Masaryk University
Faculty of Informatics

The automated testing of randomness with multiple statistical batteries

Master's Thesis

Ľubomír Obrátil

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

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.

Ľubomír Obrátil

Advisor: RNDr. Petr Švenda, Ph.D.

Acknowledgement

TODO

Abstract

The aim of this thesis was to design and develop a unified interface for accessing randomness testing software and use it to perform a large scale randomness evaluation and comparison of used software. The software tools supported by the interface are NIST Statistical Testing Suite, Dieharder and TestU01. Results of each tool are presented and interpreted in a consistent way. Using the developed interface, called Randomness Testing Toolkit, the randomness of outputs of 15 distinct cryptographic primitives was evaluated. The set of tested primitives contains were well-known algorithms such as AES and finalists of competitions eSTREAM and SHA-3. Based on the evaluation, the security margins of the algorithms with respect to statistical analysis were defined. In further research, the validity of outputs of Dieharder was assessed by analysing a significant amount of truly random data.

Keywords

randomness testing, statistical test batteries, cryptographic algorithm evaluation, RTT, NIST STS, Dieharder, TestU01, EACirc, CRoCS

Contents

1 Introduction

In modern cryptography, one of the required properties of cryptographic primitives like block ciphers or hash functions is the pseudo-randomness of their output. The algorithms lacking this attribute can leak information about the secret key or the plaintext, which is not desirable; thus the need for the randomness testing arose.

Since the advent of cryptography, multiple tools addressing this problem were developed; most of them evaluate statistical properties of the tested data. In this thesis, we closely work with three statistical tools: NIST Statistical Testing Suite [1], Dieharder [2] and TestU01 [3]. Other notable tools are Crypt-X [4] and Diehard [5], which pioneered the statistical randomness evaluation, and EACirc [6] that employs alternative, adaptive approach.

The aim of this thesis is to design and develop a unifying interface for the statistical tools. The interface would allow its users to test the randomness of arbitrary binary data in an easy and user-friendly manner. The interface would also present and interpret the results of the randomness assessment in a consistent way.

We named the interface Randomness Testing Toolkit (RTT) [7] and described its functionality and implementation details in Chapter 3. The overview of the software used in RTT is in Chapter 2.

In Chapter 4, we outline the experiments done with the RTT interface. In our first experiment we analysed large quantities of random data to establish a baseline, based on which we would interpret our further analyses. For the next experiment, we analysed the randomness of outputs of 15 well-known cryptographic algorithms[1] such as AES or Keccak. We compared the results of the experiment to the results previously obtained with EACirc and compared viability of the different approaches. Based on the outcomes, we derived the security margins of the tested algorithms with respect to the statistical analysis.

Chapter 5 contains details about our inspection of the validity of the Dieharder battery. To evaluate the validity, we processed a big amount of random data with the battery and analysed the outputs of the processing. We conclude the thesis with the summary in Chapter 6. In the chapter, we briefly list the achieved results and propose possible directions for future research and development.

In the thesis text, the plural is used, despite the fact that the presented work was done mostly by myself. Randomness Testing Toolkit was developed as a part of a greater effort at Centre for Research on Cryptography and Security[2], Masaryk University; related problems, ideas and solutions were oftentimes discussed with other researchers[3]. Research done by others is properly attributed when presented.

1. We refer to both ciphers and hashing functions simply as to cryptographic algorithms or primitives.
2. Simply referred to as CRoCS.
3. Notable people involved in randomness research and development are Radka Cieslarová, Michal Hajas, Dušan Klinec, Karel Kubíček, Matúš Nemec, Jiří Novotný, Marek Sýs, Petr Švenda, Martin Ukrop and others.

2 Used third-party statistical software

In this chapter, we will explain terminology specific to Randomness Testing Toolkit, present a quick overview of the statistical software used in the toolkit and describe the observed and unexpected behavior of the statistical software in edge cases. We also list undertaken measures to mitigate the undocumented behaviour in our further experiments.

2.1 Terminology

Throughout the thesis, we are using certain expressions in the context of Randomness Testing Toolkit and the tools and math it uses. We list these terms along with their explanations here.

(Statistical) Battery
A program developed by a third party serving as a tool for evaluation of randomness of arbitrary binary data. A statistical battery usually contains one or more statistical tests. The final result of the assessment is based on the results of the tests. Examples of statistical batteries are NIST Statistical Test Suite, Dieharder or TestU01.

(Statistical) Test
A single unit in statistical battery that measures some property of the tested binary stream (e.g. number of zeroes). The test can have multiple variants and subtests, and the result of the test is one or multiple statistics.

Null hypothesis - H_0
The hypothesis H_0 denotes the hypothesis that the tested data stream is random. Based on the results of the test, we can either reject H_0 and say that the data are not random or not reject it. In the latter case, we assume that the tested data are indeed random.

P-value
In our hypothesis testing, a p-value is a probability of obtaining equal or more extreme test result while H_0 holds true. In our situation, a p-value denotes the probability that we would get same or more extreme test results when testing truly random data. Therefore, the closer the p-value is to 0 the less is the probability the tested data is random and vice versa.

Alpha - α
Significance level based on which we either reject or not reject H_0. We can specify some interval (e.g. $(0, \alpha]$) and if the result of the test (p-value) falls into this interval, we will reject the hypothesis that the tested data is random. Alternatively, we can also reject p-values that are too extreme on both sides of the interval (outside of $\left[\frac{\alpha}{2}, 1 - \frac{\alpha}{2}\right]$).

