to producing periodic sequences – for example, an algorithm that produces the digits of π does not produce a periodic sequence – so the limitation of periodic sequences is no challenge to the idea that ∞-distributivity defines randomness. In fact, one of the formidable results of ∞-distributed sequences is that they can be produced by algorithms: one such algorithm was given in 1965 [14].

The weakness in taking the notion of ∞-distributivity alone as a definition for a random sequence appears when one considers the subsequences of an ∞-distributed sequence. If such sequences were random, it would be expected that any subsequence of a random sequence would also be random, but this does not always happen with ∞-distributed sequences. Given an ∞-distributed binary sequence X_n, one can construct a new sequence $Y_n = X_n$ except that $Y_{n^2} = 0$ for every index n. Clearly, Y_n is not random because, by construction, $Y_0, Y_1, Y_4, Y_9, \ldots, Y_{n^2} = 0$, yet Y_n is still ∞-distributed because setting squared elements to zero does not significantly change the probabilities required in the definition of k-distributivity [1]. That is, ∞-distributivity alone is too weak a definition. An apparently adequate definition is reached by making suitable restrictions to the rules governing which subsequences must be ∞-distributed. That is, not all subsequences of an ∞-distributed sequence X_n must be ∞-distributed for X_n to be qualified as random [1]. 'The secret is to restrict the subsequences so that they could be defined by a person who does not look at' X_n 'before deciding whether or not it is to be in the subsequence' [1]. To date, there does not seem to be any objection to this strategy.

4. WHAT IS A PSEUDORANDOM NUMBER GENERATOR?

Since algorithms cannot compute random sequences [15], they are left with at most producing pseudorandom number sequences displaying the desired statistical properties. A pseudorandom number generator, therefore, is an arithmetical procedure that produces a sequence of numbers that one hopes will pass sufficient statistical tests and thus appear random. It can be as simple as a function $f(x_n) = x_{n-1}^2 + 1 \bmod N$, for some fixed natural number N or as complex as Donald Knuth's 'super-random' number generator [1], shown as an illustration of how complicated algorithms do not necessarily provide any more randomness. (The function $f(x_n) = x_{n-1}^2 + 1 \bmod N$ is typically used in a method of integer factorisation known as Pollard's Rho. One interesting fact about the method is that, while polynomials like f do not have good statistical properties, the average number of steps taken by the procedure would considerably increase if a truly random sequence were used instead [16]. Sometimes true randomness is not desired.)

For illustration purposes, let us look at what a simple pseudorandom number generator looks like in the C programming language.

```
uint32_t y = 2463534242U; /* the seed */
uint32_t xorshift(void) {
  y = y ^ (y << 13);
  y = y ^ (y >> 17);
  y = y ^ (y << 5);
  return y;
}
```

This is George Marsaglia's Xorshift generator of 32 bits [17]. The variable `y` is a global variable in the `xorshift` procedure. The letter `U` at the end of the seed is just an indicator that that number is an `unsigned integer`. What this procedure does is multiply the arbitrarily set initial value 2463534242, the seed, to the number 2^{13} and add the result to `y`; that is, it adds the product to the initial value. The multiplication is done in fast computer arithmetic, namely, shifting `y` to the left by 13 bits because, in base 2 arithmetic, shifting a number to the left means adding zeros to the right of the number, which is the same as multiplying it by a power of 2. For example, by taking the number 5, which is 101 in base 2, and multiplying it by 2, one gets 10, which is 1010 in base 2. The second step divides `y` by 2^{17} and adds the result to `y`. The last step is similar. All such computations are reduced modulo 2^{32}. (ANSI X3.159-1989 asserts in section 3.1.2.5 that '[a] computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type'.) Looking at the numbers produced by this generator, they appear random to the naked eye, but the sequence does not pass even a modest contemporary battery of statistical tests.

5. DESIRABLE PROPERTIES OF GENERATORS

True random number generators have several disadvantages compared to a good pseudorandom number generator. For example, they are slower, more cumbersome to install and run, more costly and unable to reproduce the same sequence twice. (Reproducing the same sequence is important for repeating simulations and testing applications.) But a pseudorandom number generator does need a good seed, which true random number generators can provide [12].

