MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Improvements of the Randomness Testing Toolkit

Bachelor's Thesis

TOMÁŠ MAREK

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Improvements of the Randomness Testing Toolkit

Bachelor's Thesis

TOMÁŠ MAREK

Advisor: Ing. Milan Brož, Ph.D.

Department of Computer Systems and Communications

Brno, Fall 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Marek

Advisor: Ing. Milan Brož, Ph.D.

Acknowledgements

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can
span multiple paragraphs.

Keywords

keyword1, keyword2, ...

Contents

Introduction

To be done, now it is only a collection of ideas.

The desired properties of random sequence are *uniformity* (for each bit the probability for both zero and one are exactly $1/2$), *independence* (none of the bits is influenced by any other bit) and *unpredictability* (it is impossible to predict next bit by obtaining any number of previous bits). [1, p. 1-1]

1 Randomness testing

Goal of this chapter is to provide overview of randomness testing process and to explain all used terms. Explanations of both one and two level tests are accompanied by example applications.

1.1 Overview / Introduction

During a randomness test a *random sequence* is tested. In this document, a random sequence is a finite sequence of zero and one bits, which was generated by a tested random number generator. [1, p. 1-1]

Randomness test is a form of *empirical statistical test*, where we test our assumption about the tested data - the *null hypothesis* (H_0). During the randomness test it states that the sequence is *random*. Associated with the null-hypothesis is the *alternative hypothesis* (H_1), which states that the sequence is *non-random*. Goal of the test is to search for evidence against the null-hypothesis. [2, p. 2]

The result of the test is either that we *accept* the null hypothesis (the sequence is considered random), or that we *reject* the null-hypothesis (and accept the alternative hypothesis - the sequence is considered non-random). We reject the null hypothesis when the evidence found against the null-hypothesis is strong enough, otherwise we accept it. Based on the true situation of null hypothesis, four situations depicted in Table 1.1 may occur. [3, p. 417]

Table 1.1: Possible outcomes when assessing the result of statistical test.

TRUE SITUATION	TEST CONCLUSION	
	Accept H_0	Reject H_0
H_0 is True	No error	Type I error
H_0 is False	Type II error	No error

1.2 Single-level testing

A single-level test examines the random sequence directly (compare with 1.3). Before the test user must choose a *significance level*, which determines how strong the found evidence has to be to reject the null-hypothesis. The test yields a *p-value*, which is used to make the accept or reject the null-hypothesis.

The *significance level* (α) is crucial to assessing the test result and must be set before the test. The α is equal to probability of Type I Error. Usual values are $\alpha = 0.05$ or $\alpha = 0.01$ [3, p. 390], for use in testing of cryptographic random number generators lower values may be chosen. [1, p. 1-4] The lower α is set, the stronger the found evidence has to be to reject the null hypothesis.

The randomness test is defined by a *test statistic* Y, which is a function of a finite bit sequence. Distribution of its values under the null hypothesis must be known (or at least approximated). The value of the test statistic (y) is computed for the tested random sequence. Each test statistic searches for presence or absence of some "pattern" in the sequence, which would show the non-randomness of the sequence. There is infinite number of possible test statistics. [4, p. 4]

The *p-value* is the probability of the test statistic Y taking value at least as extreme as the observed y, assuming that the null hypothesis is true. In randomness testing it is equal to the probability that *perfect random number generator* would generate less random sequence. The smaller is the p-value, the stronger is the found evidence against the null-hypothesis. [3, p. 386] The p-value is calculated based on the observed y.

1.2.1 Result interpretation

Decision about the test result is based on the computed *p-value*. If the p-value is lower than the α, we *reject the null hypothesis* (and accept the alternative hypothesis), because strong enough evidence against null hypothesis was found. If the p-value is greater than or equal to the α, we *accept the null hypothesis*, because the evidence against the randomness was too weak. [3, p. 390] It is sometimes recommended to report the *p-value* as well instead of accept/reject only, as it yields more information. [2, p. 90]

The p-values close to α can be considered *suspicious*, because they do not clearly indicate rejection. Further testing of the random number generator on *other* random sequences is then in place to search for further evidence. [4, p. 5] The reason is that *randomness* is a probabilistic property, therefore even the perfect random number generator may generate a nonrandom sequence with low p-value (although it is very unlikely). The further evidence is used to differentiate between the bad generator generating a non-random sequence systematicaly and the good generator generating non-random sequence 'by chance'. [2, p. 90]

1.2.2 Example

To demonstrate how a single randomness test is made, the Frequency (Monobit) Test from NIST STS battery was chosen. [1, p. 2-2] This test is based on testing the fraction of zeroes and ones within the sequence. For a random sequence with length n the count of ones (and zeroes) is expected to be around $n/2$ (the most probable values are close to $n/2$).

For the Monobit test it is recommended that the tested sequence has at least 100 bits. The test statistic S_{obs} of the Monobit test is defined as

$$S_{obs} = \frac{|\#_1 - \#_0|}{\sqrt{n}}$$

where $\#_1$ is count of ones in the tested sequence (similarly for zeroes) and n is length of of the tested sequence. Under the null hypothesis, the reference distribution of S_{obs} is half normal (for large n). The p-vaue is computed as

$$p = erfc(\frac{S_{obs}}{\sqrt{2}})$$

where $erfc$ is the *complementary error function* used to calculate probabilities in normal distribution.