Type I and type II errors
A type I error (false positive) is the rejection of a true null hypothesis H_0 [8]. On the opposite, a type II error (false negative) is incorrectly holding to the null hypothesis, even when it should have been rejected.

Statistic
The value obtained by certain calculation from first level p-values. Multiple statistics can

be obtained from one set of p-values e.g. when testing the set for uniformity we can use Kolmogorov-Smirnov Test or Chi-Square test. Based on values of statistics of a test we can decide rejection of H_0.

Test sample
A single execution of a statistical test. The result of single test execution is one first level p-value. By repeating execution of the test, we obtain multiple p-values. By using a certain statistic (e.g. Kolmogorov-Smirnov test), we can calculate a single second level p-value.

Variant of a test
Many tests can be parametrized in some ways, possibly giving different results with the same input data. We don't treat multiple executions of a single test with different settings as separate units but rather as variants of that test.

Subtest
Some tests, even when executed only once, may measure multiple properties of the data thus providing multiple results. For example, Serial Test from Dieharder battery will measure frequencies of all 1, 2, .., 16-bit patterns in the data. We treat these measurements separately - as subtests of the test. The subtests can have multiple separate statistics.

Job
Single execution of Randomness Testing Toolkit that will analyse single binary data file with single statistical battery.

Experiment
Batch of jobs that analysed single data file with various statistical batteries and settings.

2.2 Batteries supported by RTT

In this section we provide a brief overview of the supported statistical batteries included in Randomness Testing Toolkit.

2.2.1 NIST Statistical Test Suite

The battery of statistical tests was developed by National Institute of Standards and Technology [1]. The battery implements 15 statistical tests for evaluating randomness of input data. The tests are listed in Appendix B.

The reference implementation described in [9] is not used in RTT because it is considerably slower than its optimized counterparts. The faster implementation called NIST STS optimised v5.0.2 used in RTT was developed by Marek Sýs and Zdeněk Říha [10].

2.2.2 Dieharder

Dieharder is a battery designed by Robert G. Brown at the Duke University [2]. The battery features user-friendly console interface with the possibility of fine-grain modification of the test parameters. The fact that Dieharder is included in repositories of some Linux distributions [11] adds to its popularity and ease of use. Dieharder includes all tests from the older statistical battery Diehard [5], three tests from NIST STS and several other tests

implemented by the author. The tests, along with their default settings, are listed in Appendix C.

Since the original Dieharder implementation doesn't output all of the information we needed for interpretation and evaluation of the results, we had to modify the source code of the battery. RTT uses this modified Dieharder. We modified Dieharder v3.31.1 which is the last stable maintained release on the project page.

2.2.3 TestU01

This library of statistical batteries was developed at Université de Montréal by Pierre L'Ecuyer et al. [3]. It contains a wide range of tests from NIST STS, Dieharder, and literature. It also implements various pseudo-random number generators. The statistical tests are grouped into multiple categories each intended for different use-case scenario. We will treat these categories of tests as separate batteries. TestU01 includes following ten batteries. The tests included in specific batteries are listed in TestU01 documentation [12].

Small Crush, Crush, Big Crush
Small Crush battery is very fast and needs a relatively small amount of data to run - around 250 megabytes. Small Crush is also the only battery that can be natively used for analysis of data in a binary file, for the use of Crush and Big Crush, the user has to implement PRNG with the TestU01 interface. Crush and Big Crush batteries are more stringent and need gigabytes of data and a few hours to finish while Big Crush is more time and data demanding.

Rabbit, Alphabit, Block Alphabit
Tests in these batteries are suited for testing hardware bit generators and can be applied to an arbitrary amount of data. Data can be provided either as a binary file or PRNG implementing the TestU01 interface.

PseudoDIEHARD, FIPS_140_2
Tests in PseudoDIEHARD imitate DIEHARD battery; FIPS_140_2 battery implements a small suite of tests defined in NIST FIPS 140-2 standard [13] (the definition of tests was removed from the standard in Change Notice 2). Randomness Testing Toolkit doesn't support these two batteries since they are subsets of other batteries.

Since TestU01 v1.2.3 is available only as an ANSI C library, we developed a console interface for it. The interface implements a dummy number generator that provides data from a supplied binary file to the battery. Our interface allows us to apply batteries to arbitrary binary data even when the batteries themselves don't support this feature natively.

2.3 Unexpected behavior and errors of the batteries

To examine the boundary behavior of the above-listed tools, we used them to process extremely non-random data streams. The data streams that we used as the input to the batteries were two binary data files consisting of only zeroes and ones respectively. The settings of the batteries remained set to default.

Below we list observed undocumented behavior that differs from the execution of the batteries with non-extreme input.

NIST Statistical Test Suite

Each test in the battery processed 1000 separate data streams. Each data stream was 1000000 bits long.

Tests Random Excursions and Random Excursions Variant are not applicable to all possible data streams. In a regular run with reasonably random data, this doesn't matter much, as the tests are repeated multiple times, and the final result will simply be calculated from a lesser number of p-values.

Neither of the tests can be applied to a stream full of zeroes or ones. This causes absence of results when analyzing such data. The user can find out the fact that the tests are not applicable to provided stream after he inspects logs of the program; otherwise, the interpretation of the missing results is left to him.