When choosing a pseudorandom number generator, one must know what to look for. Some of the properties one can find in pseudorandom number generators, to name a few, are good statistical properties, good mathematical foundations, lack of predictability, cryptographic security, efficient time and space performance, small code size, a sufficiently long period and uniformity [18].

In the context of computer-generated randomness, good statistical properties are effectively what is meant by 'random' [18]. Mathematical foundations allow us to be sure a pseudorandom number generator has some desirable property, such as its period, which is defined as the length of the sequence of random numbers the generator can produce before needing to repeat itself. Having a long period is surely desirable. Uniformity is a property closely related to the period. After the generator has output all its period, each number produced should occur the same number of times, otherwise it is not uniform. If it is not uniform, it is biased. Uniformity alone, without a long

period, is certainly not desirable. Consider what happens as a uniform generator is consumed. As the end of its period draws near, its uniformity effectively allows one to predict more and more of its output, since all output must occur the same number of times [18].

> For example, let us consider a case where we can show that a generator must lack uniformity in its output. Consider a generator with b bits of state, but where one of the 2^b possible states is never used, (perhaps because the implementation must avoid an all-bits-are-zero state). The missing state would leave the generator with a period of $2^b - 1$. By the pigeonhole principle, we can immediately know that it cannot uniformly output 2^b unique b-bit values.

So, the period of a generator cannot be too short, lest it repeat itself while in use, which makes it statistically unsound. A large internal state implies the possibility of a longer period because it allows for more distinct states to be represented. Yet, in terms of period size, more is not always better. For example, if one is to choose between generators with period sizes of 2^{128} and 2^{256}, one should notice that it would take billions of years to exhaust the period of the 2^{128} generator, so picking the generator with period 2^{256} does not bring a relevant advantage. 'Even a period as "small" as 2^{56} would take a single CPU core more than two years to iterate through at one number per nanosecond.' [18]

Another valuable property is unpredictability. 'A die would hardly seem random if, when I've rolled a five, a six, and a three, you can tell me that my next roll will be a one.' [18] Still, pseudorandom number generators are deterministic, and their behaviour is completely determined by their input: they produce the same sequence given the same input. So, their randomness is only apparent to an observer who does not know their initial conditions. Though the deterministic nature of pseudorandom number generators might seem more like a weakness than a strength, it is valuable for reproducing the same sequence multiple times, which is required in a number of applications, from simulations and games to the mere testing of programs. To repeat a sequence generated by a pseudorandom number generator, one needs only save its initial conditions, usually just the seed for the produced sequence. To repeat a sequence from a true random number generator, the entire produced sequence would have to be saved.

It is not immediately obvious that a procedure computed by a machine can be unpredictable, but some pseudorandom number generators output a number while keeping another one hidden from the user. The hidden information is called the pseudorandom number generator's internal state. Predicting the pseudorandom number generator entails knowing the internal state.

Unpredictability is important for applications concerned with security because predicting a pseudorandom number generator allows for various types of attacks, including denial of service [19]. If a pseudorandom number generator leaks internal state information at each output, an adversary is able to little by little infer the complete internal state, at which point the generator becomes completely predictable, at least from that point in the sequence on, which is a flaw of Mersenne Twister [20].

Predictability can be considered in two directions: forwards and backwards. A generator is said to be invertible if, once its internal state is known, the random numbers it generated previously can be recovered. Being non-invertible is vital for applications that generate cryptographic keys; if the generator is invertible and its internal state is exposed at some point in time, adversaries will be able to recover all previously generated keys. So, cryptographically secure pseudorandom number generators are not invertible. Although some applications may not be designed with cryptography in mind, it is prudent to pick the safest generator that a project can afford [18].

> [Because] we cannot always know the future contexts in which our code will be used, it seems wise for all applications to avoid generators that make discovering their entire internal state completely trivial.

Speed is another important property, particularly when considering low resource systems. An application that is too dependent on a random number generator will be as slow as the generator used. Applications running in low-resource hardware will likely trade other properties for speed and space. Many generators with good statistical properties are slow, but there are some generators that have relatively good time performance while showing acceptable statistical properties. For example, `XorShift* 64/32` [17] has good performance and good statistical properties [18], although it is not safe for cryptographic applications.

