CryptRndTest: An R package for testing the cryptographic randomness

Haydar Demirhan[1]
Hacettepe University, Department of Statistics
06800 Beytepe Ankara Turkey
and
RMIT University, School of Science
Mathematical and Geospatial Sciences
3001 Melbourne Australia

Nihan Bitirim
Council of Higher Education
06800 Cankaya Ankara
Turkey

Abstract In this article, we introduce the R package CryptRndTest that performs eight statistical randomness tests on cryptographic random number sequences. The purpose of the package is to provide a software for the implementation of recently proposed cryptographic randomness tests utilizing goodness-of-fit tests superior to the usual chi-square test in terms of statistical performance. Most of the tests included by CryptRndTest are not conducted by available software such as the R package RDieHarder or the C library TestU01. Chi-square, Anderson-Darling, Kolmogorov-Smirnov, and Jarque-Bera goodness-of-fit procedures are applied along with cryptographic randomness tests. CryptRndTest utilizes multiple precision floating numbers for sequences longer than 64-bit by the use of package Rmpfr. By this way, included tests are applied precisely for higher bit-lengths. CryptRndTest provides a user friendly interface for eight existing and recently proposed cryptographic randomness tests. As an illustrative application, CryptRndTest is used to test available random number generators in R.

Introduction

Cyptographic random numbers constitute the heart of ciphering processes. Security of the transmitted information is mostly based on the quality of random numbers used to cipher the information. Due to the efficiency considerations, pseudo random numbers that ensure some hard-to-achieve properties are used for ciphering in practice. There are a considerable amount of pseudo random number generators (RNG's) in the literature of cryptography. Suitability of these RNG's for use in cryptographic applications is evaluated by using statistical randomness tests that are specifically designed to test randomness at the level required for ciphering processes.

In a cryptographic randomness test, first, empirical distribution of a test statistic is obtained over a random number sequence by various data manipulations. Then, a statistical goodness-of-fit test is applied to evaluate significance of the difference between the empirical distribution and its theoretical counterpart at a predetermined level of significance. The need for a certain level of randomness to ensure unpredictability in cryptography context makes procedures used to check cryptographic and classical randomness different from each other. The manipulations of random number sequences are required to make the cryptographic randomness tests more sensitive to small deviations from the exact randomness than their classical counterparts. The null hypothesis of the test is "H_0 : Sequences generated by the RNG of interest are random." There are more than a hundred alternative tests for the evaluation of cryptographic randomness (L'Ecuyer and Hellekalek, 1998).

In the literature, some of these tests are grouped into test batteries or test suites (L'Ecuyer and Simard, 2007; Marsaglia and Tsang, 2002). A detailed review of test batteries is given by Demirhan and Bitirim (2015a). To be qualified as suitable, an RNG should be identified as random in a predetermined portion or all of the tests in a test battery. The basic test battery is introduced by Knuth (1998, 1981, 1969). Then, Marsaglia (1996) introduced the Diehard test battery composed of 12 randomness tests. Disadvantages of Diehard test battery were overcame by another test battery called Dieharder that is introduced by Brown et al. (2014). Dieharder includes 26 cryptographic randomness tests. It is an improvement of Diehard battery, provides a user friendly interface and a useful open source toolset for users of random numbers (Brown et al., 2014). The Dieharder test battery is implemented

[1]Corresponding author. e-mail:haydar.demirhan@rmit.edu.au

by the R package **RDieHarder** prepared by Eddelbuettel and Brown (2014). At the time of writing, Windows and OS X binaries are not available for this package. US National Institute of Standards and Technology developed the NIST battery composed of 16 tests (Sýs et al., 2014; Sýs and Říha, 2014; Rukhin et al., 2010; Rukhin, 2001; Soto, 1999). The NIST battery is still used as a straightforward tool for formal certifications and accepted as a standard test battery. Sadique et al. (2012) reviewed the tests included in NIST test battery. A suite of test batteries, TestU01, was introduced by L'Ecuyer and Simard (2007, 2014). TestU01 is a C library that combines most of the available randomness tests and RNGs in six test batteries (McCullough, 2006; L'Ecuyer and Simard, 2007). There are also smaller scale test batteries in terms of extensiveness. ENT was proposed by Walker (2014) that contains 5 statistics and tests. The Helsinki test battery is based on Ising model and random walks on lattices and was proposed by Vattulainen et al. (1995). The Crypt-X test battery, which includes 6 tests, was developed by Information Security Research Center at Queensland University of Technology (Sýs and Říha, 2014; Soto, 1999). SPRNG test battery includes some tests from the battery of Knuth (Mascagni and Srinivasan, 2000). Ruetti (2004) combined Knuth, Helsinki, Diehard, and SPRNG batteries and proposed a test battery consisting of 37 statistical and physical tests.

In addition to the tests included by test batteries, there are recently proposed cryptographic randomness tests that are not performed by test batteries. Maurer (1992) proposed a statistical test for random bit generators. Hernandez et al. (2004) proposed a new test called Strict Avalanche Criterion (SAC). Ryabko et al. (2004) proposed an adaptation of well-known chi-square test. This test is more efficient than the usual chi-square test in small samples. "Book Stack" and "Order" tests were proposed by Ryabko and Monarev (2005) for testing binary random bit sequences. Doganaksoy et al. (2006) proposed three randomness tests based on random walk process. Advantage of these tests is that it is possible to calculate exact probabilities corresponding to the test statistics. "Topological Binary Test" was introduced by Alcover et al. (2013) to test randomness in bit sequences. It counts different bit patterns of pre-determined length in a sequence of random bits.

Availability of a software for the implementation of a test battery or even that of an individual cryptographic randomness test is a critical issue on the usefulness of related test or battery. The library TestU01 is developed on ANSI C; hence, it is compiled by GNU tools instead of today's C compilers. Although TestU01 performs a wide variety of tests and their combinations, it lacks flexibility of implementation. Because the battery Dieharder is implemented by an R package, namely **RDieHarder**, it is more applicable and user-friendly than TestU01. However, unavailability of Windows and OS X binaries can be seen as a disadvantage that decreases its accessibility. A package for the implementation of the NIST battery is prepared on SUN workstation using ANSI C (Rukhin et al., 2010). Rukhin et al. (2010) provides a user guide for setting up the package and running the included tests. Ease of implementation of NIST battery is similar with TestU01. For the implementation of individual randomness tests, there are also numerous R packages such as **randtests** or **DescTools**. Although some of the tests included by these packages are also used to evaluate cryptographic randomness, they cover neither recently proposed tests nor those developed specifically to test cryptographic randomness.

The usual chi-square test is applied with nearly all of the cryptographic randomness tests in the literature. The mentioned implementations including those covered by R automatically apply chi-square test. However, there are numerous alternatives to chi-square goodness-of-fit test such as Kolmogorov-Smirnov, Anderson-Darling, or Jarque-Bera. It is apparent that because statistical qualities of these tests are better than the chi-square test, there will be a gain in performance of cryptographic randomness tests applied with better goodness-of-fit tests. Thus, we need a software that is capable of conducting actual cryptographic randomness tests such as topological binary, book stack, etc. with goodness-of-fit tests better than usual chi-square in statistical performance. When the range and variety of cryptographic randomness tests implemented by software and practicability of available software are considered, the new software should effectively implement new tests with various goodness-of-fit tests and has a user-friendly interface. The package **CryptRndTest** contributes to satisfy this need.