Dieharder

When processing extremely non-random data with Dieharder, we observed various erroneous events. The events are summarized in Table 2.1.

Test name	Stream of zeroes	Stream of ones
STS Runs	No result	No result
DAB Monobit 2	Invalid result	Invalid result
Diehard Minimum Distance (2D Circle)	Stuck execution	–
Diehard 3D Sphere (Minimum Distance)	Stuck execution	–
Marsaglia and Tsang GCD	Stuck execution	–
RGB Generalized Minimum Distance	Stuck execution	–
RGB Kolmogorov-Smirnov	Stuck execution	–
Diehard Craps	–	Stuck execution
RGB Bit Distribution	–	Stuck execution

Table 2.1: Undocumented behavior of tests in Dieharder battery

Observed errors

- **No result** Test didn't provide any p-values that would be used to calculate the final result of the test. The user is not notified of this, and the final result of the test statistic is based on default value (1.0).
- **Invalid result** Test provided resulting statistic and there was no indication of error other than that the result was again default value of the statistic (1.0). Following the definition of p-value, the interpretation of such result is that the analyzed stream was almost certainly random. This is obviously not true, as both streams are just repeating ones or zeroes respectively.

- **Stuck execution** Tests froze at a certain point in execution, did not produce any results and we were forced to kill the processes manually.

TestU01

Batteries Small Crush, Crush, Big Crush, Rabbit, Alphabit and Block Alphabit were executed. Tests that are part of multiple batteries acted in the same way across the batteries. The behavior is summarized in Table 2.2.

Test name	Stream of zeroes	Stream of ones
sstring_Run	No result	No result
sknuth_Gap	No result	–
svaria_SampleProd	All results invalid	All results invalid
svaria_AppearenceSpacings	All results invalid	All results invalid
scomp_LinearComp	All results invalid	All results invalid
scomp_LempelZiv	All results invalid	All results invalid
svaria_SampleCorr	–	All results invalid
sknuth_MaxOft	Some results invalid	Some results invalid
svaria_SampleMean	Some results invalid	Some results invalid
sspectral_Fourier3	Some results invalid	Some results invalid
sstring_HammingWeight2	Some results invalid	Some results invalid
sstring_AutoCor	Some results invalid	Some results invalid
smultin_MultinomialBitsOver	Some results invalid	Some results invalid
sstring_LongestHeadRun	–	Some results invalid
snpair_ClosePairs	Stuck execution	Stuck execution
snpair_ClosePairsBitMatch	–	Stuck execution
svaria_SumCollector	–	Stuck execution
smarsa_GCD	–	Stuck execution

Table 2.2: Undocumented behavior of tests in TestU01 library

Observed errors

- **No results** Tests reported warning and ended without any result. This is probably caused by the tests not being applicable to provided data.
- **All results invalid** All statistics of the test reported p-value very close to 1.0. This could lead the user to the interpretation that the test reports the data as an almost perfect random stream.
- **Some results invalid** Similar situation to the previous one but not all statistics of the test are close to 1.0. Results of tests statistics are either close to 0.0 or 1.0.
- **Stuck execution** Tests froze at a certain point of execution. In some cases, this is preceded by an issued warning. Tests didn't produce any results and had to be killed manually.

2.3.1 Preventive measures in RTT

Since we need to use the batteries in RTT with arbitrary binary data, we implemented following measures that mitigate above-mentioned errors in our experiments.

- Tests that don't produce any results are ignored and treated as if never executed.
- Because some tests give 1.0 as a result of their statistics when the data are clearly not random, we will reject the hypothesis of randomness either when the p-value is too close to 0 or is equal to 1. More specifically, H_0 will be rejected for all p-values that falls outside of the interval $[\alpha, 1)$.
- Each test is executed with the timeout. If the test doesn't finish within defined time limit, we will automatically terminate it and then treat it as if it didn't produce any results.

3 Randomness Testing Toolkit

One of our goals during working on this project was to create a tool that would provide fast and user-friendly analysis of randomness even for users not experienced in the field of statistical randomness testing. Process of statistical testing generally includes finding a suitable tool, installation of said tool, and an execution and interpretaion of the analysis results.

Another motivation was that the developed tool would unite the process of randomness analysis that is often done by researchers in CRoCS. Before the start of the project, each researcher used his own set of scripts and tools to analyse, which is difficult to manage and also complicates comparation of the experiments done by different people.

Therefore we developed Randomness Testing Toolkit [7], a tool that serves as an unified interface of the three statistical batteries presented in Chapter 2. The toolkit consists of several connected parts that are intended for various purposes. Apart from interfaces and data storage format, the parts are independent and can be exchanged or modified freely.

In the following sections of this chapter, we will provide detailed description of the implemented parts of the toolkit.

3.1 Local unified interface

The local unified interface is at the lowest level of the developed parts of RTT. The interface is a binary executable that accepts settings and outputs the results in common format for all batteries. The executable handles transformation of general settings into the battery specific ones, running the battery executable and then transforming gathered output into general format. The process is visualized in Figure 3.1.

