Noname manuscript No.
(will be inserted by the editor)

TestU01 and Practrand: Tools for a Randomness Evaluation for Famous Multimedia Ciphers

Lama Sleem* · Raphaël Couturier

the date of receipt and acceptance should be inserted later

Abstract New dangerous attacks have arisen as we witness the current evolution of digital data. Connecting devices, vehicles, even our own bodies to the Internet have generated enormous amounts of data that need to be secured. New security solutions and ciphers are being proposed taking into consideration all the limitations in the devices used in today's technologies. However, different factors have to be taken into consideration to prove the reliability of any cipher. One of these criteria is the randomness of the ciphered output. Usually, randomness tests are used to prove the efficiency of Pseudo Random Number Generators- PRNGs, and they are not considered in the test suite for cryptographic algorithms. This paper proposes using the well known tools Practrand [8] and TestU01 [18] to test the randomness criteria for any new/old symmetric cipher. To show our cryptographic point of view, several well known ciphers were tested by these tools. Some of them failed these tests and did not meet the desired security requirements and the sufficient statistical immunity. In fact, this paper shows that these ciphers do not generate enough randomness making them vulnerable to different kinds of attacks which reinforces our proposal.

Keywords: Internet of Things; Cryptography; Randomness tests; Unpredictability.

1 Introduction

In today's technological revolution, a security guarantee has become a major issue and a basic need for users, companies, applications, and researches alike. Many new terms and technologies have invaded the industry and the research fields. Internet of Things(IoT) is one of the most promising research topics in both engineering field and business. Connected devices are increasing day after day. In fact, Cisco's Internet of Things Group (IoTG) estimated the number of connected devices to reach 50 billion by year 2020 [20]. These connected devices exchange different kind of information as streaming of stored multimedia content(audio, video), live streaming (video conferencing, online gaming), and real-time interactive multimedia communication such as the case of surveillance. These exchanged packets should be transmitted and received continuously. However, these packets of data need to be secured against all the powerful new attacks. In order to go along with this tremendous generation of data, cryptographers are excelling at finding new solutions that can respect the new requirements. As such, using Lightweight Cryptography (LWC) has become one of the foremost desired solutions in security especially for limited sensors. It investigates the implementation of cryptographic algorithms for resource constrained devices

Sleem L.
Univ. Bourgogne Franche-Comté (UBFC), FEMTO-ST Institute, France
Tel.: +33-07-69475406
E-mail: lama.sleem@univ-fcomte.fr
*Corresponding author

Couturier R.
Univ. Bourgogne Franche-Comté (UBFC), FEMTO-ST Institute, France

[4,10] that are excessively used in today's networks. The use of limited resources, batteries, sensors, and Wireless Sensor Networks justifies the need for efficient lightweight cryptography. Smaller block sizes, modest key size, little code measure, fewer clock cycles and lower number of rounds are all factors that can cause any cipher to become lightweight. However, there must be a trade-off between the security of the cipher and the limitations of the constrained devices. In order to have an efficient and reliable cipher, it must undergo a certain number of tests. One of these tests is the **randomness test**. The security provided by these cryptographic algorithms is directly related to randomness, since compromising the randomness of the ciphered output will jeopardize the whole system. Ignoring this criterion would be extremely dangerous, since the recovery from a security breach can be extremely expensive. Randomness can be defined as the outcome of a probabilistic process that produces independent, uniformly distributed and unpredictable values that cannot be reliably reproduced [25]. The main concern in randomness is being able to produce an unpredictable output, which is an uncorrelated output having a uniform distribution with the lack of bias. Random outputs must be unpredictable, irreproducible and should prevent any attacker to learn/predict former or subsequent values. Yet, in the absence of qualified randomness tests, the quality of the cipher will not be verified. However, random sequences can generally have specific statistical properties that can be measured using different statistical tools. Some of these tools are TestU01 [18], Practrand [8], DieHard [21] and ENT [29] etc... These tools try to avoid sequences which do not verify certain statistical properties, but cannot guarantee perfect randomness. However, passing these tools will grant the cipher the minimum required randomness validity and will highly assure the statistical properties it must possess. In this work, several crytographic symmetric algorithms were implemented using Practrand and TestU01. These tools are simple to be used and they can validate the randomness of the cipher text. Some of the algorithms implemented failed these statistical tests which shows the importance of implementing any new proposed cryptographic algorithm into these tools. This work presents a practical proposal for cryptographers to validate the randomness produced by their symmetric algorithms. It can guarantee the randomness level desired in newly proposed ciphers that are responsible for protecting the different kinds of data exchanged.

