ACTA IMEKO
ISSN: 2221-870X
January 2014, Volume 03, Number 01, 1 - 8

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*
[*] *Authors in alphabetical order. See https://goo.gl/rzBAq9.*

ABSTRACT
Computer sampling and simulation requires fast random number generators; true random number generators are often too slow for the purpose, so pseudorandom number generators are usually the suitable ones. 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 don't 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 objective. The popular NIST SP 800-22 statistical test suite as implemented in the software package is inadequate for testing generators: we show a reproducible experiment whose conclusion asserts the NIST SP 800-22 statistical test suite, as implemented in the software package, cannot be trusted for the task.

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 randomness and random number generators, Acta IMEKO, vol. A, no. B, article N, month year, identifier: IMEKO-ACTA-A (year)-B-N

Section Editor: name, affiliation

Received month day, year; **In final form** month day, year; **Published** month year

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

Corresponding author: Daniel Chicayban Bastos, Luis Antonio Brasil Kowada, Raphael C. S. Machado.

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. "[Its] 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, section 3.3.4, page 105].

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`[1]. 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, it was 47 bits [2].

In 2003, Taiwan launched a project offering its citizens a smart card with which they could authenticate themselves with the government, file taxes *et cetera*. RSA keys were generated by the cards using built-in hardware random number generators advertised as having passed FIPS 140-2 Level 2 certification [17]. "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 certificate secret keys were found out of more than two million 1024-bit

[1] `PID` means process identifier and `PPID` is the `PID` of the parent process in UNIX systems. In the Linux kernel version 2.5.68 every `PID` is a natural number between 1 and 32767 in a 32-bit system. In 64-bit systems, the value can get up to 2^{22}, approximately 4.2 million [25].

RSA keys downloaded from Taiwan's national key repository [3, page 342].

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" [7].

In 2012, a survey of TLS and SSH servers was performed [4]. The entire IPv4 space was scanned, giving us 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, 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. For TLS, at least 5.23% were using default keys generated by the manufacturer and 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 bitcoin theft was related to the implementation of the pseudorandom number generator used in Android [6][9], later replaced by Google Inc. In 2015, a flaw in FreeBSD's kernel turned SSH keys and keys generated by OpenSSL vulnerable due to a possible predictability of a random number generator [8].

We do not think it is absurd to assume that, in the same way smart phones use the same libraries used in server and desktop systems, embedded systems and others will use the same or slim 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 and those called pseudorandom number generators. The former is usually associated with a physical mechanism which produces randomness by way of a physical process "such as the timing between successive events in atomic decay" [11, section 2.2.1, page 38]. The acronym TRNG stands for true random number generator and is usually used to represent them. PRNG stands for pseudorandom number generator and is the acronym used to refer to them. A pseudorandom number generator is often an arithmetical procedure performed by a machine given by an initial, hopefully random, information called *seed*. If a pseudorandom number generator has enough desirable properties to the point of being advised for cryptographic applications, then the acronym CSPRNG is often used, meaning computationally secure pseudorandom number generator.

3. WHAT IS A RANDOM SEQUENCE?

If we look at probability theory textbooks, we can see they require the concept of randomness, but most expositions carefully dodge the difficulty of precisely defining what is a random sequence, 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, compute, not to give meaning [1, section 3.5, page 142]. 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. This view[2] has been remarkably described [36, page 75] by Bertrand Russell.

> 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.

So, in the context of probability theory, *if* you have a random sequence, it can be used to instruct you on how to draw samples from a population. Given these truly random samples, then "such and such" deductions can be made. "It is essential" not to discuss whether the sequence with which we 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[3]. However, if a probability is measured as a number, it can then be compared. For example, we can assert a probability x is greater than a probability y, which is astoundingly useful.

The definition of a sequence ∞-distributed has been given serious consideration as a candidate for a definition of random sequence. To explain what is ∞-distributivity, 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.

[2] Such view of pure mathematics is often rejected in various informal ways, but we restrict ourselves to the context in which the question belongs. In the context of formal logic, it doesn't seem easy to object to such view of axiomatic systems. We do recognize that formal logic is not able to *capture* the whole of mathematics, as it has been clear since the advent of Gödel's Theorems. A through discussion of the implications of the incompleteness theorems and the relationship between mathematics and logic would take us too far afield. 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.

