MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Improvements to the Randomness Testing Toolkit

Bachelor's Thesis

TOMÁŠ MAREK

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Improvements to 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

Cryptography plays vital role in securing computer communication today. Many cryptographical primitives require to be set up using unique values (e.g. keys or nonces) to provide their functionality correctly. To acquire unique values, random number generators are used in practice.

Random number generators are tools for generating bit sequences with content that appears random. The desired properties [1, p. 1-1] of such sequence are *uniformity* (for each bit, the probability of 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).

The random number generators are of different quality in regard to fulfilling the desired properties of generated sequences. Each application of random number generators has different requirements for quality of the used generator. In cryptography, failure to fulfill the requirements may lead to compromising the security of the communication.

Therefore, there is need to assess the quality of the used random number generators. The assessment is done by mathematical analysis of the generator structure or by taking a sample output of the generator and applying a *randomness test* [2, p. 2].

Currently, there are several available programs for randomness testing. Most notable are the NIST STS[1] [3], TestU01 [4] and Dieharder [5]. Providing a unified interface to the three are the Randomness Testing Toolkit (*RTT*) [6] and its evolution, the Randomness Testing Toolkit in Python (*rtt-py*)[7].

The *RTT* contains flaws in regard to both usability and functionality. It might be replaced by the *rtt-py* in the future, however, the two implementations are different. Therefore, the main goal of this thesis is to analyse the *RTT* and the *rtt-py* with focus on finding differences between the two implementations (particularly the features missing in *rtt-py*) and on proposing improvements for both of the toolkits.

In Chapter 1, we explain the principles of randomness testing along with terms used in the following chapters. The aforementioned pro-

1. National Institute of Standards and Technology Statistical Test Suite

grams for randomness testing are presented in Chapter 2. In the first part of the analysis (described in Chapter 3), we focus on improving the *RTT* by analyzing the individual tests used by the *RTT*.

The second part of the analysis (presented in Chapter 4) is focused directly on *RTT* and *rtt-py*. In this part, we present and compare the output of *RTT* and *rtt-py*. Then, we analyse the features of both toolkits and list all differences found. Last, we propose several improvements that would extend the functionality of both *RTT* and *rtt-py*.

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. [8, 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 $\alpha = 0.01$ [8, 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. [9, 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. [8, 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. [8, 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. [9, 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

$$\epsilon = 1001100101001000001000100101100110110000110100011110101001010010010011100111001100110010010100111011$$

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.

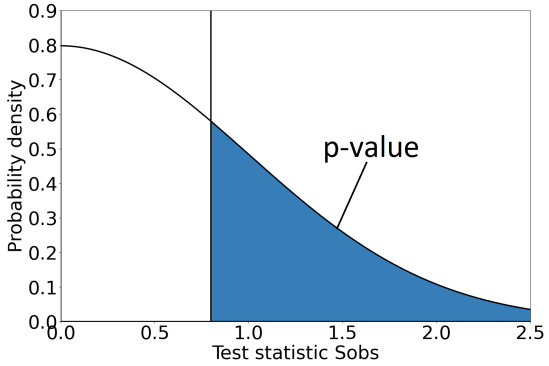

Figure 1.1: Visualization of Monobit test example p-value for test statistic value $y = 0.8$

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. [9, 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 $(0, 1]$. [10, 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*. [9, p. 6] GOF tests are a family of methods used for examining how well a data sample fits given distribution. [11, 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. [10, 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

1. Note that the p-values are not subject to accept/reject decision.

cumulative distribution function (CDF)[2] of the expected distribution 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 [11, 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. [12, 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,

2. For a given distribution and value x, the CDF returns the probability of drawing a value less than or equal to x.
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.

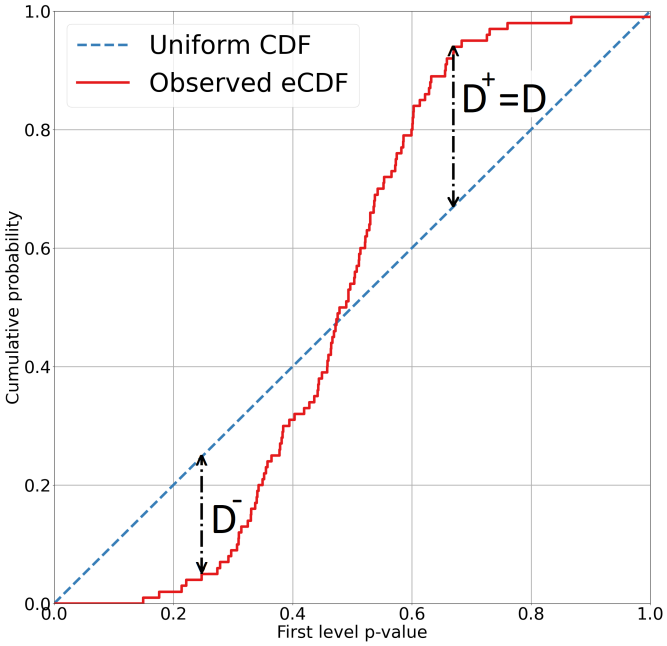

Figure 1.2: Visualization of Kolmogorov-Smirnov test statistics for expected uniform distribution and observed data sample.

the sequence is tested using the one-level Frequency (Monobit) test 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.

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

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

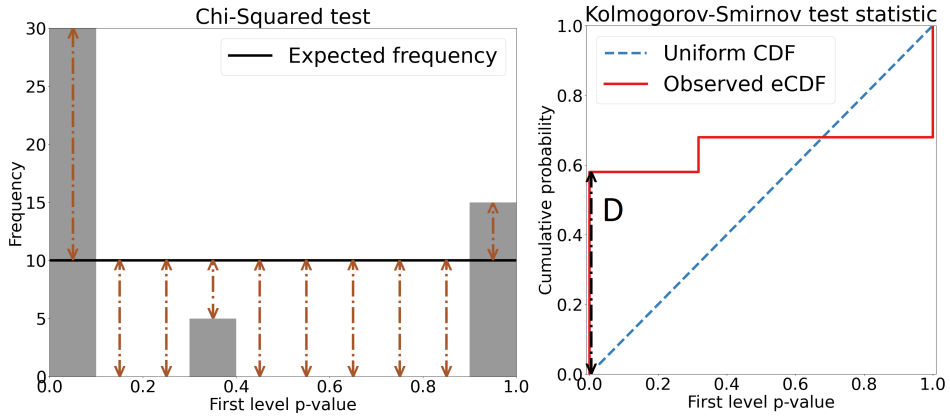

Figure 1.3: Visualisations for χ^2 and KS tests for two-level test example.

2 Programs for randomness testing

In this chapter, available programs for randomness testing are presented. First section describes *statistical testing suites* (often referred to as batteries). The second section shows *randomness testing toolkits*, which encompass several batteries together. Overview of setup and output is given in both sections.

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 described batteries are Dieharder, NISTS STS, several batteries from TestU01, FIPS and BSI.

An *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*, which are all executed at once when the individual test is executed.

rewrite better the example

Each *subtest* maps to one *two-level test*. The first-level test statistics of subtests are usually related to each other by using the same operation on the tested sequence. 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. [5]

Variant of the test is an individual test that is parametrized to search for modified pattern in the data. Only some individual tests allow such parametrization. Usually, all possible *variants* of given individual test are executed. [13, p. 2]

The following settings are available in *abstract battery*. They can be set either *globally* with same values for all *individual tests*, or *individually*:

- *number of first-level tests* - Sets how many times the first-level test is repeated (i.e. how many first-level p-values will be produced).

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

Originally, the testing batteries were designed to test the random number generators by taking data directly from them as needed. However, common way to test RNG is to take a sample of its output, store it into a file and then test contents of the file. The tested file is rewound to the beginning before each individual test (all tests are executed over the same sequence). When the generator is tested directly, every test is executed on 'fresh' sequence.

Testing files leads to need of *file rewind* detection or prevention, which are important part of each battery. File rewinds occurs when the tested file is not large enough for given test configuration. In this case, some parts of the file are tested twice during one test and the results may be biased. The required data size is calculated as *number of first-level tests* · *size of first-level sequence* and it is up to the user to prevent *file rewinds* by configuring the batteries.

2.1.1 Dieharder

The *Dieharder* battery was developed by Robert G. Brown at Duke University[5] as an extension to an older *Diehard* battery (created by George Marsaglia). The *Dieharder* is available from [14] maintained by Dirk Eddelbuettel or as package in some Unix distributions.