Most generator implementations will take just a constant amount of memory to store their state, but considering the strict constraints some applications face, the size of these constants should also lead programmers to choose one over another. Space is also related to speed: considering all other things to be equal, a generator that is able to keep its internal state completely within a processor register should outperform a competitor that needs many more bytes of internal state to be kept in main memory [18].

There are also the space constraints of code size. Such space is most likely a constant, but constants do matter for applications running in low-resource hardware. The longer the code, the more likely it will include programming errors. Such errors can be particularly difficult to detect in the context of random number generators [18].

> From […] experience, I can say that implementation errors in a random number generator are challenging because they can be subtle, causing a drop in overall quality of the generator without entirely breaking it.

Another desirable property is seekability, the ability of a generator to skip ahead or jump back in the sequence. Since pseudorandom number generators are cyclic, by skipping a sufficient number of elements, one can get back to the starting number, meaning that the ability to seek ahead also implies the ability to seek backwards. Computationally secure pseudorandom number generators are designed not to be seekable, as it is not desirable to let an adversary read the sequence backwards, discovering which numbers might have been used in the past.

6. STATISTICAL HYPOTHESIS TESTING

Statistical theory allows us to posit a hypothesis H_0 about a random number generator and devise tests to provide empirical evidence of the validity of H_0. These tests, in turn, either give us more confidence in the hypothesis H_0 or leads us to reject it. A

statistical test for a random number generator is defined by a random variable X whose distribution under H_0 can be well approximated. When X takes the value x, define $p_R = \Pr(X \geq x \mid H_0)$ and $p_L = \Pr(X \leq x \mid H_0)$ as the left and right p-value, respectively. Such p-values measure how likely it is to find a certain sample of the random number generator given H_0 is true. If it turns out that very unlikely samples occur from the random number generator, that is then strong evidence the hypothesis H_0 is not true. In fact, when testing random number generators, if either the right or left p-value is extremely close to zero, then H_0 should be rejected. If a suspicious p-value is obtained, say near 10^{-2} or 10^{-3}, one can repeat the particular test a few more times, perhaps with a larger sample size, in the hope that more tests will clarify the result [12].

In the context of testing for randomness, H_0 is usually taken to mean that the sequence is random. For each specific test, a rule must be derived that allows one to reject or not to reject H_0. Taking H_0 to mean that the sequence generated is random, the test produces a statistic with a certain probability distribution of possible values. This probability distribution must be determined by mathematical methods. From this distribution, a critical value is chosen such that a critical region in the set of possible values is determined. The statistic is then computed from the sample and compared to the critical value. If the statistic falls in the critical region, H_0 is rejected; that is, one concludes the sequence produced by the generator is not random. Otherwise, H_0 is not rejected. If the generator produces a random sequence, then the computed statistic will have a very low probability of falling in the critical region, and if such an event occurs systematically, it provides strong evidence that the sequence is not random as assumed in H_0.

Although the probability for such an event may be very low, it is not null. Incorrectly classifying a sequence produced by a generator as not random is called a type I error. Much worse would be if H_0 is not rejected when the sequence produced by the generator is not random, an error that is called type II.

The probability of type I error is usually denoted by α and is called the level of significance of the test. The probability of type II error is usually denoted by β. The value of α can be arbitrarily chosen; that is, if a specific probability of type I error is desired, say 1 %, one can set $\alpha = 0.01$ for the specific test. Doing the same for type II error is not so easy. Recall that the probability distribution for the statistic produced by the test was determined assuming the generator does indeed produce a random sequence, that is, assuming H_0 is true. In the case of type II error, H_0 is not true, so the probability distribution of the statistic test is not known. Unless this probability distribution is known, β is not a fixed value because there is an infinite number of ways that a sequence can be non-random. Each different way determines a different β.

7. THE STATE-OF-THE-ART IN STATISTICAL TESTS

Under the framework of hypothesis testing, a series of tests can be devised to analyse samples of the random number generator. There is no maximum number of tests one can apply to a random number generator, and there is no maximum number of tests a random number generator can pass that will prove it to be truly random. It is also not possible to build an algorithmic random number generator that passes all statistical tests [12]. Nonetheless, the more tests one applies to a random number generator, the more confident one becomes of its quality.