1.1 Motivation

Having this amount of exchanged contents, producing an output that can be random enough to prevent any data recovery is the main concern of any cryptographer. In order to increase the randomness produced by any cipher, one of the most important elements of the cipher is the **Cryptographic Key**. It holds the security of the whole system, so, the generation, agreement, storage and destruction of the key must be well managed. Any information about the key will lead to the knowledge of the secret message and thus a security breach will occur. Cryptographic keys must be long enough, must have a large key space and must be generated by a complex and efficient method. The keys must be unpredictable, thus must have high uncertainty (high entropy), highly independent bits, a uniform distribution, and cannot be reproduced, thus, in other words, the keys must be random. Using **Initialization Vectors** also adds randomness to the system where the same key can generate different unique outputs by adding an initial vector into the process. **Cryptographic Salts** can also add randomness especially when used for passwords to avoid easy carried out dictionary attacks [32]. **Padding strings** are also implemented extensively in key block ciphers to avoid compromising short messages and to disguise the original length of the message by adding a random padding to the plain-text block. **Nonces**, numbers used only once, are also of great importance in cryptography to avoid reusing any value [23].
As can be seen, many efforts have been exerted to add randomness to the cipher to prevent different possible attacks. An insufficient degree of randomness will expose the system to differential attacks [7], chosen-plain text attacks, known-plain text attacks etc... Therefore, randomness tests are the least cryptographers can do. **The main contribution of this paper is to accentuate the importance of using Practrand and TestU01 that can eliminate to a great extent any doubts in the randomness of the symmetric ciphered output. The aim is to show that any new proposed cipher should at least pass the tests simulated by these available, simple and free tools.**

The remainder of this article is structured as follows. In Section 2, the implemented cryptographic algorithms are briefly discussed with an explanation of TestU01 and Practrand. Then, in Section 3, the setup and the scenarios proposed for the implementation is described and a performance evaluation for all implemented algorithms is presented accompanied by a discussion for ciphers that failed the tests. Finally, Section 4 concludes this work.

2 Backgrounds and Overview

In this section, all the algorithms that were selected and implemented in this work are briefly described. These algorithms are either proposed or used for different multimedia content. Then, the tests approving/disapproving the randomness of the ciphered output, assessed through TestU01and Practrand, are given.

2.1 Overview of the tested ciphers

In order to validate the given proposal, a set of cryptographic algorithms were tested. These algorithms vary in their key space, number of rounds and mode of operation. They were selected in this work because they are well-known for their good security measures or are new proposals that need to be securely validated. They are listed below and are represented in Table 1:

– PRESENT [6]: Present is considered as one of the lightweight and ultra-lightweight ciphers. In fact, it represents a milestone in the field of lightweight cryptography. It uses 80-128 bit key with 64 bit blocks through 31 rounds. PRESENT is one of the first ciphers implemented on ultra-constrained devices.
– IDEA (International Data Encryption Algorithm) [17]: In order to reduce the memory overhead, IDEA uses only XOR, addition and modular multiplication operations. It uses 128-bit key with 64-bit blocks through 8.5 rounds where all data operations are performed in 16-bit unsigned integers.
– LBlock [34]: LBlock is a lightweight algorithm. It uses 80-bit keys and 64-bit blocks through 32 rounds. The authors chose to apply diffusion on half of the data in each round and a simple rotation on the other half, therefore, it produces ultra-lightweight implementations in both hardware and software.
– HIGHT [11]: It is another lightweight cipher based on simple computations and operations. It uses a robust round function avoiding the use of S-boxes where the key is 128-bits and 64-bit blocks are processed through 32 rounds.
– TEA (Tiny Encryption Algorithm) [30,35]: TEA is a block cipher known for its simplicity in implementation on both hardware and software. It operates on two 32-bit unsigned integers that can be derived from a 64-bit data block, and uses a 128-bit key. It has a Feistel structure with a suggested 64 rounds.
– XXTEA (Corrected Block TEA) [31]: XXTEA was proposed as a correction for TEA algorithm that suffered from several weaknesses [1]. The block has an arbitrary size, at least two words (64 bits), and the key size is 128 bits where the number of round is dependent on the block size. The number of full cycles to perform over the block is given as $6 + 52/n$ (6-32 full cycles) where n is the number of words in the block.
– RC4 [13]: The famous RC4 (Ron's Code) generates a pseudo random stream of bits (a keystream). These streams can be used for encryption by combining the generated stream with the plain-text using bit-wise exclusive-or. The most often used key is 16 bytes (128 bits), but other keys used can be between 40-bits and 256-bits.
– RC4D [16]: RC4D is an enhancement for RC4 proposed by Michael Kwasnicki. The author added a level of diffusion into the original RC4 and proposed four different versions of his code which are: **k,i,p,e**.
  Implementations with a **k** have a fixed key length of 16 bytes (128 bits) which leads to a more than 2× increase in speed and a decent size reduction. Implementations with an **i** perform the encryption in-place. It will reduce the need for intermediate buffers and will also reduce the pressure on the scarce RAM during en/decryption while providing a small increase in speed and a small reduction in size. Implementations with a **p** perform a pre-computation of the S-Box using the key. Due to the aimed small plain-text size, the S-Box is very large compared to it and also the computation thereof. A pre-computed S-Box reduces this overhead for every en/decryption to just one call of memcpy. This provides a 3× speedup but comes at the expense of RAM. Moving this to EEPROM proves to be ineffective as can be seen at the implementation with an **e**.