The Dieharder contains 31 individual tests. 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

The NIST[1] STS[2] was implemented by NIST [1] and is available from [3]. Due to the original implementation being slow, optimized version was developed by CRoCS[3] FI MU [15] available from [16]. *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.

---

1. National Institute of Standards and Technology
2. Statistical Test Suite
3. Center for Research on Cryptography and Security

More result information for each test is 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 A.2.

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

2.1.3 Test U01

TestU01 was developed by Pierre L'Ecuyer and his team at Université de Montrél to be a state-of-the-art software library oriented on testing of random number generators [9] and is available from [4]. It contains several batteries, most important are the *SmallCrush, Crush, BigCrush, Rabbit, Alphabit* and *BlockAlphabit*. Because TestU01 is a software library only, custom command-line interface was created by Ľubomír Obrátil available from [17], which is described in this subsection.

In general, the TestU01 batteries do not apply a two-level test, but only allow to repeat the single-level test. In TestU01, the *individual test* maps to the repeated *single-level test*. One run of the signle-level test is called *repetition*. If the user wishes to apply the second-level test, they have to examine the distribution of p-values on their own.

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

- *repetitions* argument sets how many times the single-level test is executed. This argument is used in all batteries.
- *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 in *BlockAlphabit* battery.

The output format is the same for all of TestU01 batteries and is printed to standard output. Only one individual test is executed during one battery run, for which the following is printed. For each repetition of single-level test the test name and all test parameters are printed. Then the test static value and its corresponding p-value are printed (or more values and p-values, if the individual test consists of more

subtests). After each repetition, a *generator state* is printed, containing information about how many data were used in all single-level tests so far. At the end of the individual test report, list of all p-values is printed. Example output is in Appendix A.3.

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*. Problems with the *bit_nb* argument are deeper described in Subsection 3.1.4.

Alphabit and BlockAlphabit

Both Alphabit and BlockAlphabit batteries contain the same nine individual tests. In the BlockAlphabit battery, the tested data are transformed to deploy *test variants*. The test variant is chosen by the *bit_w* argument parametrizing the transformation, which takes values 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[4] battery contains five tests and is based on FIPS 140-2 standard. [18] Custom command-line interface of the battery was created by Patrik Vaverčák available from [19]. The interface is based on implementation taken from [20]. No test variants are available and the length of first-level sequence is set to 2500 bytes and cannot be changed [13, p. 20]. Only one arguments is available:

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

Output of FIPS battery is printed to standard output and in user-specified file in JSON[5] 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 A.4

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[6] battery contains nine test and is based on series of standards released by BSI. [21] Custom command-line interface was created by Patrik Vaverčák, available from [19]. The tests implementations were extracted from the ParanoYa application. [13, 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.

4. Federal Information Processing Standards
5. JavaScript Object Notation
6. Bundesamt für Sicherheit in der Informationstechnik

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 A.5.

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.

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 needed) to set up parameters for each test from the battery individually based on size of the tested file and to run this test manually.

Since this approach is not convenient, Ľubomír Obrátil from CRoCS FI MU created the Randomness Testing Toolkit (*RTT*).[22] This toolkit allows users to run and configure several batteries using the same interface.

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*). [13]

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++ and 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. [22, p. 8]

RTT is available from [6], all used batteries are available from [17]. 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. [7]

RTT 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 has to be located in the working directory of the *RTT* [22, p. 10]. An example of this file is in Appendix B.1. These settings are usually set at the beginning and are not expected to change between runs. 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 optional. 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). Example of *rtt-settings.json* file is in Appendix B.1.

The battery configurations are dependent 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 they all follow the same format and are stored together in a single file. [22, p. 11] The *RTT* contains several prepared battery configurations for various sizes of tested file. Example of battery configuration file is in Appendix B.2.

For each battery, the settings are split into two parts - *defaults* and *test-specific-settings*. The *defaults* section contains IDs of individual tests to be run and default values of all battery arguments. The *test-specific-settings* is a list of all tests whose settings are different than those in

7. There is no note for the user regarding this.

defaults (along with new settings) or tests which employ test variants. In the second case, all variants to run are listed in entry for the given individual test.

RTT 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, the number of passed and executed 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 first part of the individual test report contains the name of the test and user settings. The second part of the individual test report is the second-level p-value alongside the name of statistic used (or more, in case the individual test consists of more subtests). At the end of the individual 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 Squeeze Test test results:
    Result: Passed
    Test partial alpha: 0.01000000