[3] See "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."

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 [1, exercise 1, chapter 3] to $1/2$. 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 $0, 0, 1, 1, 0, 0, 1, 1, \dots$ 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 to every periodic sequence there is a natural number k associated such that the sequence is not k-distributed. Indeed, every periodic sequence of period p is not p-distributed [37, page 248]. A periodic 3-distributed binary sequence is not easily guessed, but one can check that the sequence

$$0,0,0,1,\ \ 0,0,0,1,\ \ 1,1,0,1,\ \ 1,1,0,1,\ \ 0,0,0,1, \dots$$

is in indeed 3-distributed [1, section 3.5, equation 11, page 148].

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: in 1965 one such algorithm was given [37].

The weakness in taking the notion of ∞-distributivity alone as a definition for a random sequence appears when we consider subsequences of an ∞-distributed sequence. If such sequences were random, we would expect that any subsequence of a random sequence would also be random, but this doesn't always happen with ∞-distributed sequences. Given an ∞-distributed binary sequence X_n, we can construct a new sequence $Y_n = X_n$ except that $Y_{n^2} = 0$ for every index n. Clearly, Y_n isn't random because we know that $Y_0,\ Y_1,\ Y_4,\ Y_9, \dots, Y_{n^2} = 0$, but Y_n is still ∞-distributed [1, section 3.5, page 160] because setting squared elements to zero does not significantly change the probabilities required in the definition of k-distributivity [1, section 3.5, page 160]. That is, ∞-distributivity alone is too weak of a definition. An apparently adequate definition is reached [1, definitions R4, R5, R6, pages 161–163] 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. "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, section 3,5, page 161]. We are not aware of any objection made to this strategy.

4. WHAT IS A PSEUDORANDOM NUMBER GENERATOR?

Since algorithms cannot compute random sequences [38, section 6.4], they're 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[4] $f(x_n) = x_{n-1}^2 + 1 \bmod N$, for some fixed natural number N, and as complex as Donald Knuth's "super-random" number generator [1, section 3.1, page 5], shown as an extreme example of how complicated algorithms don't necessarily provide any more randomness. For illustration purposes, here's 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 [20]. The variable `y` is a global variable to 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 adds the result to `y`, that is, it adds the product to the initial value. The multiplication is done in fast computer arithmetic: that's the effect of shifting `y` by 13 bits to the left because, in base 2 arithmetic, shifting a number to the left means adding zeros to the right of this number, which is the same as multiplying it by a power of 2. For example, if we have the number five, which is 101 in base 2, and we multiply it by 2, we get ten, which is 1010 in base 2. The second step divides `y` by 2^{17} and adds it to `y`. The last step is similar. Assume all computations are reduced modulo 2^{32}, although the C programming language does not define what machines must do when a computation overflows an integer. When we look at the numbers produced by this generator, they appear random to us, but the sequence doesn't 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 more cumbersome to install and run, more costly, slower and cannot 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 [11, section 2.2.1, page 38].

[4] This pseudorandom number generator is typically used in the method of integer factorization known as Pollard's rho. One interesting fact about the method is that while polynomials like f don't have good statistical properties, it would considerably increase, on average, the number of steps taken by the procedure if it would be replaced by, for example, a truly random sequence [39, section 2.3, pages 27–28].

When choosing a pseudorandom number generator, we must know what to look for. Some of the properties one can find in pseudorandom number generators, to name a few, is 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 [10, section 2].

In the context of computer-generated randomness, good statistical properties are effectively what is meant by "random" [10, section 2.1]. 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, at the end of which the generator must repeat itself, so 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 we consume a uniform generator. As we draw near the end of its period, its uniformity effectively allows us to predict more and more its output, since all output must occur the same number of times [10, section 2.1.1].

> 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.

A 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 we are to choose between generators with period sizes of 2^{128} and 2^{256}, we 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." [10, section 2.4.2]

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." [10, section 2.2]. Still, pseudorandom number generators are deterministic and their behavior 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 doesn't 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, we need only save its initial conditions, usually just the seed for that sequence produced. To repeat a sequence from a true random number generator, we would have to save the entire sequence produced.

It's 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 such internal state.