Table 1: Cryptographic algorithms tested by TestU01 and Practrand.

Block Ciphers			
Algorithms	Key Length (bits)	Block size (bits)	Round number
Hight	128	64	32
Camellia	128 192,256	128	18 24
Lblock	80	64	32
Present	80 128	64	31
TEA	128	64	64
XXTEA	128	>64	>6 <32
BlowFish	>32 <448	64	16
TwoFish	128, 192, 256 bits	128	16
IDEA	128	64	8.5
3DES	168, 112, 56	64	48
Stream Ciphers			
Rabbit	128	1	1
RC4	40-256	1	1
RC4Dkip	40-256	1	1
ChaCha	128 or 256	1	8-12-20
HC-128	128	1	1

– ChaCha [3]: It is a modification of Salsa20. ChaCha is a stream cipher which uses a 256-bit key and 64-bit Nonce and is based on the 8-round cipher Salsa20/8. The changes made are designed to improve diffusion per round, thus increasing the resistance to cryptanalysis, while preserving and improving time per round. Round number in ChaCha can be 8, 12 and 20 as well as 128 and 256 bit keys. In this work, ChaCha20 was implemented with a 128 bit key.
– Blowfish [24]: It is a symmetric-key block cipher with a 64-bit block size and a variable key length from 32 bits up to 448 bits. Blowfish is a 16-round Feistel cipher and uses large key-dependent S-boxes and a highly complex key schedule.
– Twofish [26]: It is a symmetric-key block cipher derived from Blowfish cipher, with a block size of 128 bits and key sizes 128, 192, 256 bits. The number of rounds is 16 and it has a Feistel network structure.
– 3DES [12]: It is a symmetric-key block cipher, which applies the DES cipher algorithm three times to each data block. It was invented since the original DES turned out to be weak and easy to break. 3DES uses 48 rounds in its computation (transpositions and substitutions), and has a key length of 168, 112 or 56 bits.
– HC-128 [33]: HC-128 is known to be a simple and secure stream cipher. From a 128-bit key and a 128-bit initialization vector, HC-128 generates a keystream with length up to 2^{64} bits. HC-128 was designed to prove that a strong stream cipher can be built from nonlinear feedback function and nonlinear output function.
– Camellia [2]: Camellia is a block cipher with symmetric key operating in either 18 rounds for a 128-bit key or 24 rounds for 192- or 256-bit keys, and has a block size of 128 bits. Camellia was designed to be efficient for both software and hardware implementations and it is used in various devices from low-cost smart cards to high-speed network protocols.
– Rabbit [5]: Rabbit is a lightweight stream cipher, famous for its high-speed. It creates a key stream from a 128-bit key, a 64-bit initialization vector, and 513 bits of internal data. The cipher was designed with high performance in software in mind.

2.2 An overview of TestU01 & Practrand

As explained earlier, randomness is important in cryptography to ensure: (1) randomness of the cryptographic keys, (2) security against attacks, (3) privacy and anonymity, (4) and to ensure unpredictability. This can only be achieved by using high quality randomness validation tools. The two tools that are used in this work are TestU01 and Practrand that operate distinctly. TestU01 is a comprehensive C library that contains examples of PRNGs, utilities and a collection of statistical tests drawn from the academic literature of RNGs, whereas Practrand is a C++ library of pseudo-random number generators (PRNGs, or just RNGs) and statistical tests for RNGs. Previously, PractRand (standard, 1 terabyte) found bias in 78 PRNGs while TestU01 (the BigCrush) found bias in 50 PRNGs. Each tool has its own means to define the level of randomness. For example, Practrand is the only test suite to allow functionally unlimited test lengths. It requires more bits to find bias than any other test suite, and multi-threading is supported, but maximum speedup tends to be limited to about 3x. While for TestU01, there exist three tests: SmallCrush, Crush, and BigCrush. TestU01 is the only test suite with a big academic name behind it, and the only test to guarantee (on default settings anyway) that all subtest results are 100% independent. However, it does not support multi-threading.

P-value:

The metric that is used in this kind of tests is the "p-value". In statistical hypothesis testing, the p-value or probability value is the **probability of obtaining test results at least as extreme as the results actually observed during the test, assuming that the null hypothesis is correct.** More simply, as explained by Stuart Buck, vice president of research integrity at the Laura and John Arnold Foundation, you have a coin that you suspect is weighted toward heads (your null hypothesis is then that the coin is fair.) You flip it 100 times and get more heads than tails. The p-value will not tell you whether the coin is fair, but it will allow you to know the probability that you would get at least as many heads as you did if the coin was fair. Although the p-value was introduced by Karl Pearson in 1900 with his chi square test [22], it was the Englishman Sir Ronald A. Fisher, considered by many as the father of modern statistics, who in 1925 first gave the means to calculate the p value in a wide variety of situations [9].