    User settings:
        P-sample count: 45
    ************

    Kolmogorov-Smirnov statistic p-value: 0.99967827       Passed
    p-values:
        0.02780241 0.03281157 0.07298400 0.09478713 0.12281035
        0.14670543 0.16186038 0.18184761 0.21224702 0.21916748
        0.23490332 0.26466112 0.28950133 0.31814240 0.32072127
        0.32585484 0.34516851 0.36957093 0.40652134 0.40899928
        0.46020869 0.47058812 0.48112758 0.49573165 0.50032144
        0.53830674 0.53962202 0.61275461 0.62835678 0.65830224
        0.67702142 0.68517115 0.70085331 0.71660690 0.73018719
        0.75465941 0.78340636 0.78444690 0.80495625 0.82032130
        0.86425311 0.88179709 0.90818439 0.92916504 0.99883153
    ============
------------------------------------------------------------
```

Figure 2.1: The example of individual 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 [13, p. 24] and it was written in Python.

The included batteries are Dieharder, NIST STS, FIPS battery and BSI battery. From TestU01, the *rtt-py* also includes Rabbit, Alphabit and BlockAlphabit batteries. 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 [7] and implementations of all used batteries are available from [19]. Before running, both the *rtt-py* and the batteries have to be installed as described in the project's README.

Rtt-py settings

The settings of *rtt-py* use same 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* [13, 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. [7]

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. [13, p. 25] The user has to keep in mind that the *rtt-py* uses FIPS and BSI batteries, which are not used in *RTT*.

Rtt-py output

The *rtt-py* creates output in two formats – *CSV*[8] and *HTML*[9]. [13, 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.

8. Comma-separated values
9. 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

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

Test: FIPS 140-2(2001-10-10) Runs

Failed runs	0
Runs	3999

Test: FIPS 140-2(2001-10-10) Long run

Failed runs	2
Runs	3999

Figure 2.3: The example of HTML FIPS battery report 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 reports more information compared to 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). The individual test has red background in case the test *rejected* the null-hypothesis, grey background otherwise. 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

3 Tests Analysis

This chapter aims to provide more information about the tests from the practical point of view. First, we analysed how much data the tests use to help with configurations of the batteries. Next, we measured how much time each test from the batteries takes to run. Data from both parts were used to create a program for automatic creation of test configurations for *RTT* and *rtt-py* – the *Configuration Calculator*. In the last section, we present a problem with non-uniform distributions of first-level p-values.

3.1 Data Consumption

As mentioned in Subsection 2.1, an important part of preventing *file rewinds* is correct configuration of the tests. This is a complicated task because the user has to know the size of test first-level subsequences. To help with this task, I performed an analysis of first-level subsequence sizes for Dieharder, SmallCrush, Crush and Rabbit batteries.

The NIST STS, Alphabit and BlockAlphabit batteries are not shown because the first-level subsequence size is set by the user. Unlike in the Rabbit battery, the tests from the mentioned batteries were observed to use the amount of data specified by their arguments. The BigCrush battery was skipped because its first-level subsequence sizes are larger than the currently intended use of RTT.

3.1.1 Prelude

The idea was to run all tests from the batteries and calculate how many bytes were used for each individual test or test variant based on data provided by the batteries. Based on this information, a table for each battery containing this information was created.

The tests are divided into categories based on two properties. The first property is user's influence on the first-level subseqeunce size. Second property is if *all* runs of the *same* first-level tests use *equal* size of first-level subsequence, or if the first-level subseqeunce size is different for each run.

In the subsequent text, I use the following terminology. Based on the first property, the tests are called as:

- *Tests with configurable size* are the tests where the size of first-level subsequence can be set by the user.
- *Tests with fixed size* have predefined and unchangeable first-level subsequence size.

Based on the second property, the categories are called:

- *Tests with constant size* are the tests where the first-level subsequence is equal for all runs.
- *Tests with variable size* are the tests where the first-level subsequence size fluctuates.

The size of the first-level subsequence in the tests with variable size is determined by *content* of the sequence. For example, the Diehard Squeeze Test starts with number $k = 2^{31}$ and finds j, the number of iterations needed to reduce k to 1 using the reduction $k = \lceil k \cdot U \rceil$, U is uniform float on $[0, 1)$ generated from 4 bytes of the tested data.

For an *individual* test from this category and for random content of the sequence, the *real* size of the first-level sequence fluctuates around *some* value.

To examine how long are the first-level sequences of tests with variable size, we run each test from this category at least 100 times on different *random* data (taken from */dev/random*). Tests from this category are presented in their own tables containing mean length of the subsequence and differences between mean and lowest and highest observed length.

To verify that tests with constant size are indeed constant, I run them several times with different number of first-level sequences. The table for them contains only length of the subsequence.

3.1.2 Dieharder

In Dieharder both tests with configurable and fixed size are present. All tests with configurable size have a default setting for first-level sequence size, which is usually used. Therefore all tests from this battery are in the analysis viewed as tests with fixed size. The subsequence sizes are presented in three tables.

Table 3.2 contains tests with *variable sizes*. The tests with *constant* sizes are split in two tables. In Table 3.1, the tests with *no variants* are shown, the tests *with varints* are shown in Table 3.3.

Table 3.1: First-level subseqeunces sizes for Dieharder tests with *constant* sizes and no test variants.

Test ID	size (bytes)	Test ID	size (bytes)
0	153,600	12	48,000
1	4,000,020	14	796
2	5,120,000	15	400,000
3	2,400,000	17	80,000,000
4	1,048,584	100	400,000
5	8,388,608	101	400,000
6	5,592,416	102	400,000
7	2,621,484	204	40,000
8	256,004	205	614,400,000
9	5,120,000	206	51,200,000
10	96,000	209	260,000,000
11	64,000		

Table 3.2: First-level subseqeunce sizes for Dieharder tests with *variable* sizes.

Test ID	mean size (bytes)	Δ min size	Δ max size
13	9,225,521	0.039%	0.038%
16	5,402,335	0.126%	0.077%
207	452,016,414	0.019%	0.015%
208	116,881,517	0.047%	0.014%

Table 3.3: First-level subseqeunce sizes for Dieharder tests with *constant* sizes and test variants.

ID	ntup	size (bytes)	ID	ntup	size (bytes)
200	1	800,004	203	7	32,000,000
200	2	1,600,004	203	8	36,000,000
200	3	2,400,004	203	9	40,000,000
200	4	3,200,004	203	10	44,000,000
200	5	4,000,004	203	11	48,000,000
200	6	4,800,004	203	12	52,000,000
200	7	5,600,004	203	13	56,000,000
200	8	6,400,004	203	14	60,000,000
200	9	7,200,004	203	15	64,000,000
200	10	8,000,004	203	16	68,000,000
200	11	8,800,004	203	17	72,000,000
200	12	9,600,004	203	18	76,000,000
201	2	80,000	203	19	80,000,000
201	3	120,000	203	20	84,000,000
201	4	160,000	203	21	88,000,000
201	5	200,000	203	22	92,000,000
202	2	800,000	203	23	96,000,000
202	3	1,200,000	203	24	100,000,000
202	4	1,600,000	203	25	104,000,000
202	5	2,000,000	203	26	108,000,000
203	0	4,000,000	203	27	112,000,000
203	1	8,000,000	203	28	116,000,000
203	2	12,000,000	203	29	120,000,000
203	3	16,000,000	203	30	124,000,000
203	4	20,000,000	203	31	128,000,000
203	5	24,000,000	203	32	132,000,000
203	6	28,000,000			

3.1.3 TestU01 SmallCrush and Crush

In the SmallCrush and Crush batteries, only tests with *fixed* size are present. Both batteries contain tests with *variable* size, which are shown in Table 3.4 for SmallCrush and in Table 3.6 for Crush. The tests with *constant* sizes from SmallCrush are in Table 3.5, from Crush in Table 3.7.

Table 3.4: First-level subseqeunce sizes for *SmallCrush* tests with *variable* sizes.

Test ID	mean size (bytes)	Δ min size	Δ max size
3	204,733,510	0.563%	0.725%
5	98,731,035	0.095%	0.087%

Table 3.5: First-level subseqeunce sizes for *SmallCrush* tests with *constant* sizes.

Test ID	size (bytes)
1	40,000,000
2	40,000,000
4	102,400,000
6	48,000,000
7	204,800,000
8	28,800,000
9	120,000,000
10	20,000,000

Table 3.6: First-level subseqeunce sizes for *Crush* tests with *variable* sizes.

Test ID	mean size (bytes)	Δ min size	Δ max size
27	1,333,326,306	0.0163%	0.0166%
28	1,333,331,101	0.0188%	0.0170%
29	1,974,653,548	0.0189%	0.0196%
30	1,974,640,110	0.0164%	0.0146%
31	3,200,051,704	0.0204%	0.0306%
32	3,199,995,653	0.0380%	0.0244%
33	5,120,635,483	0.1408%	0.1618%
34	5,119,635,549	0.1399%	0.2516%
55	1,653,334,290	0.0065%	0.0060%
64	320,000,610	$5.31 \cdot 10^{-5}$%	$4.43 \cdot 10^{-5}$%
91	533,332,642	0.0039%	0.0037%
92	1,599,996,811	0.0049%	0.0043%

Table 3.7: First-level subseqeunce sizes for *Crush* tests with *constant* sizes.

Begin of Table			
Test ID	size (bytes)	Test ID	size (bytes)
1	2,000,000,000	52	2,048,000,000
2	1,200,000,000	53	2,048,000,000
3	400,000,000	54	2,048,000,000
4	400,000,000	56	480,000,000
5	400,000,000	57	1,440,000,000
6	400,000,000	58	600,000,000
7	400,000,000	59	1,800,000,000
8	400,000,000	60	384,000,000
9	400,000,000	61	1,152,000,000
10	400,000,000	62	2,400,000,000
11	800,000,000	63	800,000,000
12	1,200,000,000	65	600,000,000
13	1,600,000,000	66	360,000,000

Continuation of Table 3.7

Test ID	size (bytes)	Test ID	size (bytes)
14	1,680,000,000	67	680,000,000
15	1,680,000,000	68	400,000,000
16	1,920,000,000	69	668,000,000
17	1,920,000,000	70	400,000,000
18	160,000,000	71	480
19	240,000,000	72	480
20	280,000,000	73	44,739,280
21	128,000,000	74	109,400,000
22	128,000,000	75	327,800,000