Let

$$\epsilon = 1001100101001000001000100101100110110000110100011110101001010010010011100111001100110010010100111011$$

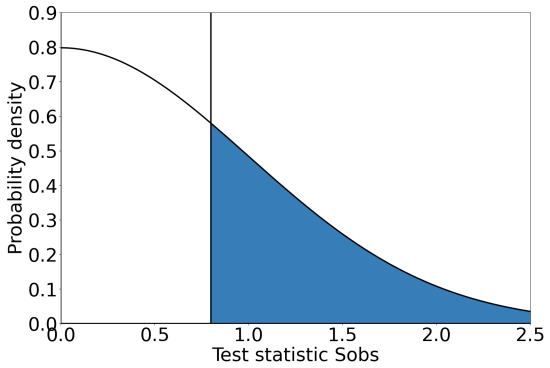


Figure 1.1: Visualization of example p-value for test statistic value $y = 0.8$

be the tested sequence. The test statistic for this sequence is

$$S_{obs} = \frac{|46 - 54|}{\sqrt{100}} = \frac{|-8|}{10} = 0.8$$

and the p-value (visualised at Figure 1.1) is

$$p = erfc\left(\frac{0.8}{\sqrt{2}}\right) \approx 0.423$$

To interpret the test, we compare the computed *p-value* to the chosen α. The *p-value* ≈ 0.423 is greater than both usual $\alpha = 0.05$ and $\alpha = 0.01$, therefore we accept the null hypothesis for both *significance levels* and the sequence ϵ is considered random.

1.3 Two-level testing

The *two-level test* is done by repeating the single-level test n times. The important part is comparing the distribution of produced p-values to the expected distribution. The two-level test allows the random sequence to be examined both locally and globally, while the single-level test examines the sequence only on the global level. This may lead to discovering local patterns, which cancel out on the global level. [4, p. 7]

To apply the two-level test the tested sequence is split into n equal-length disjoint subsequences. The same single-level test is applied to each of the subsequences (as described in Section 1.2) and its *p-values* are collected, resulting in set of n *p-values*[1]. The tests are called *first-level tests* and the p-values are called *first-level p-values*. Under the null-hypothesis, the first-level p-values of a given test statistic are uniformly distributed over the interval $(0, 1]$. [5, p. 14]

The crucial part of two-level test is examining the distribution of *observed first-level p-values*. Usually, the *goodness-of-fit* (GOF) tests are applied as the *second-level test*. [4, p. 6] GOF tests are a family of methods used for examining how well a data sample fits given distribution. [6, p. 1] The most used GOF tests in randomness testing are the χ^2 (chi-squared) and Kolmogorov-Smirnov test, another notable tests are the Anderson-Darling and Cramér-von-Mises test. [5, p. 14]

The second-level test is defined by a test statistic Y, which is a function of the first-level p-values. Test statistic value (y) is calculated from the observed *first-level p-values* and then the *second-level p-value* is calculated from y. At last, the second-level p-value is interpreted as in one-level test (as described in Subsection 1.2.1).

Alternatively, a *proportion of subsequences passing the first-level test* is used to examine the fist-level p-values uniformity. Under the null-hypothesis, it is expected for $n \cdot \alpha$ subsequences to *be rejected* (i.e. to have p-value $< \alpha$) by the first-level test (be a subject to Type I Error). The ratio of sequences passing the first-level test is expected to be around $1 - \alpha$, different ratio indicates non-uniformity of observed first-level p-values. [1, p. 4-2] No p-value is reported in this case, only the ratio.

1.3.1 Kolmogorov-Smirnov test

The one-sample Kolmogorov-Smirnov (KS) test is used in randomness testing to compare the observed first-level p-values to the uniform distribution. The Kolmogorov-Smirnov test is built on comparing the cumulative distribution function (CDF)[2] of the expected distribution

1. Note that the p-values are not subject to accept/reject decision.
2. For a given distribution and value x, the CDF returns the probability of drawing a value less than or equal to x.

and the empirical cumulative distribution function (eCDF)[3] of the observed samples.

In first variant, two test statistics are calculated. The test statistic D^+ (D^-) is the maximal vertical distance between CDF and eCDF above (under) the CDF. In second variant, only the test statistic D (maximal vertical distance between CDF and eCDF) is measured. Formally, the test statistics are defined as

$$\begin{aligned} D^+ &= sup_x\{F_n(x) - F(x)\} \\ D^- &= sup_x\{F(x) - F_n(x)\} \\ D &= sup_x\{|F_n(x) - F(x)|\} = max(D^+, D^-) \end{aligned}$$

where $F(x)$ is the CDF and $F_n(x)$ is the eCDF [6, p. 100].

1.3.2 Chi-squared test

The Pearson's χ^2 test is used to find statistically significant difference between frequencies of categories in two sets of categorical data. The first-level p-values are split into k equal-width bins (categories) and their respective frequencies are counted. The counted frequencies are compared to the expected frequencies.