The aim of this article is to describe and illustrate use of the R package **CryptRndTest** (currently in version 1.2.2) that performs some of recently proposed and basic cryptographic randomness tests. The article is mainly based on the paper of Demirhan and Bitirim (2015b). The package includes the functions `adaptive.chi.square`, `birthday.spacings`, `book.stack`, `GCD.test`, `topological.binary`, and `random.walk.tests` to perform adaptive chi-square, birthday spacing's, book stack, greatest common divisor, topological binary tests, and three tests based on the random walk process, respectively. To the best of our knowledge, the adaptive chi-square, topological binary, and the tests based on the random walk process are first implemented by a software with **CryptRndTest**. In addition to the chi-square procedure, these functions apply Anderson-Darling, Kolmogorov-Smirnov, and Jarque-Bera procedures when suitable. Because statistical performances of goodness-of-fit tests differ under various conditions, application of different goodness-of-fit procedures is a beneficial feature. This is another important utility of **CryptRndTest**. In addition, it has the following auxiliary functions: `GCD`, `GCD.q`, `GCD.big`, `Strlng2`, `toBaseTwo`, `toBaseTen`, and `TBT.criticalValue` to compute greatest common divisor under different conditions of inputs, approximately calculate the Stirling number of the second kind when the inputs are large, make base conversions precisely with large inputs, and calculate critical values for topological binary test.

The paper is organized as follows: in the next section, methodologies of the tests included in **CryptRndTest** are briefly given. Details of algorithms used to manipulate integer and bit sequences are mentioned, and applications of goodness-of-fit procedures performed by **CryptRndTest** are clarified. Parameter settings and limitations for each test are mentioned. Finally, as an illustrative application of **CryptRndTest**, random number generators available in R are tested by using the proposed package under different sequence and bit-length conditions. By this application, implementation performance of the package is analyzed, recently proposed tests are evaluated, and usage of **CryptRndTest** is illustrated.

Performed tests

Adaptive chi-square

Adaptive chi-square test was introduced by Ryabko et al. (2004). It is empirically demonstrated by Ryabko et al. (2004) that the adaptive chi-square test is more efficient than the classical chi-square test in the identification of non-random patterns in samples smaller than those required by the chi-square test. For example, when we work with 64-bit numbers the length of the alphabet is 2^{64}; hence, we need to have a sequence of length greater than $5 \cdot 2^{64}$ to apply the classical chi-square test safely. The logic behind the test is to divide the alphabet into subsets and perform chi-square test over subsets instead of individual elements of the sample. By this way, subsets are considered as a new alphabet and a new null hypothesis and its alternative are formed over the subsets. Because the number of categories required to test new hypotheses is equal to the number of subsets, the chi-square test is applied with much smaller samples. To conclude randomness, it is expected to observe a uniformity in the distribution of input numbers into the subsets. Deviations from this uniformity are detected by the adaptive chi-square test.

The function `adaptive.chi.square()` is called to apply the test. It implements the following pseudo-code algorithm:

Algorithm 1.

1. Input data as a matrix of bits or a vector of integers, the number of subsets (S) that the alphabet will be divided into, and proportion of training data set;
2. If data is represented by bits, transform data to base-10;
3. Divide whole data set into training and testing subsets with regarding input weights;
4. Identify the numbers that are seen in the sequence of interest at least once;
5. Find the frequency of occurrences for each element of alphabet in training and testing subsets;
6. For $i = 1, \ldots, S$, find the frequency of elements that are seen i-times in the training and testing subsets;
7. Apply the two-sample chi-square test with the expected and observed counts obtained at the previous step over the training and testing subsets, respectively;
8. Return value of the test statistic, corresponding p-value, and the decision on the null hypothesis.

While working with integers, the alphabet corresponds to the range of considered numbers. For instance, if 32-bit numbers are being tested, the alphabet in Algorithm 1 includes the numbers between 0 and $2^{32} - 1$. At step 4, we do not form whole alphabet, instead we count the numbers (words) that are seen at least once; and hence, the rest of the numbers have zero count. At step 7, the degrees of freedom of the test is $S - 1$.

Parameters of the adaptive chi-square test are: weight of training and testing samples (r), the length of the considered number sequence (n), and the number of subsets (S) that the alphabet is divided into. Ryabko et al. (2004) do not give strict rules for the determination of values of these parameters. They suggest to run some experiments to find the values of parameters that provide the highest statistical performance such as power and specificity. Because such a study would not be cost-effective for an individual application of the test, at least, the user may evaluate sensitivity of test results to the values of S and r. In the function `adaptive.chi.square()`, we set $r = 0.5$ by default. The value of S is set by user. That of n is determined by the length of input data. Because input data is a random sample from the RNG of interest, the value of n should be increased with increasing bit-length to successfully represent the range of numbers that will be generated by the RNG. When bit-length is greater than 64, we utilize the package **Rmpfr** to work with higher precision.

Algorithm complexity of the function `adaptive.chi.square()` is $O(n^2)$ in the worst case. Required memory is directly related with the length of input sequence. Due to the algorithm complexity of the function used to identify unique numbers at step 4, implementation time of the function `adaptive.chi.square` increases quadratically along with the length of input sequence.

Birthday Spacings

The Birthday Spacings test was given by Marsaglia and Tsang (2002). It focuses on the number of duplicated values of spacings between ordered birthdays among a year of pre-determined length. The observed duplication patterns in input numbers are compared with the patterns that should be observed under randomness. Thus, birthday spacings test detects deviations from randomness by focusing on repetition frequency of numbers to ensure uniformity. Marsaglia and Tsang (2002) propose that the number of duplicated values is approximately distributed according to the Poisson distribution. They also derive an expression for the mean rate of the Poisson distribution.

The function `birthday.spacings()` is employed to run the test. It implements the following pseudo-code algorithm:

Algorithm 2.

1. Input data as a vector of integers of size n, the number of birthdays (m), the length of year (N), the mean rate of the theoretical Poisson distribution (λ), and the number of classes (k) that is constructed for goodness-of-fit tests;
2. Reshape the first $m \cdot \lfloor n/m \rfloor$ elements of input vector as a matrix of $\lfloor n/m \rfloor$ rows and m columns;
3. Sort each row of the matrix of step 2 according to the values in columns;
4. For each row, find the distance between columns of the sorted matrix by extracting the values in the columns at the previous step;
5. Count duplicated values among the distances obtained at step 4;
6. Calculate class probabilities over the Poisson distribution with mean rate λ for $x = 0, \ldots, k$, and assign the rest of probability mass to the $(k+1)$-th class;
7. Calculate expected frequencies corresponding to the probabilities obtained at the previous step;
8. Replicate the expected counts to form the corresponding sample;
9. Apply the Anderson-Darling test to compare goodness-of-fit of the samples obtained at steps 5 and 8;
10. Apply the Kolmogorov-Smirnov test to compare goodness-of-fit of the samples obtained at steps 5 and 8;
11. Construct frequency table of the counts obtained at step 5;
12. Apply chi-square test over the frequency tables obtained at steps 7 and 11;
13. Return the values of test statistics, corresponding p-values, and decisions on the null hypothesis.