23	2,560,000,000	76	1,333,336,000
24	2,560,000,000	77	1,200,002,400
25	2,560,000,000	78	1,200,000,000
26	2,560,000,000	79	1,200,000,000
35	2,000,000,000	80	1,333,360,000
36	2,000,000,000	81	1,200,000,000
37	2,000,000,000	82	2,000,000,000
38	2,000,000,000	83	2,000,000,000
39	2,600,000,000	84	1,600,000,000
40	2,600,000,000	85	2,400,000,000
41	2,000,000,000	86	2,400,000,000
42	2,000,000,000	87	2,400,000,000
43	800,000,000	88	2,400,000,000
44	1,200,000,000	89	3,200,000,000
45	400,000,000	90	960,000,000
46	1,200,000,000	91	533,332,642
47	800,000,000	92	1,599,996,811
48	2,000,000,000	93	1,333,333,400
49	820,000,000	94	2,000,000,020
50	880,000,000	95	1,333,333,400
51	2,048,000,000	96	2,000,000,040

End of Table

3.1.4 TestU01 Rabbit

All tests in the Rabbit battery are with *configurable* size. The size is configured using the *bit_nb* argument, where the desired size is entered in *bits* and for use is rounded down to closest multiple of 32. The *bit_nb* has no default value, must be at least 500 and *at most bit_nb* bits will be read from the file. [2, p. 152] However, almost half of the tests require bigger *bit_nb* and will not run otherwise. These tests and their required sizes are in Table 3.8. The minimums were found and verified experimentally.

Table 3.8: Minimal values of *bit_nb* argument other than default needed to run tests from *Rabbit* battery.

Test ID	minimal *bit_nb*
9	31,200
10	960
11	960
15	960
16	1,920
17	3,840
21	51,200
22	5,120,000
23	52,428,800
24	960
25	30,720
26	300,480

Tests 6, 7 and 8 read only data with size $2^k (k \in \mathbb{N})$ *bits*. The possible values of k are different for each test and shown in Table 3.9. [2, p. 124-126] If the *bit_nb* is not power of two, closest lower applicable power of 2 is used. The *bit_nb* still has to be at least 500, even though smaller amount of bits may be used.

The only test with *variable* size from Rabbit battery is the test 20. It uses around 80 % of the data size specified by *bit_nb* argument. Test 5 reads significantly less data than specified. When the *bit_nb* argument specifies that the test should use 10MB of data, the test 5 uses only 3,536 *bytes*. When the test should use 100MB of data, it uses only 8,488 *bytes*.

Table 3.9: Maximal values of k for Rabbit tests which take data of size 2^k *bits*.

Test ID	maximal k
6	28
7	20
8	26

For some tests with *constant* size, there is an *upper bound* of data used for testing. The test will not read more data than this value, even when specified by the *bit_nb* argument. The size of data that are actually read from the file is different for each value of *bit_nb* greater than the *upper bound* and fluctuates around *some* value. Examples of real data size fluctuation are in the Figure 3.1.

Figure 3.1: Number of bytes actually read from tested file by Rabbit test with ID 3 for various values of *bit_nb* greater than $3.3 \cdot 10^9$ ($\approx$ 393 megabytes).

3.2 Time consumption and performance

Another analysis we performed is examination of the tests' runtimes and performance. This may be used to *exempt* tests that would take *unreasonably* long to execute or to detect problems while running the tests (for example when the test execution gets stuck on non-random data).

Because the runtimes highly depend on used machine and the idea is to make relative comparison between the tests only, they are presented in *relative* form compared to the runtime of Diehard 3d Sphere

(Minimum Distance) Test from Dieharder battery (later refered to as *time unit* or *TU*).[1]

To compare performance of the tests, *throughput* was chosen as a measure. It is presented alongside the runtime. The throughput is calculated as kilobytes[2] of data processed per time unit.

The used implementations of batteries are taken from RTT (available from [17]). The batteries were run directly (i.e. not using RTT) and each individual test was executed in separate at least one hundred times. Mean runtime over these runs is presented in this Section.

The runtimes and throughputs for Dieharder are in Table 3.10. The Dieharder tests were run without the *rate* option and with default values of *tsample* argument. Runtimes and throughputs for TestU01 SmallCrush and Crush batteries are in Table 3.13 and Table 3.14.

To measure runtimes of test with *configurable* size, I had to fix the first-level subsequence size for each test. For NIST STS battery, I set *stream size* argument to 1,000,000 (the subsequences were 125,000 bytes long). This is the lowest value which satisfies subsequence size recommendations for all the tests. [1] NIST STS runtimes and throughputs are shown in Table 3.12.

For TestU01 Rabbit, Alphabit and BlockAlphabit batteries, I fixed the *bit_nb* argument to value 52,428,800 (the subseqeunces should be 6,553,600 bytes long). This is the lowest value for which all tests from the three batteries are executed (see Subsection 3.1.4). The runtimes and throughputs for Rabbit battery are in Table 3.16. Alphabit and BlockAlphabit batteries share Table 3.15, because both batteries employ the *same* tests and no difference in runtimes was observed.

1. Running on Intel Core i7-1065G7 CPU under Ubuntu 22.04.1 WSL on Windows 10 this test took 0.021 seconds.
2. 1024 bytes

Table 3.10: Runtimes for and throughputs Dieharder battery tests without variants.

ID	runtime (TU)	Throughput (kB/TU)	ID	runtime (TU)	Throughput (kB/TU)
0	0.43	347	14	$3.03 \cdot 10^{-5}$	25598
1	2.93	1332	15	0.25	1585
2	9.28	539	16	2.86	1846
3	2.10	1116	17	75.79	1031
4	0.91	1126	100	0.19	2024
5	4.79	1710	101	2.73	143
6	33.36	164	102	4.29	91
7	10.01	256	204	0.06	604
8	0.21	1208	205	311.59	1926
9	2.86	1751	206	44.78	1117
10	0.73	128	207	272.13	1622
11	0.27	236	208	180.61	632
12	1.00	47	209	266.52	953
13	5.70	1581			

Table 3.11: Runtimes and throughputs for Dieharder battery tests with variants.

ID	ntup	runtime (TU)	Throughput (kB/TU)	ID	ntup	runtime (TU)	Throughput (kB/TU)
200	1	1.18	662	203	7	16.04	1948
200	2	1.59	983	203	8	17.64	1993
200	3	1.89	1238	203	9	19.59	1994
200	4	2.30	1359	203	10	21.24	2023
200	5	3.02	1295	203	11	23.44	2000
200	6	4.20	1116	203	12	25.24	2012
200	7	5.98	915	203	13	27.15	2014
200	8	7.06	885	203	14	29.19	2008
200	9	8.90	790	203	15	30.76	2032
200	10	11.40	685	203	16	32.89	2019
200	11	15.90	540	203	17	35.10	2003
200	12	24.84	377	203	18	36.79	2017
201	2	0.28	279	203	19	38.98	2004
201	3	0.34	347	203	20	40.36	2032
201	4	0.50	310	203	21	42.82	2007
201	5	0.87	225	203	22	44.86	2003
202	2	0.47	1666	203	23	46.18	2030
202	3	0.74	1590	203	24	49.11	1989
202	4	1.09	1436	203	25	50.01	2031
202	5	1.89	1033	203	26	51.24	2058
203	0	2.00	1956	203	27	55.30	1978
203	1	3.86	2025	203	28	57.85	1958
203	2	5.88	1994	203	29	58.84	1992
203	3	7.70	2030	203	30	61.32	1975
203	4	9.90	1973	203	31	64.50	1938
203	5	11.67	2009	203	32	65.41	1971
203	6	13.63	2006				

Table 3.12: Runtimes and throughputs for NIST STS battery with *stream size* 1,000,000 bits.

Test ID	runtime (TU)	throughput (kB/TU)
1	0.16	782
2	0.17	716
3	0.16	765
4	0.19	658
5	0.17	711
6	0.24	500
7	5.79	21
8	0.77	158
9	0.20	597
10	0.22	560
11	0.22	561
12	0.43	283
13	0.39	314
14	0.93	132
15	1.55	79

Table 3.13: Runtimes and throughputs for TestU01 SmallCrush battery.

Test ID	Runtime (TU)	Throughput (kB/TU)
1	49.20	794
2	36.48	1071
3	16.21	12361
4	16.74	5973
5	13.03	7402
6	17.24	2719
7	12.94	15451
8	16.07	1750
9	22.76	5149
10	25.56	764

Table 3.14: Runtimes and throughputs for TestU01 *Crush* battery.

Begin of Table					
Test ID	Runtime (TU)	Throughput (kB / TU)	Test ID	Runtime (TU)	Throughput (kB / TU)
1	691.72	2824	49	239.32	3346
2	412.71	2839	50	82.72	10389
3	706.53	553	51	101.95	19618
4	744.61	525	52	131.29	15233
5	1001.45	390	53	147.27	13580
6	1014.13	385	54	152.78	13090
7	1046.35	373	55	109.68	14721
8	1033.21	378	56	770.92	608
9	995.59	392	57	808.59	1739
10	1005.83	388	58	1164.27	503
11	960.15	814	59	1379.70	1274
12	972.95	1204	60	1567.96	239
13	1001.15	1561	61	2014.98	558
14	644.60	2545	62	164.67	14233
15	675.67	2428	63	513.08	1523
16	651.24	2879	64	148.90	2099
17	644.66	2908	65	843.17	695
18	296.66	527	66	207.06	1698
19	408.85	573	67	682.89	972
20	763.62	358	68	177.12	2205
21	290.98	430	69	581.06	1123
22	249.12	502	70	157.27	2484
23	356.42	7014	71	522.24	1
24	418.57	5973	72	518.17	1
25	340.36	7345	73	632.92	69
26	414.22	6035	74	561.55	190
27	175.32	7427	75	599.53	534
28	210.22	6194	76	2024.52	643
29	217.71	8858	77	708.19	1655
30	256.83	7508	78	599.47	1955
31	227.35	13745	79	371.55	3154
32	288.64	10827	80	335.88	3877

Continuation of Table 3.14					
Test ID	Runtime (TU)	Throughput (kB / TU)	Test ID	Runtime (TU)	Throughput (kB / TU)
33	249.16	20070	81	221.70	5286
34	310.39	16108	82	544.90	3584
35	210.76	9267	83	521.43	3746
36	277.00	7051	84	401.77	3889
37	685.43	2849	85	648.33	3615
38	694.09	2814	86	486.71	4815
39	1217.74	2085	87	642.08	3650
40	1261.90	2012	88	478.61	4897
41	802.82	2433	89	835.60	3740
42	474.13	4119	90	184.36	5085
43	115.48	6765	91	796.39	654
44	131.44	8916	92	952.65	1640
45	80.57	4848	93	540.12	2411
46	128.67	9108	94	659.61	2961
47	157.27	4968	95	447.90	2907
48	146.23	13356	96	527.98	3699
End of Table					

Table 3.15: Runtimes and throughputs for TestU01 Alphabit and Block-Alphabit batterires with *bit_nb* 52,428,800.

Test ID	Runtime (TU)	Throughput kb/TU
1	3.49	1835
2	3.51	1825
3	3.36	1902
4	5.72	1119
5	2.60	2459
6	2.11	3034
7	2.06	3109
8	10.52	608
9	8.07	793

Table 3.16: Runtimes and throughputs for TestU01 Rabbit battery with *bit_nb* 52,428,800

ID	runtime (TU)	throughput kb/TU	ID	runtime (TU)	throughput kb/TU
1	448.60	14	14	2.16	2964
2	16.52	387	15	2.60	2458
3	11.68	548	16	2.10	3053
4	1.59	4019	17	2.17	2953
5	20.32	0	18	3.01	2129
6	70.27	58	19	3.00	2136
7	2.57	50	20	8.55	599
8	20.29	315	21	10.62	603
9	10.62	603	22	13.26	483
10	3.62	1769	23	22.71	282
11	2.14	2988	24	9.26	691
12	2.07	3091	25	7.02	912
13	2.04	3143	26	6.08	1052

3.3 Configuration Calculator

appendix examples, git link (when ready)

The information collected in Sections 3.1 and 3.2 were used in the *Configuration Calculator*. I created this tool to automate creation of battery configuration files for *RTT*. Configuration Calculator supports all batteries that are present in *RTT* except the TestU01 BigCrush battery.

3.3.1 Batteries configurations

The configurations are created based on the length of the tested data. Number of first-level subsequences for each individual test is set so that the test will use as much data as possible without *file rewind*.