The toolkit accepts settings needed for its run from three following sources:

- **Command-line arguments** – Basic settings that can change in every run; e.g. path to input file with binary data or to file with battery configuration.
- **Toolkit settings file** – General settings for the toolkit related to the system environment e.g. paths to executables of the statistical batteries. In most of the cases, the settings stay the same for all executions of the toolkit regardless of the battery or data used. Changes in this configuration are needed only when the environment changes.
- **Battery configuration file** – Configuration file that fully configures input settings for single or multiple batteries. The configuration of the batteries can vary among the executions of the toolkit.

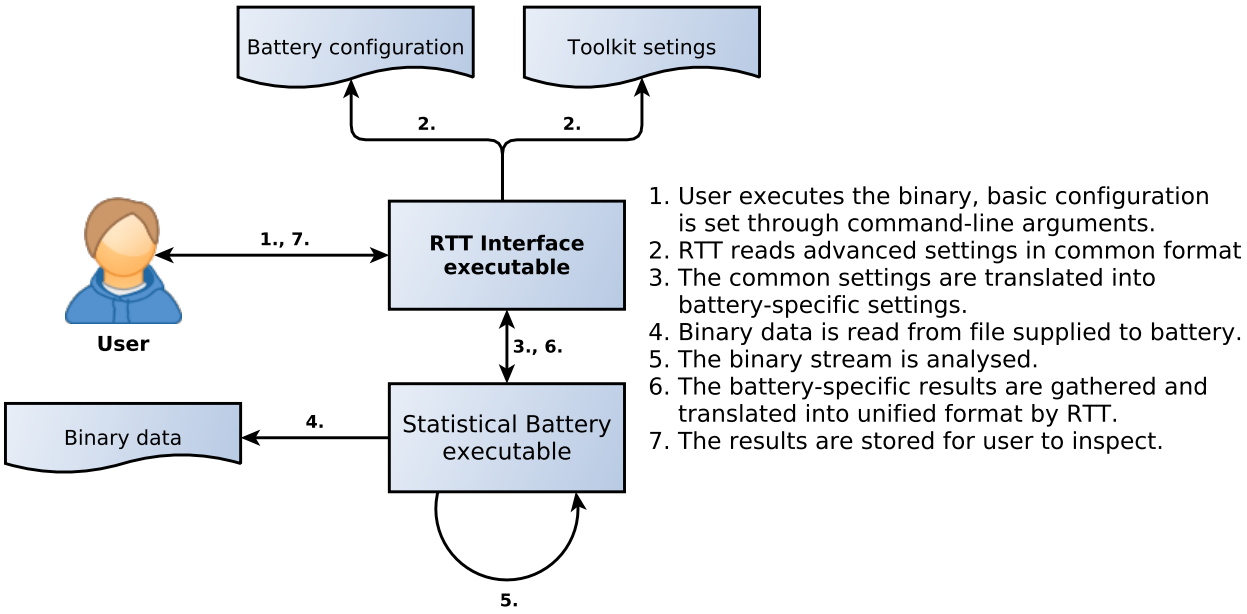

Figure 3.1: Local RTT workflow

3.1.1 Command-line arguments

The basic options of the RTT execution are set through command-line arguments. The executable recognizes following options.

- `-h` Usage of the toolkit will be printed.
- `-b <battery>` Sets the battery that will be executed during the run. Values accepted as `<battery>`: dieharder, nist_sts, tu01_smallcrush, tu01_crush, tu01_bigcrush, tu01_rabbit, tu01_alphabit and tu01_blockalphabit.
- `-t <test-id>` Optional argument. If set, only single test with id `<test-id>` will be executed in the chosen battery.
- `-c <config-path>` Path to file with configuration of the battery. Details about battery configuration are in Section 3.1.3.
- `-f <data-path>` Path to the file with the binary data that will be analyzed by the chosen battery.
- `-r <result-storage>` Sets the type of the result storage that will be used. Accepted values are `file-report` or `db-mysql`. The option doesn't have to be set; `file-report` is then used as default option. More information about result storages is in Section 3.2.
- `-eid <experiment-id>` Option is mandatory only when `db-mysql` is set as a result storage; the option is ignored otherwise. Sets ID of the experiment in the database that will be assigned the results of the toolkit execution.

3.1.2 Toolkit settings

The general settings of the toolkit are kept in separate file `rtt-settings.json`. The settings are related to the environment in which the toolkit will be executed – locations of the log files and executables, result storage options and toolkit execution options

The file is created by the user after the toolkit installation and stays the same for all of the subsequent executions of the program unless the environment changes (as opposed to input file with configuration of the batteries that can be different for each data analysis). The file `rtt-settings.json` must be located in the working directory of the executable. Basic structure of the file is in Figure 3.2.

```
{
  "toolkit-settings": {
    "logger": {
      "comment": "Program logger settings",
      "option": "value", ...
    },
    "result-storage": {
      "file": {
        "comment": "File result storage settings",
        "option": "value", ...
      },
      "mysql-db": {
        "comment": "MySQL Database storage settings",
        "option": "value", ...
      }
    },
    "binaries": {
      "comment": "Executable binaries locations",
      "option": "value", ...
    },
    "miscelaneous": {
      "nist-sts": {
        "comment": "Miscellaneous NIST STS settings",
        "option": "value", ...
      }
    },
    "execution": {
      "comment": "Battery execution settings",
      "option": "value", ...
    }
  }
}
```