At step 2 of Algorithm 2, each row of the reshaped matrix includes birthdays in columns. Total number of rows determines the size of sample that is used in goodness-of-fit tests applied at steps 9, 10, and 12. Manipulation of the input vector according to the birthday spacings test is completed at step 5. This manipulation produces the empirical sample in testing the goodness-of-fit to Poisson distribution. The Anderson-Darling test at step 9 is applied by using function `ad.test` from the package **ksamples**. The Kolmogorov-Smirnov test at step 10 is applied by using function `ks.test` from the package **stats**.

Marsaglia and Tsang (2002) give some insight into the optimal values of parameters. The mean rate is $\lambda = m^3/(4n)$. They state that for an RNG, it is harder to pass this test for increasing values of either m or n. Specifically, the case with $m = 4096$ and $n = 2^{32}$ is qualified as a compelling setting for 32-bit generators. Length of the input sequence is another important parameter. Because the size of sample used in testing the goodness-of-fit is equal to $\lfloor n/m \rfloor$, the length of the input sequence (n) should be chosen large enough to apply the goodness-of-fit tests appropriately.

Algorithm complexity of the function `birthday.spacings()` is $O(n^2)$ in the worst case. The limitation of `birthday.spacings()` is directly related with the value of m. For all combinations of m and n suggested by Marsaglia and Tsang (2002), λ is equal to 4. Following this logic, when $n = 2^{64}$ the value of m giving $\lambda = 4$ is 6,658,043. In this case, for a reliable application of goodness-of-fit tests at steps 9, 10, and 12, we need at least 133,160,860 integers and correspondingly 8,522,295,040 bits. For bit lengths higher than 32, the value of λ can be taken as 2. For instance, when $n = 2^{64}$, the corresponding value of m is 5,284,492. Thus, decreasing the value of λ does not overcome the need for a huge data set for a reliable testing. Note that use of huge data set for testing is a memory consuming operation.

Book Stack

The Book Stack test was proposed by Ryabko and Monarev (2005). Positions of the numbers on a stack are taken into consideration. In this test, randomness implies that frequency of finding each number at each position is equally likely. Departures from this equality mean that some of the words are seen more frequently in contrast to the nature of randomness. The book stack test focuses on non-uniform patterns and frequent repetitions of input numbers to detect deviations from randomness by means of unexpected autocorrelation patterns and non-uniformity.

The function `book.stack()` implements the following pseudo-code algorithm to run the test:

Algorithm 3.

1. Input data as a matrix of bits or a vector of integers and the number of subsets (k) that the alphabet will be divided into;
2. If data are represented by bits, transform data to base-10;
3. Form an array that includes the numbers from 1 to the number of unique words in the input sequence;
4. Write each element of the input vector in place of the first element of the array formed at the previous step, and move the other elements except the one written to the first cell of the array one step right;
5. Record the array obtained at the previous step;
6. Go back to step 4 until all elements of the input vector are taken into account;
7. Divide the whole alphabet into k non-overlapping subsets $(A_1, A_2, \ldots, A_k)$;
8. For each subset of alphabet, find the frequency of occurrences of the number corresponding to the position of each element of input vector in the arrays formed at steps 4 and 5;
9. Apply chi-square test with expected counts equal to $n \cdot A_i$, where $i = 1, \ldots, k$ and n is the length of input vector or number of columns of input matrix;
10. Return the value of test statistic, corresponding p-value, and decision on the null hypothesis.

In order to get an integer number of subsets, the length of input vector should be determined to get an integer as the length of subsets. Optimal value for the length of input vector is given as $n \approx B \cdot 2^{B/2}$, where B is the bit-length of considered RNG (Ryabko and Monarev, 2005; Doroshenko and Ryabko, 2006; Doroshenko et al., 2006). For an appropriate determination of number of subsets, k, Ryabko and Monarev (2005) suggest performing an empirical study. As for an appropriate bit-length, it is mentioned by Ryabko and Monarev (2005) that it is hard to set up a sensible test with much higher bit-lengths.

Algorithm complexity of the function `book.stack()` is $O(n^2)$ in the worst case. The limitation of the Book Stack test is based on the bit-length of considered RNG. For example, for $B = 64$ the length of input vector is calculated as $1.37 \cdot 10^{11}$ and we need 1 terabyte memory whereas the memory requirement is 4 megabytes for $B = 32$. Due to both memory and sensibility issues, it is not appropriate to work with high bit-lengths such as 64.

Greatest Common Divisor

Two tests proposed by Marsaglia and Tsang (2002) are based on the number of required iterations (k) and the value of greatest common divisor (GCD) obtained in the GCD operation. When perceived as random variables, both k and GCD are independently and identically distributed and their distributions can be obtained under randomness. Marsaglia and Tsang (2002) derived distributions of k with an empirical study and that of GCD theoretically under the null hypothesis of randomness. Departures from randomness imply nonconformity between empirical and theoretical distributions of k and GCD. Thus, these tests focus on the deviations from independence and uniformity.

The function `gcd.test()` is called to apply the test. The following pseudo-code algorithm is implemented by `gcd.test()` when all of the goodness-of-fit tests are set to `TRUE`:

Algorithm 4.

1. Input data as an $N \times 2$ matrix of integers, mean and standard deviation of theoretical normal distribution of k;
2. Constitute a pair of numbers from each row of input matrix;
3. Apply GCD operation to each pair formed at the previous step;

4. Store values of k for N pairs;
5. If obtained GCD is less than 3, store it as 3 and if that of GCD is greater than 35, store it as 35;
6. Generate a random sample of size N from normal distribution with input values of mean and standard deviation.
7. If the tests based on k will be conducted, go to the next step, otherwise go to step 13;
8. Apply the two sample Kolmogorov-Smirnov test in a two-sided setting to samples obtained at steps 4 and 6;
9. Apply the chi-square test to samples obtained at steps 4 and 6;
10. Standardize the values of k by using its empirical mean and standard deviation;
11. Apply the Jarque-Bera test to the standardized sample of step 10;
12. Apply the Anderson-Darling test to samples obtained at steps 4 and 6;
13. If the tests based on GCD will be conducted, go to the next step, otherwise go to step 19;
14. Construct the cumulative distribution function (cdf) of the probability function (pf) of GCD given by Marsaglia and Tsang (2002).
15. Obtain theoretical frequencies for GCD over the cdf of step 14. Specifically, if theoretical frequency of GCD is less than 3, store it as 3 and if that of GCD is greater than 35, store it as 35;
16. Replicate the expected counts to form the corresponding sample;
17. Apply the two sample Kolmogorov-Smirnov test in a two-sided setting to samples obtained at steps 5 and 16;
18. Apply the chi-square test to samples obtained at steps 5 and 16;
19. Return the values of calculated test statistics, corresponding p-values, and decisions on the null hypothesis.