For data with k categories the test statistic χ^2 is defined as

$$\chi^2 = \sum_{i=1}^{k} \frac{(x_i - m_i)^2}{m_i}$$

where x_i is the observed frequency in i-th category and m_i is the expected frequency in i-th category. For first-level p-values, the expected frequency is equal in each interval. For a correct test the expected frequency in each category must be at least five. [7, p. 171]

1.3.3 Example

In the two-level test example, I will test one sequence using both one and two-level tests to demonstrate the difference between them. First, the sequence is tested using the one-level Frequency (Monobit) test

3. For a set of observed data and value x, the eCDF returns the probability of drawing a value less than or equal to x.

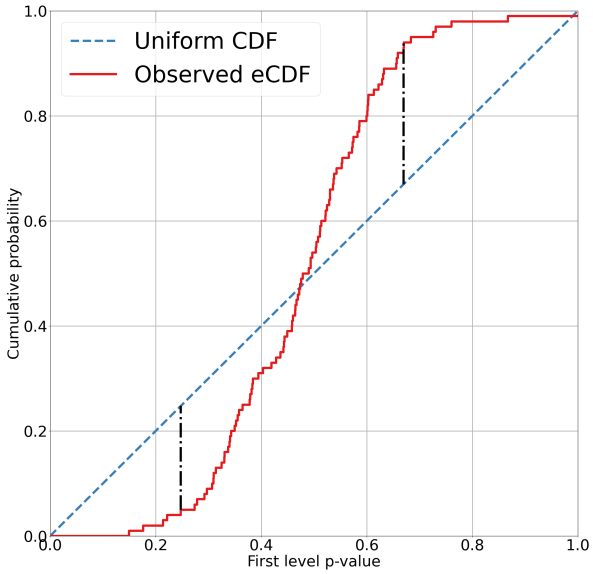


Figure 1.2: Visualization of Kolmogorov-Smirnov test statistics for expected uniform distribution and observed data sample.

Table 1.2: First-level p-values produced by Monobit test

p-value	occurrences
$1.52 \cdot 10^{-23}$	30
0.31	5
1.0	15

from NIST STS battery.[1, p. 2-2] Then the same sequence is assessed by the two-level test using the Frequency test as the first-level test and KS and χ^2 tests as second-level test. Let

$$\begin{aligned}\epsilon = \; & 15 * (100\, \textit{consecutitive zeroes}) + \\ & 15 * (100\, \textit{alternating ones and zeroes}) + \\ & 5 * (55\, \textit{zeroes and}\, 45\, \textit{ones}) + \\ & 15 * (100\, \textit{consecutive ones})\end{aligned}$$

be the tested sequence.

Result of the one-level Frequency test for the sequence ϵ is p-value ≈ 0.479. The null hypothesis is accepted for both $\alpha = 0.01$ and $\alpha = 0.05$ and the sequence ϵ is considered random. This sequence however clearly contains a pattern, therefore the probability of it being generated by a perfect random number generator is very low.

For the two-level test, the sequence ϵ is split into $n = 50$ disjoint 100 bit long subseqeunces. The Monobit test is applied on each subsequence resulting in set of first-level p-values shown in Table 1.2.

Last step is to apply the goodness-of-fit tests. The first applied test is the Pearson's χ^2 test with $k = 10$ (number of categories), the expected frequency of p-values in each category is five. The statistic of the test is

$$\chi^2 = \sum_{i=1}^{10} \frac{(x_i - 5)^2}{5} = 180$$

and the p-value of this test is $p \approx 5.06 \cdot 10^{-34}$. The null hypothesis is rejected for both $\alpha = 0.01$ and $\alpha = 0.05$.

Next, the Kolmogorov-Smirnov test is applied. The eCDF is calculated and then the D statistic is computed. The statistic is $D = 0.6$ and results in p-value $\approx 9.63 \cdot 10^{-18}$. Again, the null-hypothesis is re-

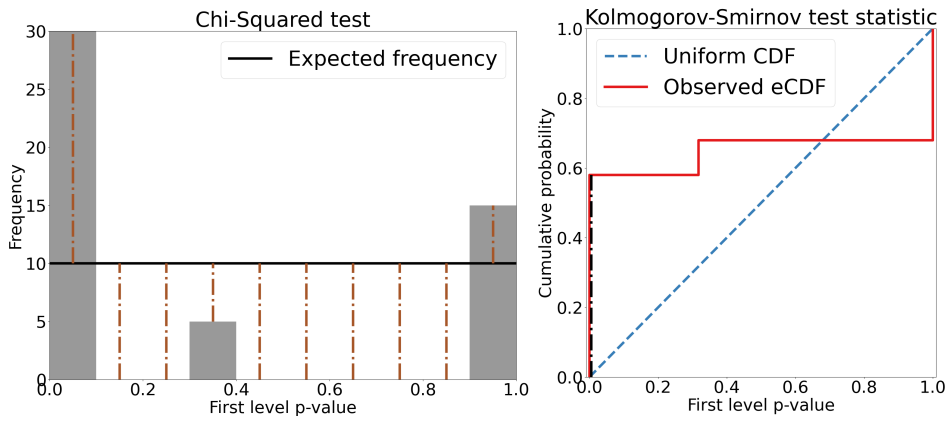


Figure 1.3: Visualisations for χ^2 and KS tests for two-level test example.

jected for both $\alpha = 0.01$ and $\alpha = 0.05$ and the sequence is considered non-random.