Figure 3.2: Basic structure of `rtt-settings.json`

The individual tags and their descriptions are listed here. For detailed overview of the options that can be set within individual tags see Appendix D; for a complete example see Appendix E.

toolkit-settings/logger
Tag containing settings related to the location of log files produced during runtime.
Settings: `dir-prefix`, `run-log-dir`, `<battery>-dir`

toolkit-settings/result-storage/file
Section with settings needed for file result storage. For details about the storage see Section 3.2.1.
Settings: `main-file`, `dir-prefix`, `<battery>-dir`

toolkit-settings/result-storage/mysql-db *Optional*
Section with settings needed for MySQL Database storage. For details about the storage see Section 3.2.2.
Settings: `address`, `port`, `name`, `credentials-file`

toolkit-settings/binaries
Section with the locations of the executables of the batteries.
Settings: `<battery>`

toolkit-settings/miscelaneous/nist-sts
Miscellaneous constant settings related to NIST STS battery.
Settings: `main-result-dir`

toolkit-settings/execution
Settings related to the execution of the statistical batteries.
Settings: `max-parallel-tests`, `test-timeout-seconds`

3.1.3 Battery configuration

The file with battery configuration defines all settings for the battery and the tests that will be executed during the program run. Each execution of RTT can have different battery configuration; the path to file with the configuration is set through command-line argument `-c`. The basic structure of the configuration file is outlined in Figure 3.3.

A single configuration file can contain settings for multiple batteries. This allows the user to have only one configuration for all of his experiments. In each battery-specific section, the user can set the default values of settings for all the tests in the battery. Additionaly, settings specific only to certain tests or test variants can be also defined.

The complete guide to all common and battery specific settings is in Appendix F; for example of possible battery configuration see Appendix G.

3.2 Storage for the analysis results

The result storages are modules in RTT interface that handle saving of the execution results for later inspection by the user. Based on the user's settings, the results can be saved in one of the two following ways.

```
{
  "randomness-testing-toolkit": {
    "<battery>-settings": {
      "comment": "Section with <battery> settings",
      "defaults": {
        "comment": "Default settings in <battery>",
        "option": "value", ...
      },
      "test-specific-settings": [
        {
          "test-id": "<id>",
          "comment": "Settings specific to single test <id>",
          "option": "value", ...
          "variants": [
            {
              "comment": "Settings of the first "
                         "variant of test <id>",
              "option": "value", ...
            },
            {
              "comment": "Settings of the second "
                         "variant of test <id>",
              "option": "value", ...
            }
          ]
        }
      ]
    }
  }
}
```

Figure 3.3: Basic structure of battery configuration file

The file storage is intended to be used when RTT is executed locally and manually by the user. After the execution, the user can inspect the human-readable reports that were generated by the module.

The MySQL database storage is used when RTT is deployed as a service on remote server. The results are stored in the database and can be viewed through web interface. It is also possible for user to install MySQL server on his local machine and setup the module so that it will use this database. The user then can work with the stored results as he sees fit.

The user can also implement his own version of the result storage module that will output the results in arbitrary format.

3.2.1 File storage

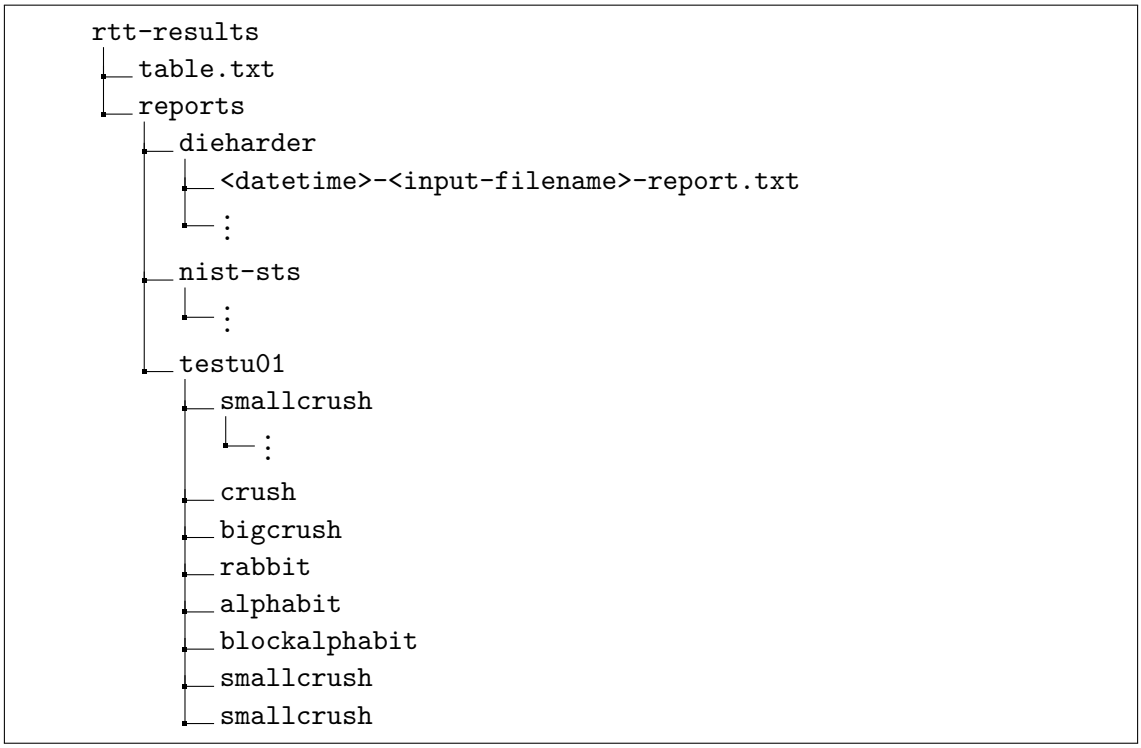

Figure 3.4: Generated files following settings from Appendix E

The file storage will store results into human-readable reports. Each execution of the RTT interface will generate single battery-specific report as shown in Figure 3.4. The newly generated files will have names based on the time of the execution and the filename of the binary data that was analysed. Possible instance of a report file is shown in Appendix H.