Mean and standard deviation of theoretical normal distribution for bit lengths other than 32 are not given by Marsaglia and Tsang (2002). We conducted extensive empirical studies, details of which are mentioned in following sections, to obtain these parameters and tabulated obtained values in Table 3.

When bit-length is increased, corresponding value of GCD mostly becomes greater than 35; hence, the operation at step 15 of Algorithm 4 gets unreasonable. Thus, we observe that it is not appropriate to conduct tests based on GCD for high bit-lengths such as 128.

The Kolmogorov-Smirnov and chi-square tests at steps 8 and 17, and 9 and 18 are applied by using functions `ks.test` and `chisq.test` from the package **stats**, respectively. The Jarque-Bera test at step 11 is implemented by using the function `jarque.bera.test` from the package **tseries**. The Anderson-Darling test is applied by using the function `ad.test` from the package **ksamples**.

Calculations of the number of required iterations and the value of GCD are time consuming tasks for bit-lengths greater than 64. To overcome this difficulty, we prepared three functions to calculate GCD-related variables. The first function `GCD.q` computes the number of required iterations, the value of GCD, and the sequence of partial quotients by using the Euclidean algorithm. The function `GCD` is the recursive version of the Euclidean algorithm and it only provides the number of required iterations and the value of GCD. The function `GCD.big` applies the Euclidean algorithm over multiple precision floating point numbers using the **Rmpfr** and provides all three outputs related with the GCD operation. While `GCD` is the fastest one, `GCD.big` gives the most precise results. It is also possible to use the binary GCD algorithm to decrease the implementation time. However, in this case it is not possible to apply tests over the number of required iterations of the Euclidean algorithm. When the GCD operation is done recursively, the algorithm complexity of `gcd.test()` is $O(\log(a))$, where a is the maximum initial input to the recursive algorithm. Memory requirement for GCD tests is directly related with the value of N.

Random walk tests

In the literature, binary sequences are analyzed in detail by using random walk process. Doganaksoy et al. (2006) proposed three tests based on the random walk stochastic process. In a random walk process, magnitude or direction of each change is determined by chance; hence, a random walk is random if increment and decrement probabilities are equal to each other. Therefore, random walk processes provide a good basis for randomness. In a random walk, a part of sequence that intersects the x-axis with two successive points is called excursion, and over all excursions, the maximum distance from the x-axis is defined as height, and the vertical distance between minimum and maximum points over y-axis is called expansion. Thus, we have three characteristics of random walk process to observe deviations from randomness. The corresponding tests are called Random Walk Excursion, Random Walk Height, and Random Walk Expansion. If there is a trend in the process, input sequence fails in the excursion test. The height test focuses on the moves with very low or high magnitude to detect non-randomness. The expansion test focuses on the anomalies in amplitude of the walk to identify non-random patterns. Because the exact probabilities corresponding to test statistics are calculated, the tests proposed by Doganaksoy et al. (2006) are also applicable for small sample sizes.

The function `random.walk.tests()` is called to apply three tests, selectively. The following pseudo-code algorithm is implemented by `random.walk.tests()` when all of the tests are to be applied:

Algorithm 5.

1. Input data as a matrix of bits of dimension $B \times k$, where B is the bit length and k is the length of input sequence;
2. Transform the input values from $\{0, 1\}$ to $\{-1, 1\}$;
3. To apply the expansion, excursion, and height tests go to steps 4, 6, and 7, respectively;
4. For each non-overlapping set of length B, sum adjacent bits starting from the first bit and increasing by one at each iteration (By this way, we get B summations for each number of interest).
5. For the Expansion test, count and store the summations of the previous step equal to zero;
6. For the Excursion test, calculate the maximum summation and the absolute value of the minimum summation among those of step 4 and store their sum;
7. For the Height test, store absolute maximum of summations obtained at step 4;
8. Calculate theoretical cdf's and pf's for the tests regarding bit-lengths and probabilities tabulated by Doganaksoy et al. (2006).
9. Calculate empirical cdf's and pf's over the counts obtained at steps 5, 6, and 7;
10. Replicate the expected and empirical pf's to form the corresponding samples;
11. Apply the Anderson-Darling test to samples obtained at the previous step;
12. Apply the two sample Kolmogorov-Smirnov test in a two-sided setting to samples obtained at step 10;
13. Apply the chi-square test to samples obtained at step 10;
14. Return the values of calculated test statistics, corresponding p-values, and decisions on the null hypothesis.

The Anderson-Darling test at step 9 is applied by using function `ad.test` from the package **ksamples**. The Kolmogorov-Smirnov test at step 10 is applied by using function `ks.test` from the package **stats**. The chi-square test at step 11 is the classical application of the test without using a predefined function. If one of the tests is not applied, all the results related with that test in output are set to -1.

Algorithm complexities of expansion, excursion, and height tests are $O(B)$, $O(B\lfloor k \cdot B \rfloor)$, and $O(B\lfloor k \cdot B \rfloor)$, respectively. The limitation of the tests is unavailability of theoretical cdf's for bit-lengths other than 32, 64, 128, and 256. Therefore, using the information given by Doganaksoy et al. (2006) the excursion is applied for bit-lengths of 16, 32, 64, 128, and 256; the height test is applied for bit-lengths of 64, 128, 256, 512, and 1024; and the expansion test is applied for bit-lengths of 32, 64, and 128. Although the size of required memory increases along with the length of input sequence, it is possible to apply the tests with reasonable sequence lengths without causing memory pressure.

Topological binary

The topological binary test was proposed by Alcover et al. (2013) to test the randomness in bit sequences. The logic behind the test is based on the number of different fixed-length bit patterns in a bit sequence. Frequency of distinct non-overlapping bit patterns over the sequence of interest is influential on the test result. In case of randomness, we expect to have many different bit patterns in the input sequence. The main strength of the topological binary test is that it focuses on the number of bit patterns rather than frequency of occurrence of numbers. Because the exact distribution of test statistic is derived, it is possible to apply the test for short bit sequences.

The function `topological.binary()` implements the following pseudo-code algorithm to run the test:

Algorithm 6.

1. Input data as a $B \times k$ matrix of bits, where B is the bit-length and k is the length of considered number sequence, and the critical value;
2. Find and store non-overlapping blocks of length B;
3. Count the number of different B-bit patterns that appear across all the k blocks;
4. If the result of step 3 is less than one, then reject the null hypothesis;
5. else if the result of step 3 is greater than $\min(k, 2^B)$, then do not reject the null hypothesis;
6. else if the result of step 3 is less than the input critical value, then reject the null hypothesis;
7. else do not reject the null hypothesis;
8. Return the result of step 3 as the value of test statistic and the decision on the null hypothesis.