Perhaps the first battery of tests was devised by Donald Knuth in 1969 [1]. In 1996, given the insufficiency of Knuth's tests, George Marsaglia published DIEHARD [21]. To supersede Marsaglia's tests, NIST, in the United States, published its own battery [22] in the year 2000, with its latest revision in 2010. Robert Brown published DieHarder in 2004. In 2007, Pierre L'Ecuyer and Richard Simard published TestU01, a C library with which C programmers can implement and test random number generators [23]:

> … empirical testing of random number generators is very important, and yet no comprehensive, flexible, state-of-the-art software is available for that, aside from the one we are now introducing. The aim of the TestU01 library is to provide a general and extensive set of software tools for statistical testing of random number generators. It implements a larger variety of tests than any other available competing library we know. [...] TestU01 was developed and refined during the past 15 years and beta versions have been available over the Internet for a few years already. It will be maintained and updated on a regular basis in the future.

TestU01's results were 'sobering' [14] for many 'respectable' and well-known random number generators [18]:

> [Pierre L'Ecuyer and Richard Simard] made a very significant contribution to the world of random-number–generator testing when they created the TestU01 statistical test suite. Other suites, such as [DIEHARD], had existed previously, but TestU01 (which included a large number of previously independently published tests, and applied them at scale) vastly increased the scope and thoroughness of the testing process.

The library comes with three predefined test batteries: `SmallCrush`, the small one, `Crush`, the medium-sized one and `BigCrush`. `SmallCrush` is the quickest, and it should finish in under a minute on most modern desktop computers. `Crush` can take a few hours, and `BigCrush` takes many hours or perhaps a day.

As for alternatives to TestU01, two other packages are competitors: PractRand 0.94 [24] and gjrand 4.2.1 [25], but neither has been formally published.

8. A NOTE ON USING THE TESTU01 LIBRARY

An inconvenience of TestU01 is that it is restricted to the C programming language. It is a C library, after all; it cannot run unless a programmer writes a program that takes advantage of the library. Besides, given that TestU01 is written in C, it would not be straightforward to use it from another programming language, as one would have to know how to access a C library from within the chosen programming language.

This inconvenience has been mitigated by `crush` [26], [27], a program capable of testing a random number generator against any of the three TestU01 batteries, given the data is available on a file on a disk or can be produced at run time. For example, suppose one would like to test one's local `/dev/urandom` against the largest TestU01 battery. It suffices to say to the shell:

```
%crush --battery big --name xyz < /dev/urandom
...
%
```

Similarly, if one has a program `p` that can produce its allegedly random data to the standard output in binary format, then `crush` can test such data against the small TestU01 library with a command such as:

```
%./p | crush --battery small --name my-prng
...
%
```

Due to the facilities of a UNIX-like system[1], `crush` eliminates the need to use the C programming language to take advantage of TestU01's default batteries.

9. A NOTE ON DEFAULT RANDOM NUMBER GENERATORS

If one is writing a new application that needs a random number generator, one should not just use random number generators offered by the system or by the chosen programming language. Most programming languages have adopted flawed generators. Java, for example, offers the package `java.Util.Random`, which is based on the pseudorandom number generator `drand48`. It failed five tests in `SmallCrush` in less than a minute [27].

The default pseudorandom number generator in both Python and PHP is `mt19937`, Mersenne Twister [20]. It passes `SmallCrush`, but actually fails the linear complexity test, which is not included in TestU01 even though it is a quick test to run and could have been part of the small battery. The number 19937 in Mersenne Twister's name is due to its huge period of size $2^{19937} - 1$. Despite having been a promising pseudorandom number generator, `mt19937` can be totally predicted after collecting a sample of size 624 [18].

In C++, besides `mt19937`, the standard library also offers `minstd` and `ranlux24`, two well-known generators, but `minstd` fails 9 tests out of 15 of the small battery, and `ranlux` is not much better [23]. C++ does offer, however, `ranlux48`, which passes `BigCrush` and can be used by applications that can afford the higher cost of such generator.

Exceptionally, some programming languages offer good random number generators as the default option. For example, the default pseudorandom number generator in the Racket programming language from the Lisp family is Pierre L'Ecuyer's `mrg32k3a` [28], which passes `BigCrush` [23].