2 Available solutions

In this chapter, available solutions for randomness testing are described. In the first section, *statistical testing batteries* (later referred to only as battery) are described. In the second section, *randomness testing toolkits*, which encompass several batteries together, are described. For both batteries and testing toolkits, overview of setup and output is given.

2.1 Statistical testing batteries

Statistical testing battery (*suite*) is a set of randomness tests with predefined parameters that allows the user to conveniently apply randomness tests. [2, p. 5] Because all of the batteries share significant similarities, an *abstract battery* is presented to demonstrate the principles. Then, each battery is presented in more detail, mapping its features to the *abstract battery*.

The *abstract battery* consist of n individual tests, which are distinguished by *test IDs*. Usually one individual test maps to one two-level test. Some individual tests consist of more *subtests*.

Each *subtest* maps to one two-level test When executing an individual test, all its subtests are executed. The test statistics of subtests are usually related to each other by using the same operation on the tested sequence.[1]

Variant of the test is individual test that is parametrized to search for modified pattern in the data. Only some of the individual tests allow such parametrization. Usually, all possible *variants* of given individual test are executed. [9, p. 2]

Usually, the following settings are required for each *individual test*. They can be set either *globally* with same values for all tests, or *individually*:

- *number of first-level tests* - Sets how many times the first-level test is repeated.

1. For example, the Diehard Craps Test plays 200 000 games of craps. The two calculated first-level test statistics are based on number of wins and GOF test of distribution of throws needed to end the game. [8]

- *size of first-level subsequence* - Sets how long are the subsequences used for first-level test.
- *choose variant* (if applicable for given test).

Important part of files testing is *file rewind* detection. File rewind occurs when the tested file is not large enough for given test configuration. In this case, some parts of the file are read twice and the results may be biased. The needed data size is calculated as *number of first-level tests · size of first-level sequence*.

2.1.1 Dieharder

The *Dieharder* battery was developed as an extension to an older *Diehard* battery. The *Dieharder* is available from [8] or as package in some Unix distributions. Both *RTT* and *rtt-py* use a version of *Dieharder* with modified output format available from [10].

31 individual tests are contained in the Dieharder. However, four of them are marked as 'Suspect' or 'Do Not Use' and should not be used. Test variants are possible in four tests.

The *Dieharder* allows all settings as described in the *abstract battery*. They are:

- *psamples* argument sets the number of first-level tests. Default value is 100.
- *tsamples* arguments sets the length of first-level subsequences. Units are *random entities*, which are different size for each test. The tests have various default values, some tests ignore this argument.
- *ntup* argument chooses *test variant* for relevant individual tests.

Results of the *Dieharder* are printed to standard output in a table of results. Each row of the table contains results of one individual test or subtest. Columns contain *test name*, value of *ntup* argument (if applicable, usually 0 otherwise), *tsample* and *psamples* values. Last two columns are the *second-level p-value* and *assessment*. Dieharder output can be modified using *output flags*, including the possibility to print all first-level p-values. Example of Dieharder output can be found in Appendix A.

The *file rewinds* are detected by Dieharder automatically. In such case message '`# The file file_input_raw was rewound <n> times`' is printed to error output. Alternatively, by using dedicated output

flag, the amount of data used and this message are printed to standard output after each individual test.

2.1.2 NIST STS

Original NIST STS[2] implementation is available from [11]. Optimized version developed by CRoCS FI MU [1] is used in RTT, available from [10] *NIST STS* contains 15 individual tests with no variants. Allowed settings are:

- *Stream count* argument denotes the number of first-level tests and must be set by the user. For a statistically meaningful result of second-level test should be at least 55.
- *Stream size* argument sets the length of first-level subsequences in *bits* and must be set by the user. The tests have various recommendations for minimal size.

The second-level p-value is computed using the χ^2 test after dividing the first-level p-values into $k = 10$ categories. The NIST STS also calculates the proportion of subsequences passing the first-level test. [1, p. 4-1].

Tests results are stored in files inside *experiments/AlgorithmTesting* folder. Table with overview of results is in file *finalAnalysisReport.txt*. Each row contains one individual test or subtest. First 10 columns are observed frequencies of first-level p-values in given category. The *second-level p-value, proportion of subsequences passing the first-level test* and *test name* follow.

More results information for each test are stored in corresponding folders. In file *stats.txt* the first-level p-values, test statistics and other computational information are stored. In file *results.txt*, only the first level p-values are stored. Example of output is available in Appendix B.

The file rewind is detected automatically by the battery. In such case, message `'READ ERROR: Insufficient data in file.'` is printed to standard output.

2. National Institute of Standards and Technology - Statistical Test Suite

2.1.3 Test U01

TestU01 was developed to be a state-of-the-art software library oriented on testing of random number generators and is available from [12]. It contains several batteries, used in *RTT* and *rtt-py* are the *SmallCrush, Crush, BigCrush, Rabbit, Alphabit* and *BlockAlphabit*. Because TestU01 is a software library only, custom command-line interface was created for use in RTT (also used in *rtt-py*), available from [10].

