Measuring randomness in IoT products

1st Daniel Chicayban Bastos
Instituto de Computação
Universidade Federal Fluminense
Niterói, Brasil

2nd Luis Antonio Brasil Kowada
Instituto de Computação
Universidade Federal Fluminense
Niterói, Brasil

3rd Raphael C. S. Machado
Instituto de Computação
Universidade Federal Fluminense
Niterói, Brasil

***Abstract*—Choosing and using a random number generator is no simple task. Most random number generators fail statistical tests, revealing patterns in their output. Good generators can lead to disasters if not properly seeded. IoT products may not afford safe random number generators if these products are built on low resource hardware because safety tends to require more resources than not. Testing the chosen random number generator under the conditions of the product is important to know its limitations. We present an easy-to-use tool we have developed to bring the industry's products the state-of-the-art in random number generator testing.**

***Index Terms*—PRNG, RNG, randomness, random number generators, statistical tests, TestU01.**

I. 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`. 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 is 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 [18]. "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 RSA keys downloaded from Taiwan's national key repository [3, page 342].

This work has been partially supported by the SHCDCiber project. Authors listed in alphabetical order: see https://goo.gl/rzBAq9.

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." [8]

In 2012, a large 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 analyzed 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 pseudo-random number generator used in Android [7] [10], 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 [9].

Most incidents mentioned involve computer and smart phone systems, but it is not absurd to assume most IoT devices will use some of the same libraries powering these applications, in the same way smart phones use the same libraries used in server and desktop systems. Worse, IoT devices will probably use slim versions of these software due to their low resource demands, generating more concern as stable implementations of verified software might be changed to fit in with the requirements of an IoT hardware or application.

II. Terminology

There are at least two types of random number generators, those called *true random generators* and those called *pseudo-random 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" [12, section 2.2.1, page 38]. The acronym TRNG stands for true random number generator and is usually used to represent them. (Sometimes RNG refers to TRNG, but we prefer to reserve RNG to mean any random number generator, pseudo or not.) PRNG stands for pseudo-random number generator and is the acronym used to refer to them. A PRNG is often an arithmetical procedure performed by a machine given by an initial, hopefully random, information called the seed. If a PRNG has enough desirable properties to the point of being advised for cryptographic applications, then the term CSPRNG is often used, meaning *cryptographically secure pseudo-random number generator*.

III. Desirable properties

TRNG have several disadvantages compared to a good PRNG. 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 PRNG does need a good seed, which TRNGs can provide [12, section 2.2.1, page 38].

When choosing a PRNG, we must know what to look for. Some of the properties one can find in PRNGs, 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 [11, section 2].

In the context of computer-generated randomness, good statistical properties are effectively what is meant by "random" [11, section 2.1]. Mathematical foundations allow us to be sure a PRNG 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 [11, 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. However, in terms of period size, more is not always better. For example, if we are to choose between generators with period size 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 2^{256} is not 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." [11, 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." [11, section 2.2]. Still, PRNGs 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. Although the deterministic nature of PRNGs 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 PRNG, we need only save its initial conditions, usually just the seed for that sequence produced. To repeat a sequence from a TRNG, 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 PRNG output a number while keeping another one hidden from the user. The hidden information is called the PRNG's internal state. Predicting the PRNG entails knowing such internal state.

Unpredictability is very important for applications concerned with security because predicting a PRNG allows for various types of attacks, including denial of service [24]. If a PRNG 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 that the generator Mersenne Twister [23] suffers — see Section VI.

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 CSPRNGs are not invertible. Although some applications may not be designed with cryptography in mind, it's a good idea anyhow to pick the safest generator for which your project allows [11, 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 IoT applications. An application that is too dependent on an RNG will be as slow as the RNG used. IoT applications will often run on very low resource hardware, so we can anticipate that programmers will 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 [11, 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 IoT 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 [11, section 2.4].

There is also the space constraints of code size. Such space is most likely a constant, but constants do matter for IoT applications. The longer the code, the more likely it will include programming errors. Such errors can be particularly difficult to detect in the context of RNGs [11, section 2.4.3].

> From personal 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. A generator that's seekable is one in which we can skip an arbitrary number of elements of the sequence. Since PRNGs are cyclic, if we skip a sufficient number of elements, we are back to its starting number, implying that the ability to seek ahead can also give us the ability to seek backwards. CSPRNGs are designed not to allow seekability as it is not desirable to let an adversary read the sequence backwards, discovering which numbers were produced in the past.

IV. Statistical tests

Statistical theory allows us to posit a hypothesis H_0 about an RNG 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 an RNG is defined by a random variable X whose distribution under H_0 can be well approximated. When X takes the value x, define $p_R = P[X \geq x \mid H_0]$ and $p_L = P[X \leq x \,|\, H_0]$ as the left and right p-value, respectively. Such p-values measure how likely it is to find a certain sample of the RNG given H_0 is true. If it turns out we get very unlikely samples from the RNG, then we're getting strong evidence the hypothesis H_0 is not true. In fact, when testing RNGs, if any of the right or left p-value is extremely close to zero, then H_0 should be rejected [12, section 2.6, page 56]. If any of the p-value is equal to 1, then the sequence appears to have perfect statistical randomness [14, 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 [12, 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, prompting us to reject 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 [14, section 1.1.5].