The main result file (in this case `table.txt`) contains summary of the result of all previous RTT executions since the last deletion of the file. Therefore, if the file does not exist in time of the execution it will be generated; the file will be only modified otherwise.

In the file, there is a simple ASCII table that contains single row for each input data file with distinct name that was analyzed. The rows contain name of the input file, time of the last modification of the row and a proportion of passed and total number of tests for each battery. In case that a single file is analysed multiple times by a single battery, the results are overwritten.

3.2.2 MySQL database storage

When using the database storage, RTT writes the results of the run directly into MySQL database. Layout of the database is shown in Figure 3.5. The tables `experiments` and `jobs` are not strictly required by the toolkit as the results are written only into table `batteries` and its sub-tables. The additional tables were added because we anticipated deployment of RTT on multiple servers and the tables store needed metadata about the execution and scheduling.

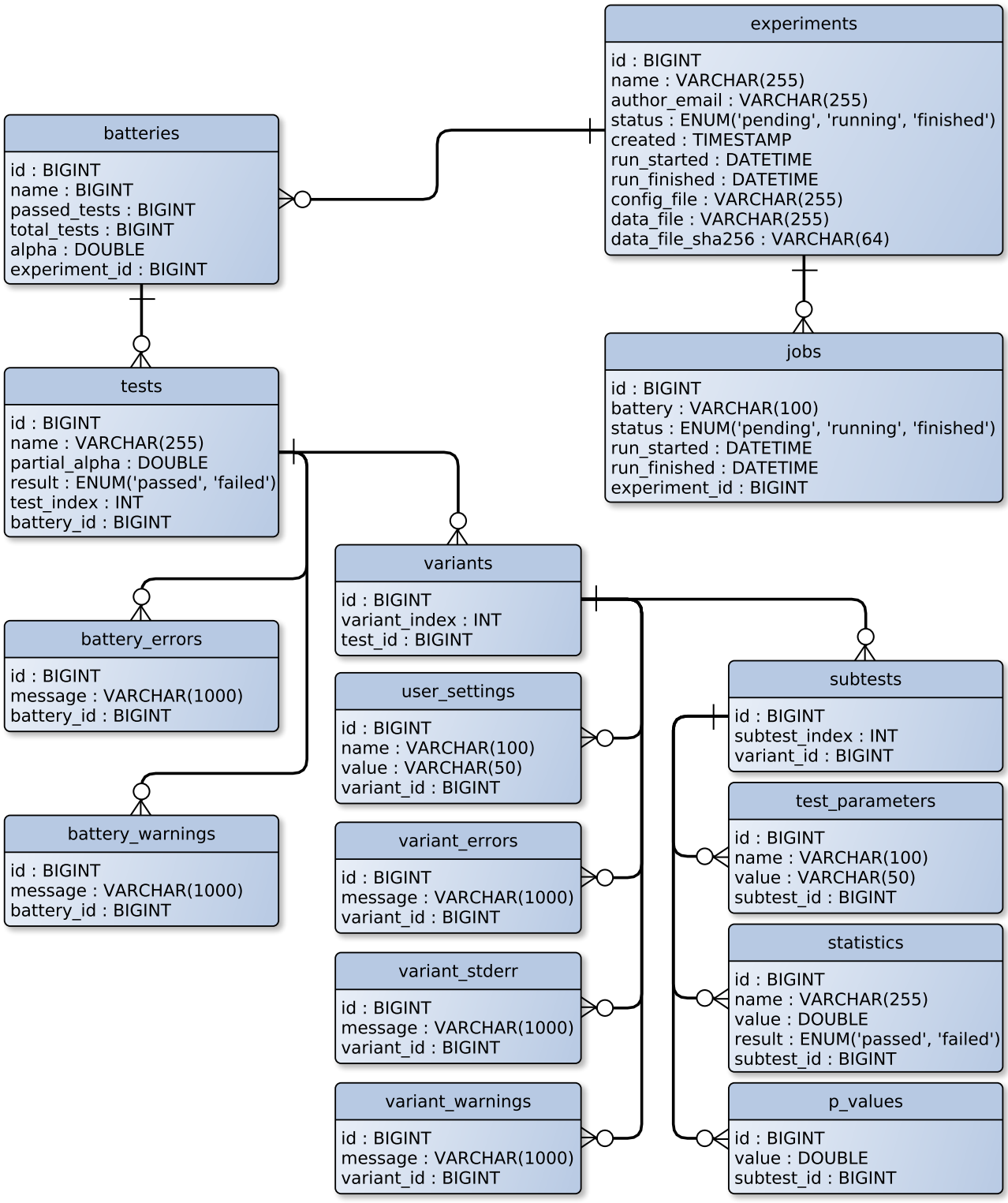

Figure 3.5: ERD Model of the database

`experiments`
We treat multiple analyses of single data stream by various batteries as a single experiment. The table stores metadata about the experiment such as name of the analysed file or used configuration file.