Although the exact distribution of test statistic is derived by Alcover et al. (2013), calculation of the Stirling numbers of the second kind with large inputs is required with bit-lengths greater than 16 for the calculation of cdf of the tests statistic. Therefore, it is hard to obtain the critical value of the test for large bit-lengths by using available functions in R packages such as the function `Stirling2` of **copula**. This case is a limitation of the function `topological.binary()`. To overcome this limitation of the test, we prepared the function `TBT.CriticalValue` to calculate required critical values for testing. Algorithm complexity of the function `topological.binary()` is $O(n^2)$ in the worst case. The required memory to run the topological binary test is related with the value of k.

Auxiliary functions

The package **CryptRndTest** has seven auxiliary functions, namely `Strlng2()`, `GCD()`, `GCD.q()`, `GCD.big()`, `toBaseTwo()`, `toBaseTen()`, and `TBT.CriticalValue()`. These functions are also suitable for individual use. `Strlng2()` is used to calculate critical values for the topological binary test implemented by `TBT.CriticalValue()`. `GCD()` and `GCD.q()` are called to calculate the greatest common divisor in the GCD test implemented by `gcd.test()`. Three possible outcomes of the greatest common divisor operation are the number of iterations, the sequence of partial quotients, and the value of greatest common divisor. `GCD()` provides all of these outcomes for any pair of integers excluding zero. Functions `toBaseTwo()` and `toBaseTen()` are used for base conversion from base 2 to 10 and vice versa for large integers.

The function `Strlng2()` is used to compute natural logarithm of Stirling numbers of the second kind for large values of inputs in an approximate manner by the approaches of Bleick and Wang (1974) and Temme (1993). In this approach, Lambert W functions are employed at the log scale to overcome memory overflows.

Due to the large factorials in the calculation of Stirling numbers of the second kind, it is nearly impossible to compute exact cdf of the topological binary test statistic for higher bit lengths without memory flows in R. The function `TBT.CriticalValue()` implements an approach for the calculation of cdf and approximately computes the required critical value for the topological binary test at a given level of α. Because `TBT.CriticalValue()` utilizes `Strlng2()`, accuracy of results decreases with increasing bit lengths and number of words under consideration. It is also possible to make exact calculations by `TBT.CriticalValue()`. In this case, the function `Stirling2` from the package **gmp** is employed instead of `Strlng2()`. Because the **gmp** uses multiple precision arithmetic, implementation time of `TBT.CriticalValue()` considerable increases. User should evaluate the trade off between implementation time and high precision.

Arguments of main and auxiliary functions of **CryptRndTest** package are summarized in Table 1.

A numerical illustration

As a numerical illustration of the package, we employed **CryptRndTest** to test the randomness of RNG's available in R. By this way, we aim to get results of the tests that have not been applied to RNG's of interest yet, figure out implementation performance of **CryptRndTest** under various scenarios, and illustrate some issues on the determination of parameters of the tests for considered scenarios. Note that it is impossible to observe the ability to control type-I error (rejection of randomness hypothesis while it is actually true) for the tests with an empirical study such as conducted in this section.

RNG's of interest are Wichmann-Hill (WH), Marsaglia-Multicarry (MM), Super-Duper (SD), Mersenne-Twister (MT), Knuth-TAOCP-2002 (KT02), Knuth-TAOCP (KT), and L'Ecuyer-CMRG (LE) (see the function `Random` in **base** package for the details of these RNG's). Applied tests are topological binary (TBT), adaptive chi-square (Achi), birthday spacings (BDS), random walk expansion (RWT.Exp), random walk height (RWT.Hei), random walk excursion (RWT.Exc), book stack (BS), and greatest common divisor (GCD). TBT, RWT.Exp, RWT.Hei, and RWT.Exc tests work with binary numbers

Table 1: Usage of test and auxiliary functions of **CryptRndTest** package.

	Function	Call
Test	GCD.test()	GCD.test(x, KS = TRUE,CSQ = TRUE, AD = TRUE, JB = TRUE, test.k = TRUE, test.g = TRUE, mu, sd, alpha = 0.05)
	random.walk.tests()	random.walk.tests(x, B = 64, Excursion = TRUE, Expansion = TRUE, Height = TRUE, alpha = 0.05)
	birthday.spacings()	birthday.spacings(x, m = 128, n = 2^{16}, alpha = 0.05, lambda, num.class = 10)
	adaptive.chi.square()	adaptive.chi.square(x, B, S, alpha = 0.05, bit = FALSE)
	book.stack()	book.stack(A, B, k = 2, alpha = 0.05, bit = FALSE)
	topological.binary()	topological.binary(x, B, alpha = 0.05, critical.value)
Auxiliary	Strlng2()	Strlng2(n, k, log = TRUE)
	GCD()	GCD(x, y)
	GCD.q()	GCD.q(x, y)
	GCD.big()	GCD.big(x, y, B)
	TBT.CriticalValue()	TBT.criticalValue(m, k, alpha = 0.01, cdf = FALSE, exact = TRUE)
	toBaseTen()	toBaseTen(x, m = 128, prec = 256, toFile = FALSE, file)
	toBaseTwo()	toBaseTwo(x, m = 128, prec = 512, num.CPU = 4)

while the rest of tests take integers as input. BDS and RWT tests are applied separately with each of Anderson-Darling, Kolmogorov-Smirnov, and chi-square goodness-of-fit tests, and GCD test is applied separately with each of Anderson-Darling, Kolmogorov-Smirnov, Jargue-Bera, and chi-square goodness-of-fit tests. The total number of applied randomness tests is 21. All the tests are applied at both 0.01 and 0.05 levels of significance and 8, 16, 32, 64, and 128-bit lengths. Considered lengths of random number sequences for each bit-length are given in Table 2.

Table 2: Lengths of random number sequences for different patterns.

	Sequence length		
Bit	Short (I)	Medium (II)	Long (III)
8	256	32768	65536
16	16384	65536	131072
32	32768	131072	262144
64	131072	262144	524288
128	131072	262144	524288

Because we get unreasonable implementation times for longer sequences at the level of 128-bit, the same sequence lengths as 64-bit are considered for 128-bit numbers.

To conduct the adaptive chi-square test, we need to determine the value of argument `S` and the proportions of training and testing samples. The latter one is taken equal. As for the value of `S`, we did not detect a significant change in the test results observed for medium sequence length for all bit-lengths for $S = 2, 3, 4$ in pilot runs. The values greater than 4 increase the implementation time whereas small values decrease resolution. Thus, it is taken as 4 for all bit-lengths to work with a reasonable degrees of freedom in the chi-square test. Also, adaptive chi-square test is applied for all bit-lengths.

Arguments of the birthday spacings test are the number of birthdays (`m`), the length of year (`n`), the mean rate of the theoretical Poisson distribution (`lambda`), and the number of classes (`num.class`), which is used for goodness-of-fit tests. In the experiments, the argument `m` was taken as 8, 128, and 4096 for 8, 16, and 32-bit-lengths, respectively. The argument `n` was set to 2^B, where B is the bit-length. The argument `lambda` was calculated by the formula given by Marsaglia and Tsang (2002). The argument `num.class` was set to 5 and 10 for 8 and 16-bit and higher lengths, respectively.

For the book stack test, length of the sample (`n`) should be determined and data should be prepared according to the value of `n`. Also, the number of subsets that the alphabet will be divided into (`k`) should be determined. The formula proposed by Ryabko and Monarev (2005) is used to calculate the value of `n`, and we set `k`=`n`$/B$.