Each battery in TestU01 allows for a different set of user settings. All possible arguments in TestU01 are:

- *repetitions* argument sets the number of first-level tests
- *bit_nb* argument sets the length of first-level subsequences in *bits*. Applicable (and mandatory) only for Rabbit, Alphabit and BlockAlphabit.
- *bit_w* argument is used to choose test variant. Only in *BlockAlphabit* battery.

Unlike other batteries, the TestU01 batteries do not employ the *second-level test*. The assessment of first-level p-values uniformity is up to the user.

rewrite output better

The output format is same of for all of TestU01 batteries and is printed to standard output. For *individual test* following is printed. For each executed first-level test the test name and all test parameters are printed. Then, for each subtest or individual test, test static value and its corresponding p-value is printed. After each first-level test, a *generator state* is printed, containing information about how many data were used in all first-level tests so far. And the end of the individual test report, list of all first-level p-values is printed. Example output is in Appendix C.

File rewind detection is done partially by the batteries using the *generator state* information, however it is up to the user to interpret it. The *generator state* contains three fiels - *bytes need for testing* (number of bytes that were indeed used for testing), *bytes read from file* (number of bytes that were read from tested file) and *total number of file rewinds*.

Because the data are read from file in 10MB long blocks, the number of file rewinds may be positive, but the number of *bytes needed for testing*

will be lower than the actual file size, causing a false alarm. It is up to the user to manually check this situation.

SmallCrush, Crush, BigCrush

Batteries from the *Crush* family were created to test general use random number generators. The batteries contain 10, 96 and 106 tests with increasing demand for data size and runtime. The intended use is to apply SmallCrush for a quick assessment. If the the sequence is accepted, more stringent Crush and BigCrush batteries are applied. The size of first-level sequence cannot be changed. [2, p. 242]

Rabbit

The Rabbit battery contains 26 individual tests. The *bit_nb* argument must be set by the user to a value of at least 500. *At most bit_nb* will be used for first-level subsequence size. [2, p. 152] However, several tests require significantly longer subseqeunce than 500 bits. Some tests use significantly shorter subsequence than specified by *bit_nb*.

Alphabit and BlockAlphabit

Both Alphabit and BlockAlphabit batteries contain the same nine individual test. In the BlockAlphabit battery, the tested data are reordered to deploy *test variants*. The test variant is chosen by the *bit_w* argument taking value from set {1, 2, 4, 8, 16, 32}. [2, p. 155] Size of the first-level sequences will be *at most* the size set by *bit_nb* argument.

2.1.4 FIPS Battery

The FIPS[3] battery contains five tests and is based on FIPS 140-2 standard. Custom command-line interface of the battery was created for *rtt-py* available from [13]. The interface is based on implementation taken from [14]. No test variants are available and the length of first-level sequence is set to 2500 bytes and cannot be changed [9, p. 20]. Only one arguments is available:

3. Federal Information Processing Standards

- *bytes count* argument sets how many bytes will be used for testing *altogether*, determining the *number of first-level tests*

Output of FIPS battery is printed to standard output and in user-specified file in JSON[4] format. First, information about accepting or rejecting the null-hypothesis is printed. Then a list of individual tests results is printed. For each individual test, the test name, number of failures and number of runs is printed. Example output is in Appendix D

File rewinds are detected automatically by the battery. In such case, the battery will not run and will print message '`Error (<filename>) ! File is not big enough`' to error output.

2.1.5 BSI battery

The BSI[5] battery contains nine test and is based on series of standards released by BSI. [9, p. 3] For use in *rtt-py* a custom command-line interface was created, available from [13]. The tests implementations were extracted from the ParanoYa application. [9, p. 16]

No test variants are available in the BSI battery. Each test has its own preset *first-level subsequence size*, which cannot be changed. One argument is available for the battery:

- *bytes count* argument sets how many bytes will be read from the tested file. The number of *first-level tests* is calculated based on this value.

Output of BSI battery is printed to standard output and in user-specified file in JSON format. It contains list of individual tests results. For each individual test, a name and information whether *total error* occurred is printed. If no *total error* occurred, number of failures and number of runs is printed as well. Example output is available in Appendix E.

File rewinds are detected automatically by the battery. In such case, the battery will not run and will print message '`File is not big enough`' to error output.

4. JavaScript Object Notation
5. Bundesamt für Sicherheit in der Informationstechnik

2.2 Testing toolkits

In the previous section different randomness testing batteries were described. The typical user, however, uses more than one battery, which means installing and running each testing battery individually. Also it is strongly recommended (sometimes even needed) to set up parameters for each test from the battery individually based on tested file and to run this test manually.

Since this approach is not convenient, Center for Research on Cryptography and Security (CRoCS) at FI MU created the Randomness Testing Toolkit (*RTT*). This toolkit allows users to run and configure eight test batteries using the same command.

This work was followed by Patrik Vaverčák from Faculty of Electrical Engineering and Information Technology at Slovak University of Technology. He created newer variant of *RTT* called Randomness Testing Toolkit in Python (*rtt-py*).