According to the Practrand reference, most of the tests produce results of a form of this "p-value". This value ranges between 0 and 1 and quantifies whether the results fall in the range of the expected truly random data. P-value for data from good ciphers or pseudo-random generators are supposed to be uniformly distributed. The results can never truly lie outside the range expected from truly random data, but they can lie absurdly close to an edge of the expected range. In fact, the resulted cipher text produced should have no correlation on different seedings of a good chosen RNG (in our case we used Splitmix64).

The evaluations treat all PractRand p-values as having two symmetric error regions, both zero and one are considered equally bad, despite the fact that PractRand tests are designed to produce zero for most expected failure circumstances. A p-value can be converted to a pass or fail result by simply comparing it to an arbitrary threshold. For example, is the p-value inferior to 0.001? Then it failed. The exact value of the arbitrary threshold used varies widely depending upon who picks it. The source for this empirical statistical test is the "The Art of Computer Programming" by Donald Knuth (Volume 2). Our aim in this work is to distinguish the cipher texts that are truly random from cipher texts that only look to be random. In practice, looking for patterns that tend to arise from non-random processes of limited complexity and very finite state size built out of standard mathematical operations (particularly those that are most efficient in binary logic) is efficient, which is the case of ciphers.

Below is an overview on how Practrand or TestU01 operate :
Tests in PractRand:

- BCFN: This test checks for long range linear correlations (bit counting); in practice this usually detects Fibonacci style RNGs that rely upon large lags to defeat other statistical tests. Two integer parameters are used:(1) the minimum "level" it checks for bias at (it checks all higher levels that it has enough data for), higher values are faster but can miss shorter range correlations. The recommended minimum level is 2, since that helps it skip the slowest parts and avoids redundancy with DC6 checking for the shortest range linear correlations, while still doing a reasonable amount of work considering how much memory it has to scan. (2) The second integer parameter helps to determine the amount of memory and cache it will use in testing. It is the log-base-2 of the size of the tables it uses internally. Recommended values are 10 to 15, larger values should be used if cache is large, lower values should be used if cache is small. Each individual "level" of this is a frequency test on overlapping sets of hamming weights. BCFN failures having the first parameter listed as high, (for example 5 or more) reflect a cyclic buffer being traversed where some patterns in the buffer content shows up after a specific spacing. On the other hand, if the first parameter listed is low, this is similar to a failure on a DC6 parametrization. Note that the first BCFN parameter is the value listed in the output, for example, a failure with BCFN(2+3,13-1), the first parameter would be 5 (2+3).

– DC6: checks for short range linear correlations (bit counting); takes several parameters that determine the size of the integers it operates on internally, the number of adjacent such integers it looks for correlations between, and which information it uses for each such integer; it is a frequency test on overlapping sets of hamming weights. The failures on short-ranged tests, especially DC6 parameterizations, are typical of small chaotic PRNGs with insufficient mixing/avalanche. This also means that the avalanche criterion proposed by Shanon is not guaranteed in such cases. Failures on FPF parameterizations can have a similar meaning.
– Gap16: A variation on the classic "Gap" test. Usually, the gap test is used to determine the significance of the interval between recurrence of the same digit. The strength of correlation is very strong for shorter sequences and slowly decreases as the sequence length increases. The Gap-16 results should have no significant correlation with the other standard tests. Gap16 failures with p-values near 1 generally mean that the PRNG output was too regular, typically reflecting a single-cycle PRNG nearing the end of its cycle length. Those with p-value near zero on the other hand indicate that some values showed up at more extreme spacings than expected, seen most often on single-cycle PRNGs and sometimes on rc4-like PRNGs.
– BRank: A standard binary matrix rank test. The most original part of it is the control logic that decides when data should be taken from the RNG output stream to make a matrix and what size matrix it should be. The parameter is a log-scale amount of time per gigabyte it spends calculating matrix ranks. Due to the coarse-grained nature of the results it produces, precise p-values are impossible for many of its subtests. Failures on BRank suggest that in some way the output, or at least a part of it, was extremely linear, producible strictly by xoring bits of previous output.
– FPF: "floating point frequency" test; it is purely an integer math test. This checks for very short range correlations, even shorter than DC6, especially those correlations involving lots of 0 bits. Technically speaking, this test does a frequency test applied to the binary format of floating point numbers storing the integer values of overlapping windows of the original data stream.