In the GCD test procedure, tests are conducted for two outputs of GCD operation that number of iterations required to find GCD (k) and GCD (g) itself. The population distribution of k is well approximated by a normal distribution and parameters of the normal distribution are given by Marsaglia and Tsang (2002) for 32-bit integers after an extensive numerical study. We observed that the parameters of population distribution differ for different bit-lengths and conducted a numerical study to figure out the values of parameters for considered bit-lengths. For this study, 10^6 30-bit true random numbers were obtained from the web service "www.random.org." Then, they were converted to 8, 16, 32, 64, and 128-bit numbers. The GCD operation was applied and mean (`mu.GCD`) and standard deviation (`sd.GCD`) of k were obtained as given in Table 3 after checking the normality of the empirical distribution by means of descriptive statistics and Anderson-Darling goodness of fit test. The values obtained for 32-bit are very close to those obtained by Marsaglia and Tsang (2002).

Table 3: Mean and standard deviation of population distribution of k.

Bit	mu.GCD	sd.GCD
8	3.9991	1.6242
16	8.8784	2.3664
32	18.4023	3.4000
64	31.3269	4.3349
128	31.8390	4.3678

As expected, mean of k increases along with bit-length, and it approaches to 35 as treated by Marsaglia and Tsang (2002). The mild increase in the values of standard deviations is due to the increasing range of the numbers that can be generated with a given bit-length. Also, the GCD test is applied for all bit-lengths. However, nearly for all 128-bit random numbers, $g > 35$. Due to the operation done at step 15 of Algorithm 4, it is unreasonable to conduct the GCD test over g for 128-bit numbers.

Topological binary test is also applied for all bit-lengths. Critical values for topological binary test are calculated by using the function `TBT.criticalValue()` for each bit and sequence length combination and presented in Table 4. Because the length of sequence being tested cannot be longer than $2^m - 1$, where m is the bit-length, critical values for medium and long sequences at 8-bit and for long sequences at 16-bit levels are not available in Table 4.

Table 4: Critical values for toplogical binary test.

	$\alpha = 0.01$			$\alpha = 0.05$		
Bit	Short	Medium	Long	Short	Medium	Long
8	153	NA	NA	153	NA	NA
16	14423	41268	NA	14423	41266	NA
32	32767	131066	262129	32767	131066	262129
64	131070	262113	523264	131070	262113	524264
128	131072	262144	524288	131072	262144	524288

NA: not available.

In the application, random numbers were generated by the Function 1 given in R codes of this vignette. Experiments were carried out by the Function 2 of related R codes. In both functions, `RNG` is the number indicating employed pseudo random number generator, `m` is the bit-length, and `len` is the length of the random number sequence. In the function `experiments()`, `cv.TBT` is the critical value of topological binary test, and `mu.GCD` and `sd.GCD` are mean and standard deviation of true distribution of k, respectively.

Random number sequences used for the performance analysis are of medium length given in Table 2 and generated by WH generator under each bit level. Five replications were made for each test. Mean implementation times calculated over five replications are shown in Table 5 in seconds. All variances of implementation times are less than 0.01. BDS, RWT, and BS tests were not applied at all bit-lengths due to reasons explained in relevant sections.

Table 5: Mean implementation time for each test in seconds.

		Tests							
Bit	Length	TBT	Achi	BDS	RWT.Exp	RWT.Hei	RWT.Exc	BS	GCD
8	32768	0.62	2.88	0.46	NA	NA	NA	< 0.01	1.31
16	65536	1.70	5.70	0.46	NA	NA	4.33	0.02	3.74
32	131072	6.68	10.88	2.10	NA	0.26	15.99	4253.01	12.32
64	262144	32.05	86.31	NA	84.21	88.74	64.68	NA	37.36
128	262144	77.16	10121.34	NA	221.16	196.96	149.29	NA	2657.62

TBT: topological binary, Achi: adaptive chi-square, BDS: birthday spacings, RWT: random walk, Exp: expansion, Hei: height, BS: book stack, GCD: greatest common divisor, Length: the length of random number sequence, NA: not available.

Implementation times of all tests from 8 to 64-bit levels are all sufficient. For 128 bits, most of the implementation times of Achi and GCD tests are taken by finding unique values in a sequence composed of multiple precision floating-point (mpf) numbers at step 4 of Algorithm 1 and the value of gcd for mpf numbers at step 3 of Algorithm 4, respectively. For these operations, mpf numbers are used via the package **Rmpfr**. The package **Rmpfr** is based on GMP GNU library and provides an interface from R to the C (Maechler, 2011a,b). Due to the use of mpf numbers via the package **Rmpfr**, there is a considerable increase in implementation time of Achi and GCD tests at 128-bit level.

However, the gain in precision is worth the delay in implementation of these tests. Performances of the tests working with binary numbers are all sufficient at 128-bit level. Implementation time of the BS test exponentially increases along with the bit-length. Although it is reasonable for 32 bits, application of the test for higher bit-lengths requires unreasonable amount of time for implementation.

All the tests were applied at both 0.01 and 0.05 levels of significance. The null hypothesis is "H_0 : Sequences generated by the RNG of interest are random" for all tests. At 0.05 level of significance, test results for all generators of interest are given in Table 6-12. Due to the similarity between results at both levels of significance, those for 0.01 level are omitted.

Table 6: Test results for WH generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	0	0	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	0	1	1	0	1	1	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
CS.k	0	0	1	1	1	1	0	1	1	1	1	1	1	0	1
AD.k	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
JB.k	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	1	0	0	1	1	1	0	1	1	1	1	1	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

For both levels of significance, success rates of RNGs over the total number of applied tests are given in Table 13. The total number of applied tests is given in the last row of Table 6 for each test scenario. For example, because the birthday spacings test is not applied for 64 bit-length, the total number of applied tests is 17 for all sequence lengths. Note that the values given in Table 13 should not be confused with issues related with statistical performance of the tests such as type I error or power. Table 13 represents the proportion of RNG's that did not fail in the given number of tests. In addition, because each test is applied individually, the information presented by Table 13 should not be perceived as the results of application of a test battery.

In general, proportion of success decreases with increasing sequence and bit-lengths. According to proportions of success, performance of WH generator is satisfactory for 16 and 32-bit numbers for all sequence lengths. The reason of getting a decreasing success rate with increasing bit-length is that the random walk tests with all goodness-of-fit tests and GCD test with Jarque-Bera goodness-of-fit test reject the randomness hypothesis while the rest of the tests mostly accept the hypothesis for bit-lengths greater than 32. In detail, WH generator successfully passes both of the TBT and Achi tests nearly in all bit-sequence length combinations. Results of AD and KS goodness-of-fit tests applied under both BDS and GCD tests (with k) are similar, and CS test more likely decides randomness of WH generator. It is unsuccessful in passing the random walk tests for high bit-lengths. BS test concludes WH's randomness under all of the test conditions. GCD with JB goodness-of-fit test rejects the null hypothesis of randomness under all test conditions but the first one. At 0.01 level of significance, there is nearly no change in the results. WH generator passes the GCD test with CS goodness-of-fit test for k at (8, I), (8, II) and (32, I) scenarios, and the BDS test with AD goodness-of-fit test at (16, II).