For applications that require cryptography, a well-known computationally secure pseudorandom number generator is based on the stream cipher ChaCha20 [29]. ChaCha20 has replaced RC4 in OpenBSD starting at version 5.4, in NetBSD in version 7.0 and replaced SHA-1 in the Linux kernel since version 4.8. These events present evidence that ChaCha20 is currently well regarded.

10. ON THE INSUFFICIENCY OF THE NIST SP 800-22 SUITE

Notwithstanding the 'sobering' results of TestU01 [23], [18], it is not hard to find publications ignoring it [30], [31], [32] while giving attention to the software package provided by NIST SP 800-22. Enough flaws of this test suite have been previously reported [33]-[37], but we will now present one more result regarding the insufficiency of the NIST SP 800-22 statistical test suite implementation.

It is known that the Fibonacci sequence is not satisfactorily random to operate as a random number generator, but a 'much better' variation was proposed in 1958 by G. J. Mitchell and D. P. Moore, though it was never published. Using an output of 32-bit integers, let us call `mm32` this pseudorandom number generator defined by the sequence

$$X_n = (X_{n-24} + X_{n-55}) \bmod m$$

where $n \geq 55$, m is even, and $X_0, X_1, \ldots, X_{54}$ are arbitrary integers not all even. The constants 24 and 55 were chosen so that the least significant bits of the sequence, that is the sequence $X_n \bmod 2$, will have a period of length $2^{55} - 1$, implying the sequence X_n must also have a period of the same length [1].

Despite `mm32`'s period length, 'it is difficult to recommend [it] wholeheartedly [because] there is still very little theory to prove that [it does or does not] have desirable randomness properties; essentially all we know for sure is that the period is very long, and this is not enough.' [1] That is, from a theoretical perspective, very little is known about `mm32`, but, assuming TestU01 has a correct implementation of the statistical tests included in its batteries and PractRand implements `mm32` correctly[2], statistical evidence suggests `mm32` does not have desirable randomness properties.

Setting `mm32` with an initial value of 0 in PractRand's implementation and submitting it to the battery `SmallCrush` in TestU01, the battery reports that `mm32` fails the gap test [1] and the weight distribution test [38] after consuming approximately 6.7 gibibits[3] from the generator in less than 10 seconds on a certain system. PractRand reports that `mm32` fails the binary rank test [39], among other failures, after consuming 2 gibibits from the generator in less than 5 seconds, and gjrand reports `mm32` also fails the binary rank test consistently, among other failures, after consuming 8 gibibits from the generator in less than 15 seconds. Nevertheless, the NIST SP 800-22 statistical test suite does not notice anything unusual with `mm32` after consuming a total of 8 gibibits from the generator, that is, after consuming 32 samples of 256 mebibits[4] each in over 15 hours[5].

Could the NIST SP 800-22 statistical test suite reject `mm32` by considering larger samples? It was found that 32 gibibytes of memory are not enough to give the NIST SP 800-22 statistical test suite a sample size of 2 gibibits. When the size was reduced to 1 gibibit, the software received a recurrent UNIX `SIGSEGV` signal. In other words, it crashes at this sample length. The same crash can be reproduced with various small sequence lengths, such as 1031 and many smaller values. Also, on sample lengths of sizes such as 256 mebibits, the NIST SP 800-22 statistical test suite is not able to properly calculate a p-value for all of its tests with its default parameters. It was not easy to make sense of the p-values produced by the overlapping template matchings test on sample lengths such as 256 mebibits for any generator tested.

Regarding comparisons, notice each battery in each software package uses a different strategy, configurable in different ways, which makes comparison rather difficult. For example, despite the fact that `SmallCrush` consumed a total of approximately

[1] Notice Windows is sufficiently UNIX-like for the purposes of running `crush`, and a Win32 binary is available on `crush`'s homepage at https://bit.ly/319bg0H.

[2] In PractRand version 0.94, the implementation is found in `src/RNGs/other/fibonacci.cpp`.

[3] One gibibit is 2^{30} bits.

[4] One mebibit is 2^{20} bits.

[5] A more efficient implementation of the NIST SP 800-22 statistical test suite has been reported [40], but its implementation could not be located.

6.7 gibibits from `mm32`, each test individually consumed far less. For example, the weight distribution test used a sample length of 200,000 and took less than a second to run. With a generator producing random numbers at run time, the library by default decides not to restart the generator as it moves from one test to another.