Tests in TestU01:

TestU01 is implemented in the ANSI C language, and offers a collection of utilities for the statistical testing of uniform random number generators (RNG). TestU01 gives the user four groups of modules to analyze the desired RNGs: (1) Implementing pre-programmed RNGs, (2) implementing specific statistical tests, (3) implementing batteries of statistical tests, and (4) running tests to all RNGs families. The tests are applied to a sample of size n produced by the RNG. The p-value will range between 0 and 1. Tests executed by TestU01 will enable one to know the optimum sample size that should be used before the generator starts failing. There are three different battery tests in this library: the Small Crush (10 tests, around 8 seconds), Crush (96 tests, around 30 minutes) and The Big Crush (160 tests, minimum 4 hours). The main aim for any generator is to pass the Big Crush test whose execution can take up to 24 hours to be completed and uses 238 random values. It lists the p-values and shows those who come outside the [0.01,...,0.99] interval. However, the drawback of TestU01 is that it works with a fixed amount of data and discards the least significant bit (for some tests even two bits) of the 32-bit numbers being tested. It is important to state that for academical reasons, TestU01 is designed to test 32-bit numbers, however, the random number generators used today produces 64-bits. Indeed, years ago, most random number generators would produce, at best 31-bit random values. In this work, we cast the output of 64 bit test to a 32-bit test.

In fact, the main concern is the P-value which is explained in details in TestU01 reference. Also, more attacks can be found in "The Art of Computer Programming", where details are given concerning the randomness tests and the attacks that can be launched if a statistical test is not passed. Based on the fact that statistical tests arose to "prove" or "disapprove" hypotheses about observed data, the aim is to show that these randomness tests can show whether there is any vulnerability in any of the proposed ciphers.

3 Proposed Scenario and Randomness Evaluation

In this section, all the aforementioned cryptographic algorithms are tested. All the codes are implemented using the C language. Two libraries are used to test these algorithms in addition to some codes implemented manually from trusted repositories. The two libraries chosen are Libgcrypt [15] and Wolfcrypt [28]. The wolfSSL library is known to be a lightweight, portable, C-language-based SSL/TLS library that is targeted at IoT because of its size, speed, and feature set. It works seamlessly in desktop, enterprise, and cloud environments as well. This library has been selected since there are two versions of the WolfCrypt cryptography library that have been FIPS 140-2 validated. Hence, it can be advocated that it is a well known and reputed cryptographic library. The other library used, is the Libgcrypt that is developed as a separate module of GnuPG in C language. It provides functions for many fundamental cryptographic building blocks. Libgcrypt was also FIPS 140 validated which makes it a reliable source for running the desired tests. Algorithms that were tested using Libgcrypt library are: Arcfour (RC4), Chacha, Camellia, Blowfish and Twofish while the algorithms tested using the Wolfcrypt library are: RC4, Chacha, HC128, Camellia, IDEA, Rabbit, and 3DES. Finally, the rest of the ciphers were taken from well known programmers and trusted repositories and they are: Hight, Lblock, Present, XXTEA, TEA, and RC4D with all its versions.

3.1 Proposed Scenarios:

In order to facilitate the implementations on both tools, the "testingRNG" [19] project released by Daniel Lemire is used which aims at making it easier to run such tests on either MacOS or Linux with a recent C compiler. The seed used in the tests is generated by using the **"splitmix"** (a fast splittable PRNG) [27] which is widely used and is a part of the standard Java API. This generator produces 64 bit numbers. Then, we first considered the **worst case scenario** where the whole plain text is just zeros and in each execution we tend to change only the key or the IV (depending on whether the algorithm has an IV or not) without changing anything else in the algorithm. The key has indeed a major effect on the randomness produced by the algorithm. Then, all the tested ciphers passed first the SmallCrush test by TestU01 which is a very quick test, most commonly used to gauge if it is even worth running the heavier tests. After that, the Bigcrush in TestU01 has been launched as well as the Practrand tests.
Then, the tests will start running for a quite long time before the result can be seen in a log file in Practrand or by checking any failure in TestU01.

According to the tests executed, Practrand was the most efficient in revealing some weaknesses in the tested algorithms. The tests have been executed for at least 4 terabytes data on Practrand and at most for 32 terabytes of data. As shown in Table 2, the algorithms that failed using Practrand are: **RC4 using both cryptographic libraries and ChaCha using Wolfcrypt library only.** Since RC4D proposed different versions of coding as mentioned before, more than one version was tested on both tools. The successful one was RC4D with both optimized and plain versions. While for **RC4Dkip, both optimized and plain versions failed the tests on Practrand and TestU01**.

Concerning the execution times, Table 3 shows the maximum and minimum times that were recorded to detect any failure which are relatively high for both TestU01 and Practrand. However, it is important to note that as the time decreases to detect any failure, the more vulnerable is the cipher and the error is too obvious to be detected by Practrand (like Chacha). Although, these tools take a quite long time to detect any failure, they are simple to be implemented.