Unpredictability is very important for applications concerned with security because predicting a pseudorandom number generator allows for various types of attacks, including denial of service [22]. 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, when the generator becomes completely predictable, at least from that point in the sequence on, which is a flaw of Mersenne Twister [21].

Predictability can be considered in two directions: forwards and backwards. A generator is said to be invertible if, once we know its internal state, we can discover the random numbers it generated previously. So 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's prudent to pick the safest generator affordable by your project [10, section 2.2].

> [...] [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 considering low resource systems. An application that is too dependent on a random number generator will be as slow as the random number 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` [10][20, section 2.3] has good performance and good statistical properties, but it's 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 equal, a generator that's able to keep its internal state completely in a processor's register should outperform a competitor which needs many more bytes of internal state to be kept in main memory [10, section 2.4].

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 [10, section 2.4.3].

> 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, if we skip a sufficient number of elements, we are back to its starting number, implying that the ability to seek ahead also gives us 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 we get very unlikely samples from the random number generator, then we're getting strong evidence the hypothesis H_0 is not true. In fact, when testing random number generators, if any of the right or left p-value is extremely close to zero, then H_0 should be rejected [11, section 2.6, page 56]. If any of the p-value is equal to 1, then the sequence appears to have perfect statistical randomness [13, section 1.1.5]. If a suspicious p-value is obtained, say near 10^{-2} or 10^{-3}, we 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 [11, section 2.6, page 56].

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 us to accept or 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, we reject H_0, that is, we conclude the sequence produced by the generator is not random. Otherwise, we accept H_0. 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 event occurs, it provides us with evidence that the sequence is not random as assumed in H_0.

Although the probability for such 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 we accept H_0 when the sequence produced by the generator is not random, an error that's 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 type II error is usually denoted by β. The value of α can be arbitrarily chosen, that is, if we would like to specify the probability of type I error to 1%, we 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 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. It is possible, however, to minimize the type II error of a certain test. The probabilities α and β are related to each other and also to the size n of the sample. If two of them are specified, the third value can be computed. Usually, a sample size n is chosen along with a probability α and a critical value is chosen such that β is smallest [13, section 1.1.5].

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 we 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's also not possible to build a random number generator that passes all statistical tests [11, section 2.2.4, page 41]. Nonetheless, the more tests we apply to a random number generator the more confident we get of its quality.

Perhaps the first battery of tests was devised by Donald Knuth in 1969 [1, section 3.3, page 38]. In 1996, George Marsaglia published DIEHARD [12] given the insufficiency of Knuth's. NIST, in the United States, published its own battery [13], in the year 2000, being last revised in 2010, to supersede Marsaglia's. 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 [14]:

> [...] 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 showed a "sobering result" [14, table I, section 7] for many well-known random number generators which were "respectable" [10, section 2.1.2]:

> [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 battery of tests: `SmallCrush`, the small one, `Crush`, the medium-sized one and `BigCrush`. `SmallCrush` is the quickest and it should finish under a minute on most modern desktop computers. Crush can take a few hours and `BigCrush` takes various hours or perhaps a day.

How about alternatives to TestU01? Two other packages compete with TestU01: PractRand 0.94 [15] and gjrand 4.2.1 [16], but neither has been formally published.

8. A NOTE ON USING THE TESTU01 LIBRARY

An inconvenience of TestU01 is that it's restricted to the C programming language. It is a C library, after all: it won't just run without a programmer to write a program that takes advantage of the library. Besides, given that TestU01 is written in C, it wouldn't be straightforward to use it from another programming language: one would have to know how to access a C library from within the chosen programming language.

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

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

Similarly, if we have 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[5], `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 programming language adopted. 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 5 tests in `SmallCrush` in less than a minute.

The default pseudorandom number generator in both Python and PHP is `mt19937`, Mersenne Twister [21]. It passes `SmallCrush`, but actually fails the linear complexity test, not included in TestU01's small battery. The linear complexity test is a rather quick test to run and could have been included in the small battery. The number 19937 in its name is due to is 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 [10, section 2.2].

In C++, besides `mt19937`, the standard library also offers `minstd` and `ranlux`, two well-known generators, but `minstd` fails 9 tests out of 15 of the small battery and `ranlux` is not much better [14, section 7].