For tests with *configurable* size, the default values were chosen the same as in Section 3.2. For NIST STS the *stream size* is set to 1,000,000 *bits* and for TestU01 Rabbit, Alphabit and BlockAlphabit, the *bit_nb* is

set to 52,428,800 *bits*. The user may choose their own value for these arguments.

For all individual test the number of first-level tests is calculated as

$$\lfloor tested\ file\ size \div firstlevel\ subsequence\ size \rfloor$$

For tests with *constant* size, the direct subsequence size is used. For tests with *variable* size the fluctuation of first-level sequence sizes has to be taken into account to prevent *file rewinds*. To achieve this, mean first-level subseqeunce size increased by a buffer is used as the subsequence size. The buffer size was chosen to be 1% of the subseqeunce size for tests from TestU01 batteries and 0.1% for tests from Dieharder. Based on analysis from Section 3.1, this should be enough to prevent file rewinds caused by the size fluctuation.

In total 7 tests are always omitted by the Configuration Calculator. Tests 5, 6, 7 and 14 from Dieharder battery are skipped because they are marked as 'Do Not Use' or 'Suspect' by Dieharder. From Crush battery tests 71 and 72 were removed due to their low performance (see Table 3.14). Running each of the two tests would take 12 times longer than running *all* of the remaining tests. The test 5 from Rabbit was also omitted due to low performance (see Table 3.16).

3.3.2 Output Format

Format of the configuration files created by the Configuration Calculator is extension of the original format used by *RTT* (described in Subsection 2.2.1). It contains no new *functional* fields, only information for the user. First such field is *omitted-tests* inside *battery-settings* – tests that will not be run. Either because the tested file is to small for them or because they were removed by default.

Second new field is *battery-defaults*. It contains default argument settings for a given battery and default settings for each individual test, along with the test name and used first-level subsequence sizes. Both *battery-settings* and *battery-defaults* contain also comments for some individual tests with information about why the test was omitted or warning that a given test is with *variable* size.

3.4 P-values uniformity

As was mentioned in Section 1.3, crucial part of two-level testing is examination of observed first-level p-values distribution. The expected theoretical distribution is uniform on interval (0,1]. [10, p. 14] Due to various reaons, the real distribution (under the null-hypothesis) may differ from the uniform distribution. This may lead to flawed results of second-level tests. In this section, reasons why this happens are presented along examples of real observed distributions of first-level p-values. It is not goal of this thesis to fix the problem, only to note it.

In the following text, some examples of tests with non-uniform distributions are presented. The data for real distributions were acquired by repeatedly generating files with 5GB of random data using the AES[3] with known key as random number generator. Each file was tested using the *RTT* with battery configuration generated by the Configuration Calculator. [4]

The reason why non-uniform distributions are observed is that the distributions of test statistic values in first-level tests are usually approximated in the calculation. This is usually done because the real distribution is know only asymptotically or because it is more efficient. Also, the distribution of test statistic values (and therefore p-values), for a fixed sequence size, is in fact discrete, but a continuous distribution is often used as approximation to calculate the p-value. This causes error, which accumulates alongside the error caused by limited-precision calculations. [10, p. 7]

Possible outcomes of previous situations are non-uniform distributions of the first-level p-values. The first situation are discrete distributions with *low* number of possible p-values. In the worst case, the tests produce lower hundreds of possible first-level p-values. For example, the Dieharder test 10 produced 221 different p-values over 10,000,000 runs, while the NIST STS test 7 produced 485 different p-values over 10,000,000 runs. Histograms of first-level p-values of the two tests are in the upper half of Figure 3.2.

Second situation are tests with high number of observed p-values, but non-uniform distributions. The real distributions are usually *close*

3. Advanced Encryption Standard

4. The dataset is part of yet unpublished manuscript SÝS, Marek; BROŽ, Milan; MAREK, Tomáš. Correcting Dieharder, NIST STS, TestU01 batteries. 2023.

to the uniform distribution, but differ in tails (values in tail are significantly more or less probable). Two example histograms of such situation are in the lower half of Figure 3.2.

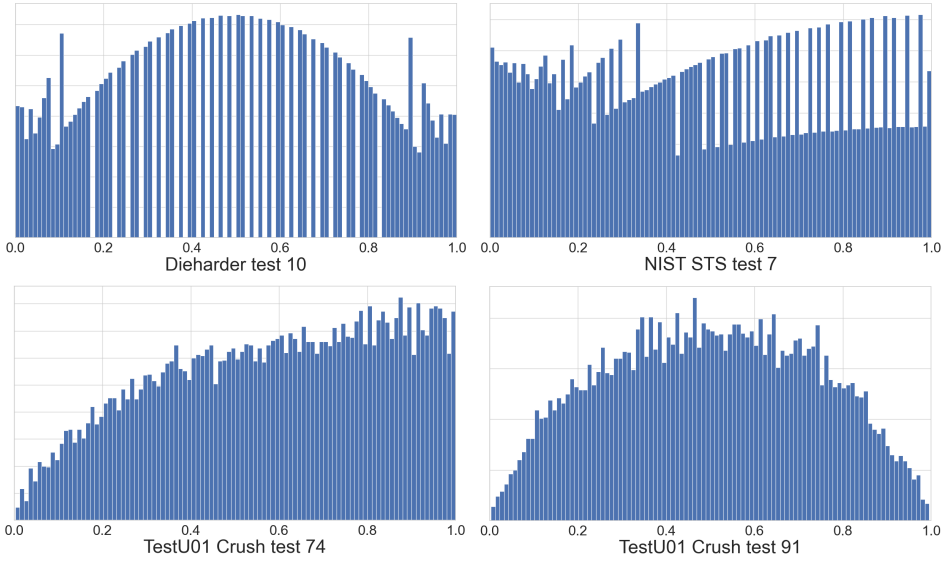

Figure 3.2: Histograms for examples of non-uniform first-level p-values distributions.

4 Implementations comparison

The *rtt-py* is supposed to be a newer version of *RTT* and might replace it in the future. Therefore, in this chapter we focus mainly on finding differences between the two. The first section aims at comparing outputs of both testing toolkits. In the second section I am at finding functional differences between them, especially the features the are present in *RTT* and not in *rtt-py*. In the last section I present general suggestions for further improvement of both *RTT* and *rtt-py*.

4.1 Output

Example of output from *RTT* can be found in Appendix B.3 and the output of *rtt-py* can be found in B.4. Both reports come from executions over the *same* data file and using the *same* battery configuration file. The output examples are shortened, but the same tests are presented in all examples if possible.

First difference in output between *RTT* and *rtt-py* is the format of the output. *RTT* offers reports in plaintext format, which contains all of the reported information (see Subsection 2.2.1). The format is human-readable, but hard to navigate. Computer processing is possible thanks to unified format, but requires rather complicated parsing.

The *rtt-py* offers an overview table in HTML and CSV formats, which are both easily human-readable and useful for quick assessment of the tests results. The CSV format is also easily computer-readable.

The full HTML format offers more information, usually regarding the test settings. For all tests, the p-values is contained. In the NIST STS, the proportion of sequences passing the first-level test is added. In the TestU01 batteries, corresponding test statistics values are part of the HTML report as well.

The main functional difference in output between *RTT* and *rtt-py* are the reported information. For this comparison, the full HTML report from *rtt-py* is taken. Both toolkits report the resulting second-level p-values for Dieharder and both the second-level p-value and proportion of sequences passing the first-level test for NIST STS batteries. However, unlike *RTT*, the *rtt-py* does not report first-level p-values, which could be useful for deeper assessment of the results. Unlike the

RTT, *rtt-py* also does not report information regarding the individual tests settings (stream size and counts) from NIST STS battery.

For the TestU01 batteries (which employ only repeated single-level tests) the reported information are the same for both toolkits – p-values and details about test settings. However, in *RTT* report, repetitions of the same individual test are grouped together and their respective p-values are printed in a group. In *rtt-py*, each repetition is treated as a separate individual test and the user has to manually group the tests by their names. This is important, because the TestU01 does not perform the two-level test and if the user wishes to apply it, they have to do so on their own. The FIPS and BSI batteries are not mentioned in this comparison, because they are only executed in the *rtt-py*.

Both toolkits also take different approach in parsing the results. In *RTT*, the results are parsed after *all* tests finish their executions using regular expressions. Originally, the regular expressions used for TestU01 batteries sometimes crashed while parsing. This issue has already been fixed. [1] Furthermore, when parsing large batteries outputs, the *RTT* may crash due to *out of memory* error.

In *rtt-py*, the results are parsed after each individual test is executed and stored in pre-defined objects. After all tests from given battery are executed, corresponding report is generated.

Another part of the output of both toolkits is handling errors and warning from the batteries. The *RTT* informs the user about the number of errors and warnings that occurred on the standard output. The detection of errors and warnings in *RTT* is done by searching for words 'error' or 'warning' in the battery output. At the beginning of the report file, all tests that produced errors or warnings are listed. The whole line containing the warning or error is printed into the corresponding individual test report.

The *rtt-py* ignores errors and warnings from test batteries. The most notable example why this is a problem are the *file rewinds* (as described in Section 2.1).

In this case, the test will read some parts of the data more than once and inform the user about this situation. The test will still produce result, which will, however, be biased by repeated parts of the tested

1. `https://github.com/crocs-muni/randomness-testing-toolkit/commit/7457ba6e820a886f4582f271b2b1377d14152c60`

file. This may lead to incorrect interpretation of the results and to Type I or II error. Since the *rtt-py* ignores this, there is no way for the user to be informed about this situation.

4.2 Implementations differences

In this section I focus on finding the functional differences between *RTT* and *rtt-py*. The primary goal is to find features, that are missing in the *rtt-py* and should be implemented to allow the replacement of *RTT* with *rtt-py*.

Both *RTT* and *rtt-py* employ different set of batteries. The Dieharder, NIST STS, TestU01 Rabbit, Alphabit and BlockAlphabit batteries are run in both implementations. The FIPS and BSI batteries are only executed in the *rtt-py*. The TestU01 *Crush* family of batteries is only run by *RTT*. The program options of *rtt-py* however clearly suggest, that these batteries should also be run. There is no explanation why neither in [13] or [7].