11. CONCLUSIONS

Choosing a random number generator is no simple task. It should not be underestimated. Default pseudorandom number generators offered by popular programming languages usually do not offer enough statistical properties. It was argued that the NIST SP 800-22 statistical test suite, as implemented in the software package and last revised in 2010, is inadequate for testing random number generators. With `crush` [26], [27], testing a random number generator against the state-of-the-art in statistical tests is a trivial matter.

REFERENCES

[1] D. Knuth, The Art of Computer Programming, volume 2, 3rd edition, Addison-Wesley, Boston, 1997, ISBN: 978-0-201-89684-8.

[2] Linux Programmer's Manual, 2017, See '`man 5 proc`'.

[3] I. Goldberg, D. Wagner, Randomness and the Netscape browser, Dr Dobb's Journal-Software Tools for the Professional Programmer 21(1) (1996) pp. 66-71.

[4] National Institute of Standards and Technology (NIST), Security requirements for cryptographic modules, Federal Information Processing Standards Publication (FIPS PUB) 140-2 (May 2001). Online [Accessed 29 October 2020]. https://goo.gl/a0Sze

[5] D. J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, N. Heninger, T. Lange, N. Van Someren, 'Factoring RSA keys from certified smart cards: Coppersmith in the wild', in: Advances in Cryptology – ASIACRYPT 2013. K. Sako, P. Sarkar (editors). Springer, Berlin, Heidelberg, 2013, ISBN 978-3-642-42044-3, pp. 341-360.

[6] A Debian weak key vulnerability. CVE-2008-0166 (2008).

[7] N. Heninger, Z. Durumeric, E. Wustrow, J. A. Halderman, Mining your Ps and Qs: Detection of widespread weak keys in network devices, 21st USENIX Security Symposium 2012, Bellevue, USA, pp. 205-220.

[8] Bitcoin.org, Android security vulnerability Alert Notice, 11 August 2013. Online [Accessed 29 October 2020]. https://goo.gl/zK1Hpm

[9] K. Michaelis, C. Meyer, J. Schwenk, Randomly failed! the state of randomness in current Java implementations, in: Topics in Cryptology – CT-RSA 2013. LNCS, vol. 7779. E. Dawson (editor). Springer, Heidelberg, 2013, 978-3-642-36095-4_9, pp. 129-144.

[10] S. H. Kim, D. Han, D. H. Lee, Predictability of Android OpenSSL's pseudo random number generator, Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, ACM, 2013, Berlin, Germany, pp. 659-668.

[11] J.-M. Gurney, URGENT: RNG broken for the last four months (2015). Online [Accessed 29 October 2020]. https://goo.gl/KtQhD5

[12] P. L'Ecuyer, 'Random number generation', in: Handbook of Computational Statistics. J. Gentle, W. K. Härdle, Y. Mori (editors). Springer Berlin, Heidelberg, 2012, ISBN 978-3-642-21550-3, pp. 35-71.

[13] B. Russell, Mysticism and Logic and other Essays, 2nd edition, George Allen & Unwin LTD, London, 1917.

[14] D. Knuth, Construction of a random sequence, BIT 5 (1965), pp. 246-250.

[15] M. Sipser, Introduction to the Theory of Computation, 3rd international edition, CENGAGE Learning, 2013, ISBN 978-1-133-18781-3.

[16] D. Chicayban Bastos, Uma versão quântica do algoritmo Rô de Pollard (master's thesis), Universidade Federal Fluminense, Niterói, 2019.

[17] G. Marsaglia, Xorshift rngs, Journal of Statistical Software 8(14) (2003), pp. 1-6.

[18] M. E. O'Neill, PCG: A family of simple fast space-efficient statistically good algorithms for random number generation, Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 2014.

[19] S. A. Crosby, D. S. Wallach, Denial of service via algorithmic complexity attacks, USENIX Security Symposium, 2003, Washington, DC, USA, pp. 29-44.

[20] M. Matsumoto, T. Nishimura, Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACMT Transactions on Modeling and Computer Simulation (TOMACS) 8(1) (1998) pp. 3-30.

[21] G. Marsaglia, DIEHARD, a battery of tests for random number generators, CD-ROM, 1996.