According to proportions of success, SD generator mostly passes the tests for 16 and 32-bit integers for all sequence lengths, and 8-bit integers for short and long sequences. Detailed test results for SD generator at 0.05 level of significance are similar to that of WH generator for TBT, Achi, BDS, RWT, and BS tests. It is better in GCD test with JB goodness-of-fit test for k. At 0.01 level of significance, CS

Table 7: Test results for SD generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	0	1	1	0	0	0	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
CS.k	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0
AD.k	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1
JB.k	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	1	1	1	0	1	1	0	1	1	0	1	0	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

Table 8: Test results for MT generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	0	1	1	0	0	0	1	0	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	0	1	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	0	1	1	0	1	0	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
CS.k	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
AD.k	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
JB.k	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	0	0	1	1	1	1	1	1	1	0	0	1	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

Table 9: Test results for MM generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	0	1	1	0	0	1	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
CS.k	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
AD.k	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1
JB.k	1	0	1	1	0	1	1	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	1	1	1	0	1	1	1	1	1	1	0	0	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

Table 10: Test results for LE generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	1	1	1	0	0	0	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
CS.k	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1
AD.k	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1
JB.k	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	0	1	1	1	1	1	1	1	1	1	1	1	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

Table 11: Test results for KT02 generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	0	1	1	0	0	0	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	0	0	0	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1
CS.k	1	1	0	1	1	1	1	1	0	1	1	1	0	1	1
AD.k	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
JB.k	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	1	1	1	1	1	1	1	1	1	1	0	1	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

Table 12: Test results for KT generator at 0.05 level of significance.

	Bit-length														
	8			16			32			64			128		
	Sequence length														
Test	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
TBT	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
Achi	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
BDS.AD	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.KS	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-
BDS.CS	0	1	1	0	1	1	-	-	-	-	-	-	-	-	-
RWT															
AD.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
AD.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
AD.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
KS.Exc	-	-	-	1	1	1	1	1	1	1	1	1	0	0	0
KS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
KS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
CS.Exc	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0
CS.Exp	-	-	-	-	-	-	1	1	1	0	0	0	0	0	0
CS.Hei	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0
BS	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-
GCD															
KS.k	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1
CS.k	1	1	1	1	0	1	1	1	1	0	1	1	0	1	1
AD.k	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
JB.k	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
KS.g	1	1	1	1	1	1	1	1	1	0	0	0	-	-	-
CS.g	1	0	1	1	0	1	1	1	1	1	0	1	-	-	-

I: short sequence for given bit-length, II: medium sequence for given bit-length, III: long sequence for given bit-length, AD: Anderson-Darling, CS: chi-square, KS: Kolmogorov-Smirnov, JB: Jarque-Bera, Prop: proportion of success in applied tests, - : not available.

Table 13: Success rates for RNGs over the tests applied by **CryptRndTest**.

Level of Significance	RNG	Bit-length 8			16			32			64			128		
		Sequence length I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
0.01	WH	0.92	0.58	0.50	0.93	1.00	0.86	0.86	0.93	0.93	0.47	0.47	0.47	0.33	0.27	0.33
	SD	0.92	0.58	0.75	0.93	0.93	0.93	0.93	0.93	0.93	0.41	0.47	0.35	0.33	0.33	0.27
	MT	0.92	0.58	0.58	1.00	1.00	0.93	0.93	0.93	0.93	0.47	0.41	0.47	0.33	0.33	0.33
	MM	0.92	0.58	0.75	0.93	0.93	1.00	1.00	0.93	0.93	0.47	0.41	0.35	0.33	0.33	0.27
	LE	0.92	0.67	0.58	0.93	0.93	0.79	1.00	0.93	0.93	0.47	0.47	0.47	0.33	0.27	0.33
	KT	0.92	0.67	0.58	0.93	0.93	0.93	1.00	0.93	0.93	0.41	0.47	0.47	0.27	0.33	0.33
	KT02	0.92	0.67	0.58	1.00	0.93	1.00	1.00	0.93	0.86	0.47	0.41	0.47	0.27	0.33	0.33
0.05	WH	0.83	0.50	0.50	0.93	0.93	0.86	0.79	0.93	0.93	0.47	0.47	0.47	0.33	0.27	0.33
	SD	0.92	0.58	0.75	0.93	0.93	0.86	0.86	0.79	0.86	0.41	0.47	0.29	0.33	0.33	0.27
	MT	0.67	0.50	0.58	0.93	0.86	0.86	0.93	0.86	0.93	0.41	0.41	0.47	0.33	0.33	0.33
	MM	0.92	0.58	0.75	0.93	0.93	0.93	1.00	0.86	0.93	0.47	0.41	0.35	0.33	0.33	0.27
	LE	0.92	0.67	0.58	0.93	0.93	0.79	0.93	0.86	0.93	0.47	0.41	0.47	0.33	0.27	0.33
	KT	0.92	0.58	0.58	0.93	0.86	0.93	1.00	0.93	0.93	0.41	0.41	0.47	0.27	0.20	0.33
	KT02	0.83	0.58	0.42	1.00	0.93	0.93	1.00	0.93	0.79	0.47	0.41	0.47	0.27	0.33	0.33
Number of tests		12	12	12	15	15	15	15	15	15	17	17	17	15	15	15

goodness-of-fit test applied with GCD test cannot reject the null hypothesis for 4 scenarios.

Reaction of the tests for MT, MM, and LE generators is similar to that of WH generator. According to proportions of success, success rates of MT generator are satisfactory for 16 and 32-bit numbers for all sequence lengths; and that of MM generator is very satisfactory for 16 and 32-bit numbers for all sequence lengths, and 8-bit numbers for short and long sequence lengths. Success proportions of LE, KT, and KT02 generators are high for 16 and 32-bit numbers for all sequence lengths, and 8-bit numbers for short sequences. BS test rejects randomness of KT02 generator for 8-bit numbers for all sequence lengths at 0.05 level of significance. However, it cannot reject the null hypothesis for 8-bit numbers for all sequence lengths for $\alpha = 0.01$.

For 64-bit numbers, only random walk excursion test with AD and KS goodness-of-fit tests cannot reject the null hypothesis for all RNG's. None of the random walk tests decides randomness of RNG's for 128-bit numbers. RNG's pass TBT, Achi, and GCD for k with AD, KS, and CS goodness-of-fit tests for almost all sequence lengths. This situation decreases the proportion of success for 64 and 128-bit numbers. This result would be due to the conservativeness of random walk height, random walk expansion tests, and GCD test with Jarque-Bera goodness-of-fit test for higher bit lengths.