After testing the first scenario, three different scenarios that consider different inputs were proposed to show that this scenario (the worst scenario) is the most efficient scenario for Practrand, and allows the detection of any failure more rapidly. The four different proposed scenarios can be summarized as the following:

	Algorithm	TestU01	Practrand	Error Type in Practrand
Libgcrypt	RC4	✓	✗	FPF
	ChaCha	✓	✓	-
	Camellia	✓	✓	-
	BlowFish	✓	✓	-
	TwoFish	✓	✓	-
WolfCrypt	RC4	✓	✗	FPF
	ChaCha	✓	✗	FPF GAP BCFN DC6
	HC-128	✓	✓	-
	Camellia	✓	✓	-
	IDEA	✓	✓	-
	Rabbit	✓	✓	-
	3DES	✓	✓	-
Git Repositories	Hight	✓	✓	-
	LBlock	✓	✓	-
	Present	✓	✓	-
	XXTEA	✓	✓	-
	TEA	✓	✓	-
	RC4D_plain	✓	✓	-
	RC4D_optimized	✓	✓	-
	RC4Dkip_plain	✗	✗	BCFN
	RC4Dkip_optimized	✗	✗	BCFN

Table 2: Successes and Failures obtained by Practrand and TestU01 with the worst case scenario.

	TesU01		Practrand	
	Minimum	Maximum	Minimum	Maximum
Algorithm	Rabbit	3DES	Chacha (WolfCrypt)	PRESENT
Time (hr)	≈ 7	≈ 43	≈ 2	≈ 164

Table 3: Minimum and Maximum CPU Time taken to finish the BigCrush in TestU01 and the tests in Practrand for the tested ciphers.

1. Scenario one: The first scenario used where the plain text is set to zeros. This scenario is considered the easiest scenario for Practrand to detect any failure with. It is the reason for why we considered this in the first place, since when a cipher passes this test, it means that it passed the highest challenge for any cipher. It is the easiest scenario for Practrand, yet the hardest one for any cipher to pass. The aim here is to set the hardest challenge for the cipher and validate its randomness.
2. Scenario two: In this scenario we proposed a **fixed** Latin text as a plain text. As an example, consider the "Lorem ipsum" plain text which is a placeholder text commonly used to demonstrate the visual form of a document or a typeface without relying on meaningful content. This can be considered as the case of testing a fixed image as well. This case is harder to detect by Practrand than scenario one.
3. Scenario three: In this scenario we proposed to test a semi-**random** input. The most 1000 used words in English literature were used as an input and every time the plain text is chosen randomly from these words. This is still harder to detect for Practrand compared to the first two scenarios.
4. Scenario four: Finally, this scenario proposed testing a **random** input. In this scenario we chose a random choice of the plain text from a set of letters (from a to z) with low case characters only. It can be extended to consider other upcase letters and include characters but the logic is still the same. This is the hardest scenario for Practrand to detect any failures with.

To show that the first scenario is the hardest for the cipher to pass and the easiest one to detect by Practrand, we have tested Chacha algorithm that previously failed in the WolfCrypt library and succeeded in the Libgcrypt library. The results of testing Chacha using the three other scenarios can be summarized in Table 4. The results are reasonable and as said, it takes 256 Gigabytes fore Practrand to detect the failure in Chacha in scenario 1 while 16000 Gigabytes (16 Terabytes) to detect the failure by scenario 4.

	Scenario	Time	Data processed (Gigabytes)	Success	Fail
Chacha	1	2905 sec (≈ 1 hr)	256		✗
	2	7708 sec (≈ 2 hr)	512		✗
	3	51435 sec (≈ 14 hr)	4000		✗
	4	123691 sec (≈ 34 hr)	16000		✗

Table 4: Results of Chacha by WolfCrypt when tested with scenario 1,2,3,4 by Practrand.

3.2 Results interpretation and discussion

The simplest way to interpret test results in Practrand is to look for the word "FAIL" in the output. It will appear on the right-hand side and easy to be noticed.

The ciphers used produces a series of temporary result summaries as it goes along. Each result summary has a header showing the cipher tested, the number of bytes tested, the time taken, and the RNG seed used (so you can reproduce the results later if desired). The body of the result summary is a table showing all the irregular results followed by a statement of how many results were omitted from the table because they were regular. If the table would have zero entries then the table is skipped. If no results were omitted then the number of omitted results is skipped. The table of the result in Practrand has four columns: (1) "Test Name", a name for the sub-test the corresponding, (2) "Raw" not of much use to end users, (3) "Processed", either a p-value or "pass" or "fail". (4) "Evaluation", describing the result. "FAIL" means that the tested cipher un-vaguely failed that sub-test, while "suspicious" means that that result should not happen often on a good RNG/cipher but should happen occasionally.

The failing results can be summarized as follows. RC4 in Libgcrypt failed Practrand after 1 terabytes of data indicating the error: FPF-14+6/16:cross. RC4 in Wolfcrypt also failed after 1 terabytes of data signaling the same error: FPF-14+6/16:cross. This clearly shows that the same issue in RC4 is found in both libraries. As it is known, RC4 does not have the required diffusion layer to increase the randomness of the output, and therefore, it has been breached. In fact, as mentioned before, FPF shows that there is a correlation in the ciphered output and that de-correlation fails.