Regarding the batteries, the *RTT* runs one user-chosen battery per run. In the *rtt-py*, all batteries are run by default. The user can choose batteries to omit from the run. The *RTT* runs either all tests from the current battery configuration, or only single test chosen by the user, while in *rtt-py* all tests from given configuration are run. Also the *rtt-py* allows the user to test more files with data during one run, while *RTT* allows testing of a single file only.

Another difference between toolkits is handling the *test-specific-settings* field from battery configuration files. In *RTT*, this field is optional for all batteries. However, in *rtt-py* this field is *required* for Dieharder and NIST STS battery and the configuration will not parse otherwise. This may cause problems with compatibility of configuration files between *RTT* and *rtt-py*.

Another difference with this field is that *rtt-py* ignores *repetitions* field from *test-specific-settings* of TestU01 batteries, therefore all test are executed with. This also might to be the reason why *Crush* family batteries are not run, because this field is needed for *meaningful* use of the *Crush* family batteries, while the remaining TestU01 batteries can be *meaningfully* used without changing the *repetitions* argument.

Closely tied with the warnings and errors are the output logs of the executed batteries. They are usually used to examine the warnings and errors raised by the batteries and to decide about the severity of the warning or error. Another usage may be to examine further details of the report that are not reported by the used toolkit.

In *RTT*, the output logs for each battery are stored in the *results* folder. In *rtt-py*, no batteries output logs are being stored. Both *RTT* and *rtt-py* create run logs containing the exact arguments used to run the batteries.

As was described in previous section, unlike *RTT*, the *rtt-py* does not report first-level p-values from Dieharder and NIST STS. The first-level p-values can be usefull for deeper examination of the tests results. In TestU01, the p-values are grouped by test in *RTT*, but not in *rtt-py*. Furthermore, as opposed to *RTT*, the *rtt-py* ignores the warning from battery outputs.

The *RTT* allows parallel execution of several individual tests at once (maximal number of parallel exectuions is set by the user in *rtt-settings.json* file). In *rtt-py*, no parallelism is supported and all individual tests are executed serially.

One of the original ideas of *RTT* was to provide interface for easy addition of new batteries, possibly by the user. In *rtt-py*, the user only has to implement classes for settings parsing, executions and results parsing, which all follow predefined patterns. These classes are then easily integrated into the *rtt-py*.

Compared to this, the source code of *RTT* is much more interconnected. The user has to implement several classes regarding the executions and results parsing as well. For the settings parsing, however, the user has to extend the already existing class. The user also has to extend several another classes responsible for setting up and executing the batteries. That makes adding a new battery into the *RTT* harder than into the *rtt-py*

The *RTT* also offers storing the test results in configured database, which is not offered by the *rtt-py*. This functionality is, however, often not used. Because the used database connection is not supported on some Unix distributions, the *RTT* may be compiled without it.

Table 4.1: Overview table comparing the features of *RTT* and *rtt-py*.

Feature	RTT	rtt-py
Missing batteries	FIPS, BSI	TestU01 SmallCrush, Crush, BigCrush
Dieharder and NIST first-level p-values	yes	no
TestU01 p-values from single individual test	grouped	each repetition separate
Battery errors and warning	yes	no
Battery output logs	yes	no
Run logs	yes	yes
Parallel executions	yes	no
Readiness for new battery	less	more
Batteries in one run	single	all
Tests in one run	all or single	all

4.3 Proposed improvements

First proposed improvement for both *RTT* and *rtt-py* is better way to configure the batteries. While *RTT* offers some prepared configuration files, they are not available for all file sizes. Also, some of the configuration files contained configurations that resulted in *file rewinds* or the tests read less data than was possible, resulting in testing only a part of the file. I implemented this improvement in the form of Configuration Calculator (see Subsection 3.3).

Second proposed improvement are better second level-tests. As is shown in Subsection 3.4, p-values of many first-level tests do not follow the uniform distribution. The improved second-level tests should take these non-uniform distributions into account and compare the observed distributions to the real distributions of the tests. The improved second-level tests also should provide second-level assessment of the TestU01 batteries.

Third proposed improvement is to extend the output format of *RTT* to computer-readable format, possibly using JSON format as in configurations. This could be used not only for quick computer-made results assessments, but also for the custom second-level tests.

5 Conclusion

We analysed the *Randomness Testing Toolkit* and its newer, alternative implementation, the *Randomness Testing Toolkit in Python*. The analysis focused on finding differences between the two implementations in order to allow future replacement of *RTT* with *rtt-py*, and on proposing improvements for both toolkits.

First, we analysed the individual tests present in the randomness testing batteries. During the tests analysis we collected information about *data consumption* and *runtimes* of the individual tests. We used the data to detect tests with low performance. As such were marked TestU01 Crush tests with ID 71 and 72, and TestU01 Rabbit test with ID 5. In the future, the data may be used for further analysis in combination with other factors (for example "strength" of the test).

We created the *Configuration Calculator* based on the data from the tests analysis – tool for automated creation of battery configuration files for both *RTT* and *rtt-py*. This tool has already been used by CRoCs and will be published to use with the *RTT*.

We performed the analysis of both *RTT* and *rtt-py* by comparing the features of both implementations. The focus was on output of the toolkits (both format of the reports and the information contained in them) and features, that are missing in the *rtt-py* compared to the *RTT*.

The most notable differences between *RTT* and *rtt-py* are the output format (plaintext in *RTT*, HTML in *rtt-py*) and that, unlike *RTT*, the *rtt-py* does not report first-level p-values for Dieharder and NIST STS battery. Furthermore, the *rtt-py* does not report errors from batteries. In regard to functionality, *rtt-py* does not contain TestU01 *Crush* family of batteries and *RTT* does not contain FIPS and BSI batteries. The proposed improvements for both toolkits are to implement second-level tests which take into account the non-uniform distributions of first-level p-values of the individual tests, and computer-readable format of the ouput.

In the future, this thesis can be followed by implementing the missing features of the *rtt-py* listed in the implementations comparison, so that it can fully replace the older *RTT*. Furthermore, both *RTT* and

rtt-py can be extended based on the proposed improvements from the last chapter, particularly the improved second-level tests.

A Batteries output examples

A.1 Dieharder

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
 file_input_raw|                  <tested file>|  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
          sts_serial|   4|    100000|     100|0.91202326|  PASSED
          sts_serial|   4|    100000|     100|0.97759751|  PASSED
          sts_serial|   5|    100000|     100|0.22260482|  PASSED
          sts_serial|   5|    100000|     100|0.28835866|  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: Example of first-level p-values printout from *Dieharder* battery

A.2 NIST STS

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <tested file>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 33  21  29  26  33  33  40  40  24  21  0.098526   0.9900    Frequency
 33  27  28  27  23  32  25  45  29  31  0.262249   1.0000    BlockFrequency
 25  36  29  27  25  34  31  29  32  32  0.906970   0.9900    CumulativeSums
 32  26  20  35  36  27  29  39  28  28  0.407091   0.9900    CumulativeSums
 35  35  25  32  34  30  27  25  25  32  0.810470   0.9967    Runs
 24  32  32  34  24  35  23  34  33  29  0.685579   0.9867    LongestRun
 23  37  39  19  29  37  26  32  27  31  0.178278   0.9967    Rank
 35  22  31  27  34  27  28  31  33  32  0.856907   0.9900    FFT
 43  24  33  28  30  30  33  24  29  26  0.407091   0.9833    Universal
 28  33  29  28  27  26  40  24  32  33  0.699313   0.9900    ApproximateEntropy
             .
             .
             .
 22  27  17  11  17  19  19  16  21  19  0.427082   0.9894    RandomExcursions
 23  17  23  24  21  15  18  19  14  14  0.568055   0.9947    RandomExcursions
 12  20  15  15  17  21  28  21  21  18  0.324180   0.9947    RandomExcursions
 22  29  10  16  18  14  25  16  21  17  0.071670   0.9840    RandomExcursions
 19  17  15  24  23  16  15  16  25  18  0.568055   0.9787    RandomExcursions
 17  18  18  24  24   8  20  18  18  23  0.260784   1.0000    RandomExcursions
 15  21  19  23  19  24  18  13  23  13  0.468595   0.9947    RandomExcursions
 19  18  14  23  23  15  17  23  17  19  0.761937   0.9947    RandomExcursions
 35  35  23  31  30  29  35  34  28  20  0.514124   0.9933    Serial
 33  32  39  29  30  28  27  33  28  21  0.664861   1.0000    Serial
 39  26  30  23  30  29  40  28  23  32  0.339799   0.9867    LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 0.975064 for a
sample size = 300 binary sequences.

The minimum pass rate for the random excursion (variant) test
is approximately 0.971133 for a sample size = 188 binary sequences.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure A.3: Example of results table from the *NIST STS* battery.