Summary

Statistical analysis of randomness of a cryptographic random number generator is a critical and necessary task to make use of the generator in cryptographic applications. Many cryptographic randomness tests are available for this task including recently proposed ones. Although there are several alternatives, chi-square test is frequently employed within these cryptographic randomness tests as a goodness-of-fit test. In this regard, this article describes the package **CryptRndTest** that conducts frequently used and newly proposed 8 cryptographic randomness tests along with Anderson-Darling, Kolmogorov-Smirnov, chi-square, and Jarque-Bera goodness-of-fit tests. Totally, **CryptRndTest** runs 21 tests. It also provides auxiliary functions for the calculation of greatest common divisor, sequence of partial quotients resulting from the greatest common divisor operation, the base conversion from 2 to 10 and vice versa, and the Stirling numbers of the second kind. All of these auxiliary functions also work with long integers by the use of multi-precision floating point numbers.

The limitations of the package are mostly related to the memory and CPU capacities of the computer used to run functions of **CryptRndTest**. Because, increasing bit-length considerably decreases the implementation speed of tests working over integers, this can also be seen as a limitation for high bit-lengths.

Acknowledgements

This work is fully supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. 114F249 of ARDEB-3001 programme.

Bibliography

P. Alcover, A. Guillamon, and M. Ruiz. A new randomness test for bit sequences. *Informatica*, 24: 339–356, 2013. [p2, 7, 8]

W. Bleick and P. Wang. Asymptotics of stirling numbers of the second kind. *Proceedings of the American Mathematical Society*, 42:575–580, 1974. [p8]

R. Brown, D. Eddelbuettel, and D. Bauer. Dieharder: A random number test suite (version 3.31.1). URL: http://www.phy.duke.edu/ rgb/General/dieharder.php, 2014. [Online; accessed 25-February-2014]. [p1]

H. Demirhan and N. Bitirim. Statistical testing of cryptographic randomness. *Journal of Statisticians: Statistics and Actuarial Sciences*, Submitted, 2015a. [p1]

H. Demirhan and N. Bitirim. Cryptrndtest: An r package for testing the cryptographic randomness. *The R Journal*, Submitted, 2015b. [p2]

A. Doganaksoy, C. Calik, F. Sulak, and M. Turan. New randomness tests using random walk. In *Proceedings of National Cryptology Symposium II*, Turkey, 2006. [p2, 6, 7]

S. Doroshenko and B. Ryabko. The experimental distinguishing attack on rc4. Cryptology ePrint Archive, Report 2006/070, 2006. http://eprint.iacr.org/. [p5]

S. Doroshenko, A. Fionov, A. Lubkin, V. Monarev, and B. Ryabko. On zk-crypt, book stack, and statistical tests. Cryptology ePrint Archive, Report 2006/196, 2006. http://eprint.iacr.org/. [p5]

D. Eddelbuettel and R. Brown. Rdieharder: An r interface to the dieharder suite of random number generator tests. http://cran.r-project.org/web/packages/RDieHarder/vignettes/RDieHarder.pdf, 2014. [Online; accessed 16-June-2015]. [p2]

J. Hernandez, J. Sierra, and A. Seznec. The sac test: a new randomness test, with some applications to prng analysis. *In:Proceedings of the International Conference Computational Science and Its Applications*, pages 960–967, 2004. [p2]

D. Knuth. *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 1 edition, 1969. [p1]

D. Knuth. *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 2 edition, 1981. [p1]

D. Knuth. *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 3 edition, 1998. [p1]

P. L'Ecuyer and P. Hellekalek. Random number generators: Selection criteria and testing. *Random and Quasi-Random Point Sets Lecture Notes in Statistics*, 138:223–265, 1998. [p1]

P. L'Ecuyer and R. Simard. Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33:Article 22, 2007. [p1, 2]

P. L'Ecuyer and R. Simard. Testu01: A c library for empirical testing of random number generators - user's guide, compact version. http://www.iro.umontreal.ca/~simardr/testu01/guideshorttestu01.pdf, 2014. [Online; accessed 24-February-2014]. [p2]

M. Maechler. Arbitrarily accurate computation with r: Package ?rmpfr?, 2011a. [Online; accessed 18-December-2015]. [p10]

M. Maechler. Rmpfr: R mpfr - multiple precision floating-point reliable. r package version 0.4-2. https://cran.r-project.org/web/packages/Rcpp/Rcpp.pdf, 2011b. [Online; accessed 18-December-2015]. [p10]

G. Marsaglia. Diehard: A battery of tests of randomness. http://stat.fsu.edu/~geo/diehard.html, 1996. [Online; accessed 25-February-2014]. [p1]

G. Marsaglia and W. Tsang. Some difficult to pass tests of randomness. *Journal of Statistical Software*, 7: 3, 2002. [p1, 4, 5, 6, 9, 10]

M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000. [p2]

U. Maurer. A universal statistical test for random bit generators. *Journal of Cryptology*, 5:89–105, 1992. [p2]

B. McCullough. A review of testu01. *Journal of Applied Econometrics*, 21:677–682, 2006. [p2]

M. Ruetti. *A Random Number Generator Test Suite for the C++ Standard-Diploma Thesis*. Institute for Theoretical Physics, ETH Zurich, 2004. [p2]

A. Rukhin. Testing randomness: A suite of statistical procedures. *Theory of Probability and Its Applications*, 45:111–132, 2001. [p2]

A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf`, 2010. [Online; accessed 17-June-2015]. [p2]

B. Ryabko and V. Monarev. Using information theory approach to randomness testing. *Journal of Statistical Planning and Inference*, 133:95–110, 2005. [p2, 5, 9]

B. Ryabko, V. Stognienko, and Y. Shokin. A new test for randomness and its application to some cryptographic problems. *Journal of Statistical Planning and Inference*, 123:365–376, 2004. [p2, 3]

J. Sadique, U. Zaman, and R. Ghosh. Review on fifteen statistical tests proposed by nist. *Journal of Theoretical Physics and Cryptography*, 1:18–31, November 2012. [p2]

J. Soto. Statistical testing of random number generators. In *Proceedings of the 22nd National Information Systems Security Conference*, USA, 1999. National Institute of Standards and Technology. [p2]

M. Sýs and Z. Říha. Faster randomness testing with the nist statistical test suite. In C. R. et al., editor, *Security, Privacy, and Applied Cryptography Engineering Lecture Notes in Computer Science*, pages 272–284, Portugal, 2014. Springer. [p2]

M. Sýs, P. Švenda, M. Ukrop, and V. Matyáš. Constructing empirical tests of randomness. In A. H. Mohammad S. Obaidat and P. Samarati, editors, *SECRYPT 2014 Proceedings of the 11th International Conference on Security and Cryptography*, pages 229–237, Portugal, 2014. SCITEPRESS-Science and Technology Publications. ISBN 978-989-758-045-1. [p2]

N. Temme. Asymptotic estimates of stirling numbers. *Studies in Applied Mathematics*, 89:233–243, 1993. [p8]

I. Vattulainen, T. Ala-Nissila, and K. Kankaala. Physical models as tests of randomness. *Physical Review Engineering*, 52:3205–3213, 1995. [p2]

J. Walker. Ent - a pseudorandom number sequence test program. `https://www.fourmilab.ch/random/`, 2014. [Online; accessed 25-February-2014]. [p2]