Exceptionally, some programming languages offer good alternatives. For example, the default pseudorandom number generator in the Racket programming language, from the Lisp family, is Pierre L'Ecuyer's `mrg32k3a` [19], which did pass `SmallCrush` when we tested it, but also passes `BigCrush` [14, section 7, table I].

If an application requires cryptography, a well-known computationally secure pseudorandom number generator is based on the stream cipher ChaCha20 [18]. 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 [14, table I, section 7][10, section 2.1.2], published in 2007, it's not hard to find publications ignoring it [32][34][35, section 4, page 304] while giving attention to the software package provided by NIST SP 800-22. Enough flaws of the NIST SP 800-22 statistical test suite have been previously reported [29, 33]. We present now one more result regarding the insufficiency of the NIST SP 800-22 statistical test suite implementation.

It's known that the Fibonacci sequence, taken as a random number generator, is not satisfactorily random, but a "much better" variation, though never published, was proposed in 1958 by G. J. Mitchell and D. P. Moore [1, section 3.2.2, page 26]. 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 period of length $2^{55} - 1$, implying the sequence X_n must also have a period of the same length [1, section 3.2.2, page 27].

Despite `mm32` having provably a period of a certain 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, section 3.2.2, page 28]. 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[6], 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, section 3.3.2, page 60] and the weight distribution test [28, section 4.4, page 188], after consuming approximately 6.7 gibibits[7] from the generator in less than 10 seconds on a certain

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

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

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

system. PractRand reports that `mm32` fails the binary rank test [31], 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 testing 8 gibibits from the generator in less than 15 seconds. Nevertheless, the NIST SP 800-22 statistical test suite approves `mm32` after consuming a total of 8 gibibits from the generator, that is, after consuming 32 samples of 256 mebibits[8] each, in over 15 hours[9].

Could NIST SP 800-22 statistical test suite reject `mm32` by considering larger samples? We found that 32 gibibytes of memory are not enough to give NIST SP 800-22 statistical test suite a sample size of 2 gibibits. When we reduced the size 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, 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: we could not make sense of the p-values produced by the overlapping template matchings test on sample lengths such as 256 mebibits, for any generator we 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 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 don't offer enough statistical properties. We have argued that the NIST SP 800-22 statistical test suite, as implemented in the software package, last revised in 2010, is inadequate for testing random number generators. With the tool we have developed, testing a random number generator against the state-of-the-art in statistical tests is a trivial matter.

REFERENCES

[1] Donald Knuth, "The Art of Computer Programming", volume 2, 3rd edition, 1997. Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-89684-8.

[2] Ian Goldberg and David Wagner, "Randomness and the Netscape browser," Dr Dobb's Journal-Software Tools for the Professional Programmer, vol. 21, 1, pp. 66-71, 1996.

[3] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. "Factoring RSA keys from certified smart cards: Coppersmith in the wild." In International Conference on the Theory and Application of Cryptology and Information Security, pp. 341-360. Springer, Berlin, Heidelberg, 2013.

[4] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. "Mining your Ps and Qs: Detection of widespread weak keys in network devices." Presented as part of the 21st USENIX Security Symposium 2012, pp. 205-220.

[5] Bitcoin.org, Android security vulnerability Alert Notice, August 11th, 2013. See https://goo.gl/zK1Hpm.

[6] Michaelis, K., Meyer, C., Schwenk, J.: Randomly failed! the state of randomness in current Java implementations. In: Dawson, E. (ed.) CT-RSA 2013. LNCS, vol. 7779, pp. 129–144. Springer, Heidelberg (2013).

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

[8] John-Mark Gurney: URGENT: RNG broken for the last four months (2015). See https://goo.gl/KtQhD5.

[9] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. "Predictability of Android OpenSSL's pseudo random number generator." In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 659–668. ACM, 2013.

[10] Melissa E. O'Neill. "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation." Technical Report HMC-CS-2014-0905, 2014, Harvey Mudd College, Claremont, CA.

[11] Pierre L'Ecuyer. "Random Number Generation." In Handbook of Computational Statistics, James Gentle, Wolfgang Karl Härdle, and Yuichi Mori (Eds.), 2012. Springer Berlin Heidelberg, 35–71.

[12] George Marsaglia. "DIEHARD, a battery of tests for random number generators." CD-ROM, 1996.