```
                    FREQUENCY TEST
               -------------------------------------------
               COMPUTATIONAL INFORMATION:
               -------------------------------------------
               (a) The nth partial sum = -852
               (b) S_n/n               = -0.000852
               -------------------------------------------
SUCCESS              p_value = 0.394214

                    FREQUENCY TEST
               -------------------------------------------
               COMPUTATIONAL INFORMATION:
               -------------------------------------------
               (a) The nth partial sum = 1036
               (b) S_n/n               = 0.001036
               -------------------------------------------
SUCCESS              p_value = 0.300202

                    FREQUENCY TEST
               -------------------------------------------
               COMPUTATIONAL INFORMATION:
               -------------------------------------------
               (a) The nth partial sum = 102
               (b) S_n/n               = 0.000102
               -------------------------------------------
SUCCESS              p_value = 0.918757

               .
               .
               .

                    FREQUENCY TEST
               -------------------------------------------
               COMPUTATIONAL INFORMATION:
               -------------------------------------------
               (a) The nth partial sum = -612
               (b) S_n/n               = -0.000612
               -------------------------------------------
SUCCESS              p_value = 0.540538
```

Figure A.4: Example of p-values, test statistics and other information from NIST STS's *stats.txt* file.

A.3 TestU01

```
*********************************************************
HOST = <hostname>, Linux

Generator providing data from binary file.


smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,    d = 1073741824,    t = 2,    p = 1


      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =      27.1051


-------------------------------------------------------
Total expected number = N*Lambda      :      27.11
Total observed number                 :      23
p-value of test                       :    0.75


-----------------------------------------------
CPU time used                    :  00:00:01.07

Generator state:
========= State of the binary file stream generator =========
          Bytes needed for testing:     40000000
          Bytes read from file:         41943040
          Total number of file rewinds: 0


========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        Generator providing data from binary file.
 Number of statistics:  20
 Total CPU time:   00:00:22.23

 All tests were passed
```

Figure A.5: Example of individual test repetition report from *TestU01* SmallCrush battery.

```
=== First level p-values/statistics of the test ===
0.50363745
0.86329283
0.10179171
0.46486353
0.26887034
0.53671637
0.92729269
0.88570386
0.61845505
0.66933429
0.74972300
0.58255307
0.08685074
      .
      .
      .
0.63414594
0.33428285
0.72446307
0.71331799
0.79189429
0.56142960
0.24677228
0.13605986
0.36237823
0.33936047
0.56275714
0.34780855
0.54042762
0.26310714
================================================
```

Figure A.6: Example of p-values printout from *TestU01* battery.

A.4 FIPS battery

```
{
  "accepted": false,
  "sequence": <tested file>,
  "tests": [
    {
      "name": "FIPS 140-2(2001-10-10) Monobit",
      "num_failures": 1,
      "num_runs": 2000
    },
    {
      "name": "FIPS 140-2(2001-10-10) Poker",
      "num_failures": 0,
      "num_runs": 2000
    },
            .
            .
            .
    {
      "name": "FIPS 140-2(2001-10-10) Long run",
      "num_failures": 0,
      "num_runs": 2000
    },
    {
      "name": "FIPS 140-2(2001-10-10) Continuous run",
      "num_failures": 0,
      "num_runs": 2000
    }
  ]
}
```

Figure A.7: Example of output from *FIPS* battery.

A.5 BSI battery


```
{
  "sequence": <tested file>,
  "tests": [
    {
      "error": false,
      "name": "T0 - Words",
      "num_failures": 0,
      "num_runs": 127
    },
    {
      "error": false,
      "name": "T1 - Monobit",
      "num_failures": 0,
      "num_runs": 257
    },
    {
      "error": false,
      "name": "T2 - Poker",
      "num_failures": 0,
      "num_runs": 257
    },
    {
      "error": false,
      "name": "T6 - Uniform Distribution",
      "num_failures": 4000,
      "num_runs": 4000
    },
    {
      "error": false,
      "name": "T7 - Homogenity",
      "num_failures": 0,
      "num_runs": 83
    },
    {
      "error": false,
      "name": "T8 - Entropy",
      "num_failures": 0,
      "num_runs": 193
    }
  ]
}
```


Figure A.8: Example of output from *BSI* battery.

B Examples from testing toolkits

B.1 RTT settings

```
{
    "toolkit-settings": {
        "logger": {
            "dir-prefix": "results/logs",
            "run-log-dir": "run-logs",
            "dieharder-dir": "dieharder",
            "nist-sts-dir": "niststs",
            "tu01-smallcrush-dir": "testu01/smallcrush",
            "tu01-crush-dir": "testu01/crush",
            "tu01-bigcrush-dir": "testu01/bigcrush",
            "tu01-rabbit-dir": "testu01/rabbit",
            "tu01-alphabit-dir": "testu01/alphabit",
            "tu01-blockalphabit-dir": "testu01/blockalphabit"
        },