2.2.1 Randomness Testing Toolkit

RTT was created in 2017 and its main idea was to combine *Dieharder*, *NIST STS* and all batteries from *TestU01* mentioned in Subection 2.1.3 into one program. It was written in C++. The concept is that *RTT* acts only as a unified interface of the batteries. Each test battery is executed by *RTT* as a separate program. The *RTT* then collects the output and processes it into a unified format. [15, p. 8]

RTT is available from CRoCS GitHub repository [16]. All of the batteries used in RTT are available from a single GitHub repository [10]. Before running, user has to install both the *RTT* and used batteries as described GitHub project wiki. If the user intends to run NIST STS, the *experiments* folder has to be moved (or linked) to *working directory* of RTT. This is not noted anywhere.

Settings

The *RTT* needs to be set up by the user before running. The first part of user settings contains *general settings* of the *RTT*, the second part contains individual *batteries configurations*. Each of these parts is stored in its own JSON file.

The *general settings* are stored in *rtt-settings.json* file, which is located in the working directory of the *RTT* [15, p. 10]. These settings are not expected to change. The most important setting from the general part are paths to the executable binaries of individual statistical test batteries. This is the only setting that has to be manually filled in by the user.

The storage database can also be filled in by the user, but this functionality is often ignored. The following general settings have implicit values and do no need to be changed unless the user wishes to. They are paths to storage directories for results and logs of individual runs and execution options (test timeout and maximal number of parallel executions of tests).

The battery configurations are dependant on the size of the tested file, therefore the file with the battery configuration is specified for each run of the *RTT* as one of its arguments. These configurations are different for each battery (see Section 2.1), but settings for all of the batteries are stored together in a single file. [15, p. 11] The *RTT* contains several prepared battery configurations for various sizes of tested file.

Output

The output of *RTT* is in a plain text format. The most important part of the output is the direct report, which is saved in the *results* directory. At the beginning of the report are general information – the name of the tested file, the name of the used battery, ratio of passed and failed individual tests and battery errors and warnings in case there were any.

After the general information is a list of results of individual tests in a unified format. The fist part of the individual test report contains the name of the test and user settings. The second part of the single test report are the second-level p-values alongside the names of statistic used (usually Kolmogorov-Smirnov or χ^2 statistic). At the end of the single test report is a list of first-level p-values produced by the test. Example of the output can be seen in Figure 2.1.

```
------------------------------------------------------------
Diehard Birthdays Test test results:
    Result: Passed
    Test partial alpha: 0.01

    User settings:
        P-sample count: 65
    ************

    Kolmogorov-Smirnov statistic p-value: 0.46269520       Passed
    p-values:
        0.01554128 0.01704044 0.07338199 0.08890865 0.13047059
        0.14641850 0.14648858 0.14985241 0.15741014 0.17234854
        0.17570707 0.18313806 0.19708195 0.21929163 0.23582928
        0.23875056 0.24659048 0.24810255 0.26921690 0.29350665
        0.29444024 0.29618689 0.30017915 0.30767530 0.32816499
        0.33671597 0.33723518 0.33723518 0.35046577 0.36986762
        0.38616538 0.40739822 0.42316216 0.42606175 0.42712489
        0.46376818 0.47710967 0.51301110 0.55638736 0.58615965
        0.58816320 0.62212002 0.63106447 0.65794861 0.66078115
        0.66209483 0.67060673 0.69336319 0.70343506 0.72259414
        0.74451995 0.79749441 0.81986290 0.85442793 0.88851953
        0.88897431 0.89604503 0.92240447 0.93852901 0.93852901
        0.95468456 0.96540827 0.96785289 0.96922576 0.99790555
    ============
------------------------------------------------------------
```

Figure 2.1: The example of single test report from the *RTT*

2.2.2 Randomness Testing Toolking in Python

The Randomness Testing Toolkit in Python (*rtt-py*) was created by Patrik Vaverčák. It is supposed to be an improved version of *RTT* sharing the same concept [9, p. 24] and it was written in Python.

The included batteries are Dieharder, NIST STS, FIPS battery and BSI battery. *Rtt-py* also includes Rabbit, Alphabit and BlockAlphabit batteries from TestU01. The Crush family batteries are not run, even though arguments of *rtt-py* suggest that they are included. The *rtt-py* allows for multiple files to be tested at once.

The *rtt-py* is available from [17] and implementations of all used batteries are available from [13]. Before running, both the *rtt-py* and the batteries have to be installed as described in the project's README.

Settings

The settings of *rtt-py* use similar format as the original *RTT* (as described in Subsection 2.2.1). The *general settings* from the *RTT* should be one-way compatible with *rtt-py* [9, p. 25]. In reality there is a problem with settings for the NIST STS's experiments directory. Also, no database connection is implemented in *rtt-py*, therefore the *mysql-db* attribute is ignored. [17]

The second part of user settings are the battery configurations. They use the same format as in *RTT* (as mentioned in 2.2.1) and are interchangeable. [9, p. 25] The user has to keep in mind that the *rtt-py* uses FIPS and BSI batteries, which are not used in *RTT*.