[13] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Stefan D. Leigh, M. Levenson, M. Vangel, Nathanael A. Heckert, and D. L. Banks. "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" NIST. Special Publication 800-22 Rev 1a. 2010.

[14] Pierre L'Ecuyer and Richard Simard. "TestU01: a C library for empirical testing of random number generators." ACM Transactions on Mathematical Software (TOMS), 2007, 33(4):22.

[15] Chris Doty-Humphrey, 2018. See https://goo.gl/HwU9g5.

[16] Gary Johnson, 2014. See https://goo.gl/2AxRWu.

[17] National Institute of Standards and Technology (NIST). "Security requirements for cryptographic modules." Federal Information Processing Standards Publication (FIPS PUB) 140-2 (May 2001). See https://goo.gl/a0Sze.

[18] Daniel J. Bernstein. "ChaCha, a variant of Salsa20." In Workshop Record of SASC, 2008, volume 8, pages 3–5.

[19] Pierre L'ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. "An object-oriented random-number package with many long streams and substreams." Operations research 50, no. 6 (2002): 1073-1075.

[20] George Marsaglia. "Xorshift rngs." Journal of Statistical Software 8, no. 14 (2003): 1-6.

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

[22] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks." In USENIX Security Symposium, 2003, pp. 29-44.

[23] Douglas R. Stinson. "Cryptography, Theory and Practice", 3rd ed., 2006. Chapman and Hall, CRC. ISBN: 978-1-58488-508-5.

[24] Linux Programmer's Manual, 2017. See "`man 4 random`".

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

[26] Michael O. Rabin. "Digitalized signatures and public-key functions as intractable as factorization." Technical report MIT/LCS/TR-212, 1979, Massachusetts Institute of Technology. See https://bit.ly/2WSjSXL.

[27] Daniel Chicayban Bastos, Luis Antonio Brasil Kowada, and Raphael C.S. Machado. "Measuring randomness in IoT products." In 2019 II Workshop on Metrology for Industry 4.0 and IoT, pp. 466-470. IEEE, 2019.

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

[9] A more efficient implementation of NIST SP 800-22 statistical test suite has been reported [30], but we could not locate its implementation.

[28] M. Matsumoto, and Yoshiharu Kurita. "Twisted GFSR generators." ACM Transactions on Modeling and Computer Simulation (TOMACS) 2, no. 3 (1992): 179-194.

[29] Zhu S., Ma Y., Lin J., Zhuang J., Jing J. (2016) 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. Springer, Berlin, Heidelberg.

[30] Alin Suciu, Marton, K., Nagy, I., Pinca, I. (2010). Byte-oriented efficient implementation of the NIST statistical test suite. International Conference on Automation, Quality and Testing, Robotics. 2. 1-6. 10.1109/AQTR.2010.5520837.

[31] George Marsaglia and L.-H. Tsay. Matrices and the structure of random number sequences. Linear Algebra and its Applications, 67:147–156, 1985.

[32] Kunihito Hirano, Taiki Yamazaki, Shinichiro Morikatsu, Haruka Okumura, Hiroki Aida, Atsushi Uchida, Shigeru Yoshimori, Kazuyuki Yoshimura, Takahisa Harayama, and Peter Davis. “Fast random bit generation with bandwidth-enhanced chaos in semiconductor lasers.” Opt. Express 18, 5512-5524 (2010).

[33] Song-Ju Kim, Ken Umeno, and Akio Hasegawa. “Corrections of the NIST statistical test suite for randomness.” arXiv preprint nlin/0401040 (2004).

[34] M. A. Zidan, Radwan, A. G., and Salama, K. N. (2011). “Random number generation based on digital differential chaos.” 2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS). DOI:10.1109/mwscas.2011.6026266.

[35] Mario Stipcevic and Çetin Kaya Koç, chapter “True Random Number Generators” in “Open problems in mathematics and computational science”. Springer, 2014.

[36] Bertrand Russell. “Mysticism and logic and other essays.” George Allen & Unwin LTD, Museum Street, London, 2nd edition, 1917.

[37] Donald Knuth. “Construction of a random sequence.” BIT 5, 246–250, 1965.

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

[39] Daniel Chicayban Bastos. “Uma versão quântica do algoritmo Rô de Pollard”. (Master’s thesis). Universidade Federal Fluminense, Niterói, 2019.