        "result-storage": {
            "file": {
                "main-file": "results/testbed-table.txt",
                "dir-prefix": "results/reports",
                "dieharder-dir": "dieharder",
                "nist-sts-dir": "niststs",
                "tu01-smallcrush-dir": "testu01/smallcrush",
                "tu01-crush-dir": "testu01/crush",
                "tu01-bigcrush-dir": "testu01/bigcrush",
                "tu01-rabbit-dir": "testu01/rabbit",
                "tu01-alphabit-dir": "testu01/alphabit",
                "tu01-blockalphabit-dir": "testu01/blockalphabit"
            },

            "mysql-db": {    // Database storage does not have to be filled,
                             // because it is optional functionality.
                "address": "",
                "name": "",
                "port": "",
                "credentials-file": ""
            }
        },

        "binaries": {        // Paths to executables of test batteries. Only
                             // this setting has to be filled in by the user.
            "nist-sts": "/rtt-statistical-batteries/nist-sts",
            "dieharder": "/rtt-statistical-batteries/dieharder",
            "testu01": "/rtt-statistical-batteries/testu01"
        },

        "miscellaneous": {
            "nist-sts": {
                "main-result-dir": "experiments/AlgorithmTesting/"
            }
        },

        "execution": {
            "max-parallel-tests": 8,
            "test-timeout-seconds": 40000
        }
    }
}
```

Figure B.1: General settings for *RTT* stored in *rtt-settings.json* file.

B.2 RTT battery configuration

```
{
  "randomness-testing-toolkit": {
    "dieharder-settings": {
      "defaults": {               // Default settings for all tests in the
        "test-ids": [             // battery including IDs of tests to be
          "0-4",                  // executed
          "200-204"
        ],
        "psamples": 100
      },
      "test-specific-settings": [  // List of tests with settings different
        {                          // from defaults or tests with variants
          "test-id": 2,
          "psamples": 81
        },
        {
          "test-id": 200,
          "variants": [            // List of all test variants to be executed
            {                      // for a given test
              "arguments": "-n 1",
              "psamples": 100
            },
            {
              "arguments": "-n 2",
              "psamples": 100
            }
          ]
        }
      ]
    },
    "nist-sts-settings": {         // Settings for next battery
        ...
    }
  }
}
```

Figure B.2: Example of battery configuration file for *RTT*.

B.3 Report from *RTT*

```
***** Randomness Testing Toolkit data stream analysis report *****
Date:    11-12-2023
File:    ../input/100MB.dat
Battery: Dieharder

Alpha:   0.01
Epsilon: 1e-08

Passed/Total tests: 21/25

Battery errors:

Battery warnings:
    Dieharder - Diehard OPERM5 Test (1) - variant 1: execution of test produced
error output.
    Dieharder - Diehard Birthdays Test (0) - variant 1: execution of test
produced error output.

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

    User settings:
        P-sample count: 2730
    ************

    Standard error output:
        # The file file_input_raw was rewound 3 times
    !!!!!!!!!!!!

    Kolmogorov-Smirnov statistic p-value: 0.25840105       Passed
    p-values:
        0.00056992 0.00206555 0.00239503 0.00239503 0.00341881
                   .
                   .
                   .
        0.99729956 0.99775183 0.99955111 0.99955111 0.99981347
    ============
------------------------------------------------------------

------------------------------------------------------------
Diehard OPERM5 Test test results:
    Result: Passed
    Test partial alpha: 0.01000000

    User settings:
        P-sample count: 104
    ************

    Standard error output:
        # The file file_input_raw was rewound 3 times
    !!!!!!!!!!!!

    Kolmogorov-Smirnov statistic p-value: 0.94228892       Passed
    p-values:
        0.01168130 0.03021779 0.06515499 0.06906779 0.07556090
                   .
                   .
                   .
        0.97121682 0.97877357 0.98985380 0.99694389
    ============
------------------------------------------------------------
```

Figure B.3: Example of report file from *RTT*.

B.4 Report from *rtt-py*

Input file: ../input/100MB.dat

Test 0: diehard_birthdays

ntuples	0
tsamples	100
psamples	2730
p-value	0.25840131

Test 1: diehard_operm5

ntuples	0
tsamples	1000000
psamples	104
p-value	0.94228891

Test 2: diehard_rank_32x32

ntuples	0
tsamples	40000
psamples	81
p-value	0.10185292

Figure B.4: Example of HTML report from *rtt-py*.

Results overview

	Failure rate	../input/100MB.dat
diehard_operm5_0 (DIEHARDER)	0.0	0.942289
diehard_birthdays_0 (DIEHARDER)	0.0	0.258401
rgb_lagged_sum_32 (DIEHARDER)	0.0	1.000000
sts_runs_0 (DIEHARDER)	0.0	0.116814
dab_dct_0 (DIEHARDER)	0.0	0.572732
marsaglia_tsang_gcd_0 (DIEHARDER)	0.0	0.323520
marsaglia_tsang_gcd_1 (DIEHARDER)	0.0	0.233607
diehard_3dsphere_0 (DIEHARDER)	0.0	0.752377
rgb_bitdist_0 (DIEHARDER)	0.0	0.132347
rgb_bitdist_1 (DIEHARDER)	0.0	0.329393
rgb_bitdist_2 (DIEHARDER)	0.0	0.234500
rgb_bitdist_3 (DIEHARDER)	0.0	0.457294
diehard_craps_0 (DIEHARDER)	0.0	0.563657
diehard_craps_1 (DIEHARDER)	1.0	0.000232
diehard_runs_0 (DIEHARDER)	0.0	0.433677
diehard_runs_1 (DIEHARDER)	0.0	0.020177
diehard_parking_lot_0 (DIEHARDER)	0.0	0.308323
diehard_squeeze_0 (DIEHARDER)	0.0	0.576888
sts_monobit_0 (DIEHARDER)	0.0	0.262109
diehard_2dsphere_0 (DIEHARDER)	0.0	0.498425

Figure B.5: Example of HTML overview table from *rtt-py*.

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. *Random Bit Generation | CSRC* [online]. [visited on 2023-11-10]. Available from: `https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software`.
4. *Empirical Testing of Random Number Generators* [online]. [visited on 2023-11-12]. Available from: `http://simul.iro.umontreal.ca/testu01/tu01.html`.
5. *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`.
6. *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`.
7. *pvavercak/rtt-py: Randomness testing toolkit in python* [online]. [visited on 2023-11-14]. Available from: `https://github.com/pvavercak/rtt-py`.
8. MOORE, David S.; NOTZ, William I. *The Basic Practice of Statistics*. Macmillan Learning, 2021. ISBN 1-319-38368-8.

9. 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.

10. 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.

11. D'AGOSTINO, Ralph B.; STEPHENS, Michael A. *Goodness-of-Fit-Techniques*. Routledge, 1986. ISBN 0-8247-8705-6.

12. SHESKIN, David J. Parametric and nonparametric statistical procedures. *Boca Raton: CRC*. 2000.

13. 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.

14. EDDELBUETTEL, Dirk. *eddelbuettel/dieharder* [online]. [visited on 2023-11-21]. Available from: `https://github.com/eddelbuettel/dieharder`.

15. SÝS, Marek; ŘÍHA, Zdeněk. Faster randomness testing with the NIST statistical test suite. In: *Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings 4*. Springer, 2014, pp. 272–284.

16. SÝS, Marek. *sysox/NIST-STS-optimised* [online]. [visited on 2023-11-21]. Available from: `https://github.com/sysox/NIST-STS-optimised`.

17. *rtt-statistical-batteries* [online]. [visited on 2023-11-10]. Available from: `https://github.com/crocs-muni/rtt-statistical-batteries`.

18. NIST. *SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES*. 2001. Available also from: `https://csrc.nist.gov/pubs/fips/140-2/upd2/final`.

19. VAVERČÁK, Patrik. *rtt-statistical-batteries* [online]. [visited on 2023-11-13]. Available from: https://github.com/pvavercak/rtt-statistical-batteries.

20. 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.

21. KILLMANN, Wolfgang; SCHINDLER, Werner. *A proposal for: Functionality classes and evaluation methodology for true (physical) random number generators*. 2001. Available also from: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_Functionality_classes_evaluation_methodology_for_true_RNG_e.html.

22. 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.