Output

The *rtt-py* creates output in two formats – *CSV*[6] and *HTML*[7]. [9, p. 36] Both of these report formats contain overview table. Each row from the table represents results of one individual test or subtest. The first column contains the name of the test and the name of the battery it belongs to. The second column contains *failure rate* - ratio of sequences not passing the first-level test.

6. Comma-separated values
7. Hypertext Markup Language

Results overview

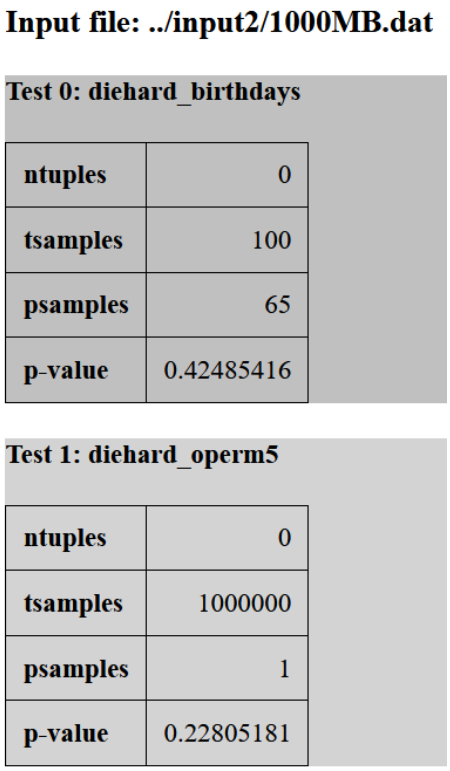
	Failure rate	../input2/1000MB.dat	../input2/1000MB_2.dat
rgb_minimum_distance_0 (DIEHARDER)	0.0	0.754103	0.407390
rgb_permutations_0 (DIEHARDER)	0.0	0.931074	0.184047
diehard_operm5_0 (DIEHARDER)	0.0	0.228052	0.680182
sts_monobit_0 (DIEHARDER)	0.0	0.279259	0.096535
sts_serial_0 (DIEHARDER)	0.0	0.279259	0.096535
sts_serial_1 (DIEHARDER)	0.0	0.972893	0.052121
sts_serial_2 (DIEHARDER)	0.0	0.621721	0.618407

Figure 2.2: The example of the overview table from the *rtt-py*

Each of the following columns is named after one tested file. The record contains either second-level p-value reported by the test, or number of failed runs – this depends on the battery. Example of this table can be seen at figure 2.2.

The output in the HTML format contains than the output in the CSV format. For each battery and for each tested file an HTML file with reports is generated.

In each report file there is a list of reports for each individual test or subtest from the given battery containing the result (either reported p-value, or number of failed runs). It may contain additional information such as settings of the test or other information connected to the result, depending on the battery and on the executed test. Example of the report can be seen in figure 2.3



Input file: ../input2/1000MB.dat

Test 0: diehard_birthdays

ntuples	0
tsamples	100
psamples	65
p-value	0.42485416

Test 1: diehard_operm5

ntuples	0
tsamples	1000000
psamples	1
p-value	0.22805181

Figure 2.3: The example of HTML Dieharder report from the *rtt-py*

3 Tests Analysis

We can choose from various test statistics. Most of the test statistics in widely used test batteries work with data of fixed length. TODO: REF ANALYSIS CHAPTER However, in some tests data with varying length are tested. These statistics further split into two categories. In the first category, the length of tested data is preset by user. These can be further viewed as fixed-length tests. In the second category, the length of tested data is determined during the testing process.

3.1 Data Consumption

several big tables, mention exact parameters the tests were run with

3.2 Time Consumption

again some big tables, choose one test as a reference and the rest will be relative. mention exact parameters, maybe add throughput?

3.3 Configuration Calculator

goal of the config calc, description, usage etc...

3.4 P-Values

Various problems with test p-values distributions, will probably be split into more sections

4 Implementations Comparison

4.1 Output

Mentioned differences
for both RTT and rtt-py - subset or whole?

4.2 Missing Features of *rtt-py*

4.3 Proposed improvements

included things: adding first-level p-values,

5 Conclusion

A Dieharder output

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
 file_input_raw|              ../input/5GB_2.dat|  1.98e+07  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.83818462|  PASSED
      diehard_operm5|   0|   1000000|     100|0.91218248|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.74062573|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.72202153|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.54621918|  PASSED
        diehard_opso|   0|   2097152|     100|0.96877577|  PASSED
        diehard_oqso|   0|   2097152|     100|0.64303740|  PASSED
         diehard_dna|   0|   2097152|     100|0.64407925|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.37839401|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.09580878|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.33358085|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.93274571|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.08383126|  PASSED
     diehard_squeeze|   0|    100000|     100|0.97500761|  PASSED
        diehard_sums|   0|       100|     100|0.62861437|  PASSED
        diehard_runs|   0|    100000|     100|0.45829724|  PASSED
        diehard_runs|   0|    100000|     100|0.02341244|  PASSED
       diehard_craps|   0|    200000|     100|0.78964194|  PASSED
       diehard_craps|   0|    200000|     100|0.90416388|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.93600147|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.79305849|  PASSED
         sts_monobit|   1|    100000|     100|0.68009432|  PASSED
            sts_runs|   2|    100000|     100|0.18224901|  PASSED
          sts_serial|   1|    100000|     100|0.84229425|  PASSED
          sts_serial|   2|    100000|     100|0.75720963|  PASSED
          sts_serial|   3|    100000|     100|0.84291363|  PASSED
          sts_serial|   3|    100000|     100|0.96460124|  PASSED
```

Figure A.1: Example of results table from the *Dieharder* battery.