V. The state-of-the-art in testing

Under the framework of hypothesis testing, a series of tests can be devised to analyze samples of the RNG. There is no maximum number of tests we can apply to an RNG and there is no maximum number of tests an RNG can pass that will prove it to be truly random. It's also not possible to build an RNG that passes all statistical tests [12, section 2.2.4, page 41]. However, the more tests we apply to an RNG 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 [13] given the insufficiency of

Knuth's. NIST, in the United States, published in the year 2000 its own battery [14] to supersede Marsaglia's, being last revised in 2010. Robert Brown published *DieHarder* in 2004. In 2007, Pierre L'Ecuyer and Robert Simard published TestU01, a C library with which C programmers can implement and test RNGs [15]:

> [...] empirical testing of RNGs 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 RNGs. 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" [15, table I, section 7] for many well-known RNGs which were "respectable" [11, section 2.1.2]:

> [L'Ecuyer and 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* [16] and *gjrand* [17], but neither has been formally published.

VI. The status quo

If one is writing a new application that needs an RNG, one should not just use RNGs offered by the system or programming language adopted. Most programming languages have adopted flawed generators. Java, for example, offers the package `java.Util.Random` which is based on the PRNG `drand48`. It failed 5 tests in `SmallCrush` in less than a minute.

The default PRNG in both Python and PHP is `mt19937`, Mersenne Twister [23]. 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 PRNG, `mt19937` can be totally predicted after collecting of sample of size 624 [11, 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 [15, section 7].

Exceptionally, some programming languages offer good alternatives. For example, in Racket, from the Lisp family, its default PRNG is Pierre L'Ecuyer's `mrg32k3a` [21], which did pass `SmallCrush` when we tested it, but also passes `BigCrush` [15, section 7, table I].

If an application needs cryptographic security, a well-known CSPRNG is based on the stream cipher ChaCha20 [20]. 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 show some evidence that ChaCha20 is currently well regarded.

VII. A new easy-to-use tool

An inconvenience of TestU01 is that it's restricted to the C programming language. It is a C library, after all: we can't just run it. In addition, how would we test an RNG provided by, say, another programming language? The typical way to test an RNG against TestU01 batteries is to implement it in C by using TestU01's C interfaces.

With the aim of bringing TestU01 to every developer, we wrote a program that tests any RNG at all that can run in the system as a standalone program.

Our strategy is to implement an RNG in C whose source of randomness is not an arithmetical procedure, but the *standard input* of the process. In other words, when our program starts, it reads its *standard input* and uses the data it reads as the source of randomness for the generator. This way it can take random data from any source at all, so long as the source writes its random data in *little-endian* binary form to the *standard output*.

This way we effectively separate the RNG from TestU01. Testing any RNG is a matter of telling the system's shell to run the RNG piping its output to the *standard input* of our program. For example, to test *Xorshift* [22], represented below by the program *xs32*, against the *SmallCrush* battery of tests from the TestU01 library, we just need to say to the shell:

```
%./xs32 | crush --battery small \
                --name xs32
...
%
```

TRNGs should be tested too. If a TRNG produces a file containing the data, testing that data is analogous to the last command. For example,

```
%cat trng.bin | crush --battery big \
                      --name trng.bin
...
%
```

runs the `BigCrush` battery of tests against the data saved in file `trng.bin`.

VIII. Future work

TestU01 is intrinsically 32-bit based, making it inconvenient to test generators whose output size is not 32 bits. For example, an RNG with an output size of 31 bits will have a zero in its most significant bit, stored in an integer of 32 bits, which will definitely be noticed by TestU01. To understand this difficulty, let's consider an example. Suppose an RNG has output size of 29 bits. Each number produced by the RNG misses 3 bits to fill an integer data type of 32 bits. How can we get 3 new random bits? One trivial solution is to consume the next number from the generator and reduce it modulo 2^3, creating the 3 new bits we need. However, many statistical tests in TestU01 are more sensitive to the most significant bits (also called high bits) than to the least significant bits (also called low bits), so we are currently considering whether this is an interesting solution.

Also, if the RNG has more than 32 bits, similar inconveniences arise. For example, to test an RNG whose output size is of 64 bits, we could write a generator that makes two random numbers from each number produced by the 64-bit generator. However, because many tests tend to care more for high bits than low bits, a 64-bit generator should be tested at least twice, once with the numbers without any modification and a second time with the numbers' bits reversed [19, section 3]. Would that be enough? We could, after all, shuffle the bits in other ways too. Writing programs to make all these adjustments is a major inconvenience. We are also currently considering these difficulties.

References

[1] Donald E. 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] Bernstein, Daniel J., 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] Heninger, Nadia, 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 (USENIX Security 12), pp. 205-220. 2012.

[5] Dörre, Felix, and Vladimir Klebanov. "Pseudo-random number generator verification: A case study." In Working Conference on Verified Software: Theories, Tools, and Experiments, pp. 61-72. Springer, Cham, 2015.

[13] Marsaglia, G. (1996). "DIEHARD, a battery of tests for random number generators." CD-ROM.

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

[7] 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)

[8] A Debian weak key vulnerability. CVE-2008-0166 (2008). URL: https://goo.gl/NzDrii

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

[10] Kim, Soo Hyeon, 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.

[11] O'Neill, Melissa. E. (2014). "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. URL: https://goo.gl/dw1uqW

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

[14] Bassham, Lawrence E., 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.

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

[16] Doty-Humphrey, C. (2014). PractRand. URL: https://goo.gl/HwU9g5.

[17] Johnson, G. (2014). gjrand. URL: https://goo.gl/2AxRWu.

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

[19] Vigna, Sebastiano. "An experimental exploration of Marsaglia's xorshift generators, scrambled." ACM Transactions on Mathematical Software (TOMS) 42, no. 4 (2016): 30.

[20] Bernstein, D. J. (2008). "ChaCha, a variant of Salsa20." In Workshop Record of SASC, volume 8, pages 3–5.

[21] L'ecuyer, Pierre, 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.

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

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

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