`jobs`
The table is used for scheduling execution of RTT on remote server. It holds information about the batteries that will be executed and experiment to which they belong. Single job represents a single execution of RTT.

`batteries`
Each row represents results of a single execution of RTT. The analysis results are written to this table and its sub-tables by the database storage. The results of multiple RTT executions can be assigned to a single experiment.

`battery_errors`, `battery_warnings`
Stores information about errors or warnings that happened during RTT runtime.

`tests`
Contains information about the tests that were executed.

`variants`
Information about the variants of the tests that were executed.

`variant_errors`, `variant_stderr`, `variant_warnings`
The errors, warnings and standard error output that were extracted from the outputs of the test variants are stored in the tables.

`user_settings`
Settings that were set by the user in the battery configuration file.

`subtests`
Table with the subtests that were executed.

`test_parameters`
Options and parameters that were extracted from the outputs of the subtests.

`statistics`
List of all extracted statistics and their results.

`p_values`
List of all extracted p-values.

3.3 Remote service

Randomness Testing Toolkit can be deployed on a single or multiple servers as a service, allowing the users to perform the binary data analysis without the need for the installation of the toolkit and without execution of the tool on their local machine. The statistical testing is, in certain configurations, demanding on computational time and resources. Having the RTT installed on remote machines allows us to scale the resources available for the toolkit and further speed-up the analysis. The user only needs to provide the data for the toolkit and choose (or provide his own) configuration for the batteries. After the testing, the user is notified and can inspect the results.

In the deployment part of the project, we developed an utility project that handles setup and installation of the local interface on single or multiple machines. The deployment project also handles setup of auxiliary scripts and tools that are required for result database and data storage and that will be used by the machines used for statistical testing computation. The database and storage can be deployed on separate machines or on single server that will handle all of the tasks along with the computation of the analysis.

The interaction of the user with the infrastructure is visualized in Figure 3.6. In following section we outline the different purposes of the servers.

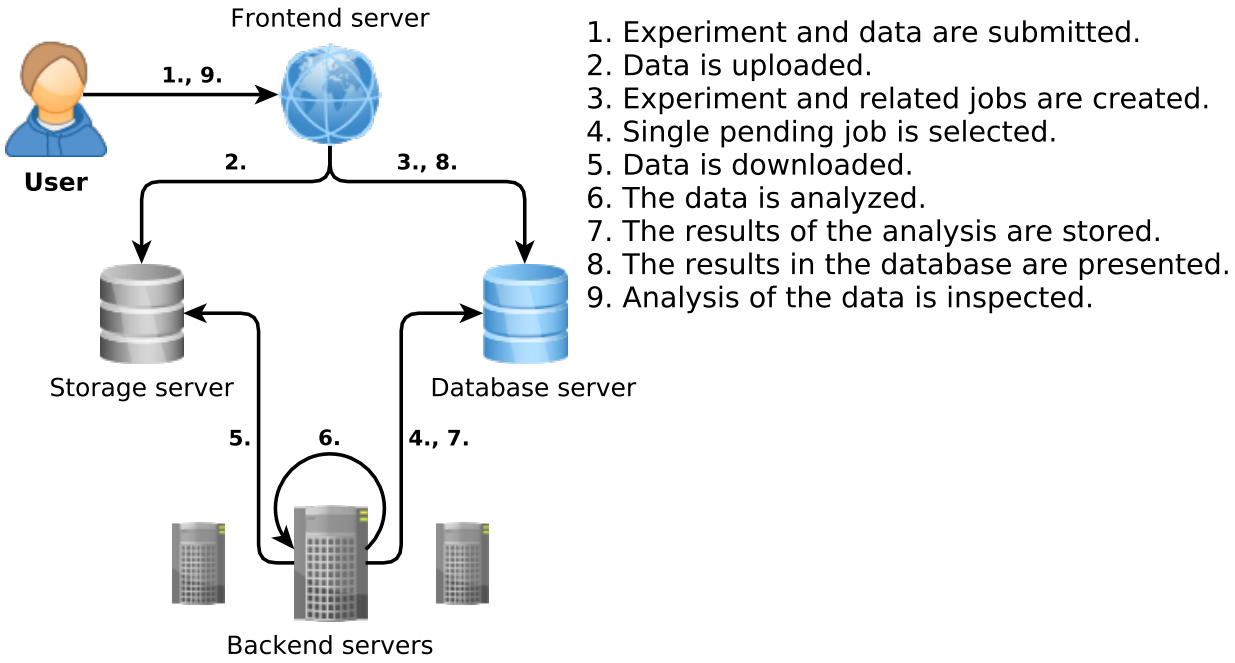

Figure 3.6: Interaction between user and servers

3.3.1 Database server

The machine hosting MySQL server with created database for RTT as shown in Figure 3.5. The database is used for storing the results of the analyses as well as the information used for scheduling and distribution of jobs to backend servers.

3.3.2 Storage server

The server hosts a storage space that acts as a temporary deposit for the data that is not yet distributed and analysed on the backend servers. After the distribution and analysis, the data is removed from the machine.