```
#=============================================================================#
#                            Values of test p-values                          #
#=============================================================================#
|0.04369416|
|0.04827681|
|0.05784669|
|0.06688939|
|0.07899242|
|0.08939748|
|0.09554753|
|0.10181189|
|0.11226737|
|0.11663826|
|0.12257360|
|0.12304411|
|0.12414260|
|0.13178605|
|0.13699214|
.
.
.
|0.95590312|
|0.95849805|
|0.96228421|
|0.97049026|
|0.97974257|
|0.98378624|
|0.98427673|
|0.98622772|
|0.99085826|
#=============================================================================#
```

Figure A.2: Examples of first-level p-values printout from *Dieharder* battery

B NIST STS output

C TestU01 output

D FIPS battery output

E BSI battery output

F RTT settings

G RTT output

H rtt-py out

Bibliography

1. BASSHAM III, Lawrence E; RUKHIN, Andrew L; SOTO, Juan; NECHVATAL, James R; SMID, Miles E; BARKER, Elaine B; LEIGH, Stefan D; LEVENSON, Mark; VANGEL, Mark; BANKS, David L, et al. *SP 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology, 2010. Available also from: `https://csrc.nist.gov/Projects/random-bit-generation/Documentation-and-Software`.
2. L'ECUYER, Pierre; SIMARD, Richard. *TestU01 - A Software Library in ANSI C for Empirical Testing of Random Number Generators - User's guide, compact version*. 2002. Available also from: `http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf`.
3. MOORE, David S.; NOTZ, William I. *The Basic Practice of Statistics*. Macmillan Learning, 2021. ISBN 1-319-38368-8.
4. L'ECUYER, Pierre; SIMARD, Richard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*. 2007, vol. 33, no. 4, pp. 1–40.
5. SÝS, Marek; OBRÁTIL, Lubomír; MATYÁŠ, Vashek; KLINEC, Dušan. A Bad Day to Die Hard: Correcting the Dieharder Battery. *Journal of Cryptology*. 2022, vol. 35, pp. 1–20.
6. D'AGOSTINO, Ralph B.; STEPHENS, Michael A. *Goodness-of-Fit-Techniques*. Routledge, 1986. ISBN 0-8247-8705-6.
7. SHESKIN, David J. Parametric and nonparametric statistical procedures. *Boca Raton: CRC*. 2000.
8. *Robert G. Brown's General Tools Page* [online]. Brown, Robert G. [visited on 2023-11-10]. Available from: `https://webhome.phy.duke.edu/~rgb/General/dieharder.php`.

9. VAVERČÁK, Patrik. *Aplikácia na štatistické testovanie pseudonáhodných postupností*. Bratislava, 2022. Available also from: https://opac.crzp.sk/?fn=detailBiblioFormChildEBUS6&sid=D9F0BB0A8DA9926980A890D31B33&seo=CRZP-detail-kniha. Master's thesis. Slovak University of Technology in Bratislava, Faculty of Electrical Engineering. Supervised by Matúš JÓKAY.
10. *rtt-statistical-batteries* [online]. [visited on 2023-11-10]. Available from: https://github.com/crocs-muni/rtt-statistical-batteries.
11. *Random Bit Generation | CSRC* [online]. [visited on 2023-11-10]. Available from: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software.
12. *Empirical Testing of Random Number Generators* [online]. [visited on 2023-11-12]. Available from: http://simul.iro.umontreal.ca/testu01/tu01.html.
13. VAVERČÁK, Patrik. *rtt-statistical-batteries* [online]. [visited on 2023-11-13]. Available from: https://github.com/pvavercak/rtt-statistical-batteries.
14. MORAES HOLSCHUH, Henrique de. *fips.c · master · Henrique de Moraes Holschuh / rng-tools · GitLab* [online]. [visited on 2023-11-13]. Available from: https://salsa.debian.org/hmh/rng-tools/-/blob/master/fips.c.
15. OBRÁTIL, Ľubomír. *The automated testing of randomness with multiple statistical batteries*. Brno, 2017. Available also from: https://is.muni.cz/th/uepbs/. Master's thesis. Masaryk University, Faculty of Informatics. Supervised by Petr ŠVENDA.
16. *crocs-muni/randomness-testing-toolkit: Randomness testing toolkit automates running and evaluating statistical testing batteries* [online]. [visited on 2023-11-14]. Available from: https://github.com/crocs-muni/randomness-testing-toolkit.
17. *pvavercak/rtt-py: Randomness testing toolkit in python* [online]. [visited on 2023-11-14]. Available from: https://github.com/pvavercak/rtt-py.