For ChaCha the results in Wolfcrypt were disastrous in terms of randomness. All the tests that are available in Practrand failed after 256 Gigabytes of data. GAP, FPF, DC6 and BCFN were all violated. In particular, a major failure that occurred in one of the oldest and most important tests was the GAP test [14]. This means that some patterns are being detected which might risk the leakage of any useful data.

For the new proposed diffused RC4, which is considered by the author as a chance of reviving the original breached RC4, the author succeeded in adding the new diffusion layer. The original versions of the code, the plain and the optimized ones, did well in Practrand and TestU01. However, the other versions that are based on adding some functionalities to the code, RC4Dkip, did not succeed neither TestU01 nor Practrand using both of their versions, the optimized and the plain codes. It failed after 4 terabytes of data showing the exact error in the plain and optimized versions which is the BCFN(2+0,13-0,T) error. Basically, BCFN tests for long range patterns in the distribution of 0s and 1s. So failing BCFN generally means that given whether 0s or 1s were more common in the previous 32 or 64 byte (n-1) blocks, you have enough information to guess which is more common in the nth with an accuracy that is above 50%, in this error n is 13.

For TestU01, the value that enables us to decide whether this cipher succeeded the test or not is the list of p-value. If the values come outside the [0.01,...,0.99] interval, then this cipher fails TestU01. When a p-value is extremely close to 0 or to 1 (for example, if it is less than 10^{-10}), one can obviously conclude that the generator fails the test. In this case a failure is is defined as a p-value $\leq$ (1.0e-10) or $\geq$ (1-1.0e-10). To see the summarized result of the failure in TestU01, there exists a file ./summarize.pl *.log that contains all the summarized crushed encountered during the test. There were 10 crushes, in the "msb" and "lsb" tests for the RC4dkip cipher.

The results obtained back-boned our point of view and our proposal. To test the cipher with scenario one is the best option for detecting any failures by Practrand. While choosing a semi-random or random input is a more challenging case for Practrand.
On the other side, RC4 stream cipher was also tested with the different scenarios (using wolfCrypt library). With the different scenarios proposed, it only failed with using the first scenario (plain text set to zeros). Even with 32 Terabytes of data, scenarios two, three and four did not allow any failure detection by Practrand.

Therefore, implementing these tests still prove, in our point of view, the importance of using these free tools to validate the level of randomness. An attacker; especially today; can leverage from any vulnerability in the ciphered text/image even though this can appear after terabytes of data.

After these tests, it can be deduced that whenever a cryptographer wants to propose a new algorithm, it is important to test the new proposal using the simple methodology proposed. This will either highlight some flaws in the considered cipher or it can help in choosing/building a better cryptographic library that can be used without jeopardizing the security level. Using these free tools which are easy-implemented ones can validate the randomness of any cipher in the field of symmetric cryptography.

4 Conclusion

As stated in the beginning of this paper, the importance of randomness is investigated by many researchers who work in the domain of cryptography. It is crucial to have a high level of randomness in any cipher text. There are many tests that are used to measure the randomness of the cipher text, such as the correlation, the Probability Density Functions and many more. As the randomness of the cipher text decrease, the more vulnerabilities are born with it. Randomness plays a major role in cryptography and the major goal for any cryptographer is to ensure the safety and the reliability of the proposed algorithm. Validating the randomness of the ciphered outputs using the methodology proposed in this paper is quite simple yet efficient. The approach proposed relies on using TestU01 and Practrand tests that are originally designed to test the randomness of RNGs. This approach can only be used for symmetric ciphers since we are changing the key, or the initial vector or the Nonce and considering the worst case scenario for the plaintext (all zeroes). These parameters usually exist in symmetric approaches only. Different ciphered outputs in different well-known algorithms were tested and these ciphers are usually used in securing different multimedia content. The results obtained showed the following: RC4 in both WolfCrypt and Libgcrypt libraries as well as ChaCha implemented in WolfCrypt failed Practrand test. In addition, the new proposed RC4 with an additional diffusion layer failed TestU01 and Practrand in one of its coding versions, RC4Dkip. Even if the majority of the implemented algorithms succeeded TestU01 and Practrand, it was shown that others did not. Considering other scenarios that can be also tested, as stated in this paper, the easiest one to detect any failure with is the first scenario, yet it is the hardest for the cipher to pass. To conclude, it is important to ensure the required randomness by availing all the tools that are available to do so. This interpretation of results provides the cryptographer with a way to validate the randomness of cryptographic ciphers before risking the security or breach of data.

Acknowledgement

This paper was partially supported by funds by the EIPHI Graduate School (contract "ANR-17-EURE-0002"). Simulations were conducted on the servers of the "Mésocentre de calcul de Franche-Comté". We would like to thank them for accepting our request and for giving us access to their machines.