[22] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, S. D. Leigh, M. Levenson, M. Vangel, N. A. Heckert, D. L. Banks, A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST, Special Publication 800-22 Rev 1a, 2010.

[23] P. L'Ecuyer, R. Simard, TestU01: a C library for empirical testing of random number generators, ACM Transactions on Mathematical Software (TOMS) 33(4) (2007) p. 22.

[24] C. Doty-Humphrey, PractRand, 2018. Online [Accessed 29 October 2020]. https://goo.gl/HwU9g5

[25] G. Johnson, gjrand, 2014. Online [Accessed 29 October 2020]. https://goo.gl/2AxRWu

[26] D. Chicayban Bastos, L. A. Brasil Kowada, R. C. S. Machado, Measuring randomness in IoT products. II Workshop on Metrology for Industry 4.0 and IoT, 2019, Naples, Italy, pp. 466-470.

[27] D. Chicayban Bastos, L. A. Brasil Kowada, Medindo a qualidade de geradores de números aleatórios, IV Workshop sobre Regulação, Avaliação da Conformidade, Testes e Padrões de Segurança, 2019, Campinas, Brazil.

[28] P. L'Ecuyer, R. Simard, E. Jack Chen, W. D. Kelton, An object-oriented random-number package with many long streams and substreams, Operations Research 50(6) (2002) pp. 1073-1075.

[29] D. J. Bernstein, ChaCha, a variant of Salsa20, Workshop Record of SASC 8 (2008) pp. 3-5.

[30] K. Hirano, T. Yamazaki, S. Morikatsu, H. Okumura, H. Aida, A. Uchida, S. Yoshimori, K. Yoshimura, T. Harayama, P. Davis, Fast random bit generation with bandwidth-enhanced chaos in semiconductor lasers, Opt. Express 18 (2010) pp. 5512-5524.

[31] M. A. Zidan, A. G. Radwan, K. N. Salama, Random number generation based on digital differential chaos, IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS), 2011, Seoul, South Korea, pp. 1-4. DOI: https://doi.org/10.1109/mwscas.2011.6026266

[32] M. Stipcevic, Ç. K. Koç, 'True random number generators', in: Open Problems in Mathematics and Computational Science. Ç. K. Koç (editor). Springer, 2014, ISBN 978-3-319-10682-3, pp. 275-315.

[33] S. Zhu, Y. Ma, J. Lin, J. Zhuang, J. Jing, 'More powerful and reliable second-level statistical randomness tests for NIST SP 800-22', in: Advances in Cryptology, ASIACRYPT 2016. Lecture Notes in Computer Science, vol 10031. J. Cheon, T. Takagi (editors). Springer, Berlin, Heidelberg, 2016, ISBN 978-3-662-53886-9, pp. 307-329.

[34] Song-Ju Kim, K. Umeno, A. Hasegawa, Corrections of the NIST statistical test suite for randomness, arXiv preprint nlin/0401040 (2004).

[35] F. Pareschi, R. Rovatti, G. Setti, On statistical tests for randomness included in the NIST SP800-22 test suite and based on the binomial distribution, IEEE Transactions on Information Forensics and Security 7(2) (2012) pp. 491-505.
[36] K. Hamano, The distribution of the spectrum for the discrete Fourier transform test included in SP800-22, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 88(1) (2005) pp. 67-73.
[37] K. Hamano, T. Kaneko, Correction of overlapping template matching test included in NIST randomness test suite, IEICE transactions on fundamentals of electronics, communications and computer sciences 90(9) (2007) pp. 1788-1792.
[38] M. Matsumoto, Y. Kurita, Twisted GFSR generators, ACM Transactions on Modeling and Computer Simulation (TOMACS) 2(3) (1992) pp. 179-194.
[39] G. Marsaglia L.-H. Tsay, Matrices and the structure of random number sequences, Linear Algebra and its Applications 67 (1985) pp. 147-156.
[40] A. Suciu, K. Marton, I. Nagy, I. Pinca, Byte-oriented efficient implementation of the NIST statistical test suite. International Conference on Automation, Quality and Testing, Robotics, 2010, Cluj-Napoca, Romania, pp. 1-6.
DOI: https://doi.org/10.1109/AQTR.2010.5520837