References

1. Andem, V.R.: A cryptanalysis of the tiny encryption algorithm. Ph.D. thesis, University of Alabama (2003)
2. Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., Tokita, T.: Camellia: A 128-bit block cipher suitable for multiple platforms—design andanalysis. In: International Workshop on Selected Areas in Cryptography, pp. 39–56. Springer (2000)

3. Bernstein, D.J.: Chacha, a variant of salsa20. In: Workshop Record of SASC, vol. 8, pp. 3–5 (2008)
4. Biryukov, A., Perrin, L.P.: State of the art in lightweight symmetric cryptography (2017)
5. Boesgaard, M., Vesterager, M., Pedersen, T., Christiansen, J., Scavenius, O.: Rabbit: A new high-performance stream cipher. In: International Workshop on Fast Software Encryption, pp. 307–329. Springer (2003)
6. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: Present: An ultra-lightweight block cipher. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 450–466. Springer (2007)
7. Coppersmith, D.: The data encryption standard (des) and its strength against attacks. IBM journal of research and development **38**(3), 243–250 (1994)
8. Doty-Humphrey, C.: Practically random: C++ library of statistical tests for rngs. URL: https://sourceforge.net/projects/pracrand (2010)
9. Fisher, R.A.: Statistical methods for research workers. In: Breakthroughs in statistics, pp. 66–70. Springer (1992)
10. Hatzivasilis, G., Fysarakis, K., Papaefstathiou, I., Manifavas, C.: A review of lightweight block ciphers. Journal of Cryptographic Engineering **8**(2), 141–184 (2018)
11. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.S., Lee, C., Chang, D., Lee, J., Jeong, K., et al.: Hight: A new block cipher suitable for low-resource device. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 46–59. Springer (2006)
12. Karn, P., Metzger, P., Simpson, W.: The esp triple des transform. Tech. rep. (1995)
13. Kaukonen, K., Thayer, R.: A stream cipher encryption algorithm "arcfour" (1999)
14. Kendall, M.G., Smith, B.B.: Randomness and random sampling numbers. Journal of the royal Statistical Society **101**(1), 147–166 (1938)
15. Koch, W., Schulte, M.: The libgcrypt reference manual. Free Software Foundation Inc pp. 1–47 (2005)
16. Kwasnicki, M.: Strong encryption for small payloads on Arduino (2018). URL `https://kwasi-ich.de/blog/2018/03/05/mcu_encryption/`. [Online; 2018]
17. Lai, X., Massey, J.L.: A proposal for a new block encryption standard. In: Workshop on the Theory and Application of of Cryptographic Techniques, pp. 389–404. Springer (1990)
18. L'Ecuyer, P., Simard, R.: Testu01: Ac library for empirical testing of random number generators. ACM Transactions on Mathematical Software (TOMS) **33**(4), 22 (2007)
19. Lemire, D.: testingRNG (2018). URL `https://github.com/lemire/testingRNG`. [Online; 2018]
20. Manogaran, G., Lopez, D., Thota, C., Abbas, K.M., Pyne, S., Sundarasekar, R.: Big data analytics in healthcare internet of things. In: Innovative healthcare systems for the 21st century, pp. 263–284. Springer (2017)
21. Marsaglia, G.: Diehard test suite. Online: http://www. stat. fsu. edu/pub/diehard **8**(01), 2014 (1998)
22. Pearson, K.: X.on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **50**(302), 157–175 (1900)
23. Rogaway, P.: Nonce-based symmetric encryption. In: International Workshop on Fast Software Encryption, pp. 348–358. Springer (2004)
24. Schneier, B.: Fast software encryption, cambridge security workshop proceedings (1994)
25. Schneier, B.: Applied cryptography: protocols, algorithms, and source code in C. john wiley & sons (2007)
26. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: Twofish: A 128-bit block cipher. NIST AES Proposal **15**, 23 (1998)
27. Steele Jr, G.L., Lea, D., Flood, C.H.: Fast splittable pseudorandom number generators. In: ACM SIGPLAN Notices, vol. 49, pp. 453–472. ACM (2014)
28. wolfSSL User Manual: User Manual – Version 3.9.0, wolfSSL (2016). URL `https://www.wolfssl.com/docs/wolfssl-manual/`. [Online; 2016]
29. Walker, J.: Ent: a pseudorandom number sequence test program. Software and documentation available at/www. fourmilab. ch/random/S (2008)
30. Wheeler, D.J., Needham, R.M.: Tea, a tiny encryption algorithm. In: International Workshop on Fast Software Encryption, pp. 363–366. Springer (1994)
31. Wheeler, D.J., Needham, R.M.: Correction to xtea. Unpublished manuscript, Computer Laboratory, Cambridge University, England (1998)
32. Wille, C.: Storing passwords-done right. last updated **1** (2004)
33. Wu, H.: The stream cipher hc-128. In: New stream cipher designs, pp. 39–47. Springer (2008)
34. Wu, W., Zhang, L.: Lblock: a lightweight block cipher. In: International Conference on Applied Cryptography and Network Security, pp. 327–344. Springer (2011)
35. Yu, Y., Yang, Y., Fan, Y., Min, H.: Security scheme for rfid tag. Auto-ID Labs Fudan University, White Paper (2006)