Scripts on the server

- `clean_cache.py` – Script that handles removal of the data files that were already analysed. The script is executed periodically.

3.3.3 Backend server

Single or multiple machines that host the installed toolkit and statistical batteries. The analysis of the data is executed on these machines. If there are multiple backend servers, the computations are scheduled based on the information stored in the database. Prior to the analysis, the data is downloaded from the storage and stored in local cache. After the computation, the results are stored on the database server and the data is deleted.

Both jobs and experiments can be in one of the three states – `pending`, `running` and `finished`; the descriptions of the states are in Table 3.7. Scheduling of the computation is based on the state of the experiments and jobs.

Status	Job	Experiment
`pending`	The job is waiting for execution.	All of the jobs related to the experiment are waiting for execution.
`running`	The job is being executed on backend server.	Some or all related jobs are being executed.
`finished`	The jobs execution ended.	The execution of all related jobs ended.

Table 3.7: States of jobs and experiments

Scripts on the server

- `clean_cache.py` – Script that handles removal of the locally cached data files that were already analysed. The script is executed periodically.
- `run_jobs.py` – Script responsible for choosing the correct computational job, downloading the data from the storage server and the execution of the analysis. Th script is executed periodically. The jobs are picked from the database table `jobs` with following priority:
  1. Jobs of those experiments whose data was downloaded earlier and the data is present in the local cache of the server.
  2. Any jobs of the experiments that are in pending state.
  3. Any jobs that are in pending state.

  The rules are designed in such a way that a minimal amount of data is transferred over the network. If there are enough distinct pending experiments for all servers then each server will pick single experiment and execute all its jobs and the data of the experiment is transferred only once.

3.3.4 Frontend server

The user interacts only with the frontend server. The machine accepts data and requests for creating an experiment. After the request, the server creates the experiment and the related jobs in the database and uploads the data into the storage. The server also presents the results from the database to the users.

The server can be directly accessed through SSH. The direct access is meant for experienced users that need to analyze big volumes of data. Additionaly, an optional web interface can be deployed on the frontend server. The web interface can be used for interactive browsing of the analysis results as well as for submitting undemanding amount of data and experiments by the users.

The direct access to the server is granted to the user by the system administrator. The RTT users can only access part of the system that is separated from the main system using `chroot jail` [14]. The isolated environment contains only the tools and software that is necessary for submitting and processing the experiments. The data for the testing can be uploaded to the server from user's local machine, downloaded from other network location using `sftp` or generated directly on the server. The data generation is realized with help of the generator tool [15] that was developed at CRoCS and is capable of producing outputs of numerous round-reduced cryptographic primitives.

Experiment submission utility

To create a new experiment, the user has to use utility `submit_experiment` that is installed on the server. The utility will transfer the user's data into the storage and will insert new computational jobs into the database. The utility accepts following command-line arguments.

- `-h, --help` Prints the usage of the utility.
- `-n, --name <name>` Sets the name of the experiment.
- `-e, --email <address>` Optional argument. When set, an email with brief results of the analysis will be sent to `<address>` after the end of the computation.
- `-c, --cfg <config-path>` Path to file with configuration of the batteries that will be included in the experiment. Details about configuring the batteries are in Section 3.1.3.
- `-f, --file <data-path>` Path to file with binary data that will be analysed by the statistical batteries included in the experiment.
- `-a, --all_batteries` Flag argument that will include all of the defined batteries except TestU01 Big Crush in the experiment.
- `--nist_sts` Switch for NIST Statistical Test Suite
- `--dieharder` Switch for Dieharder
- `--tu01_smallcrush` Switch for TestU01 Small Crush
- `--tu01_crush` Switch for TestU01 Crush
- `--tu01_bigcrush` Switch for TestU01 Big Crush
- `--tu01_rabbit` Switch for TestU01 Rabbit
- `--tu01_alphabit` Switch for TestU01 Alphabit
- `--tu01_blockalphabit` Switch for TestU01 Block Alphabit

The flags for the batteries switch their inclusion in the experiment. For example, if we use arguments `"--all_batteries --dieharder"`, then the batteries included in the experiment will be the batteries added by the `"--all_batteries"` flag **except** the Dieharder battery. However, if we would use only argument `"--dieharder"`, then Dieharder will be the **only** battery used is the experiment.

3.3.5 Web Interface for RTT

The web interface is aiming to be useful in the basic use-case, where the user has some binary data in a single file that he wants to analyse. The user will upload the data through the website and will wait for the end of the analysis; the results will be shown on the website after the computation. The web interface is not dependent on or required by the underlying server infrastructure that executes analysis of the data. It was developed only to serve as a user-friendly access point to Randomness Testing Toolkit. For development, we used Django (cit) framework for its ease of use.

In most of the cases, one does not have to create new configuration for the batteries that will be executed during the analysis; a suitable configuration will be chosen for him automatically. Alternatively one can also manually choose the desired configuration or create his own configuration from the scratch according to his preferences.

The limitation of the interface is that the manual creation of large number of experiments would be too time-consuming and error-prone for the user, as the process can't be trivially automated. However the web interface is not intended to be used in such situations; creating the experiments through direct access to the frontend server would be much more viable.

User roles

We separate users into three categories in the context of the web interface application. It is worth noting that the user accounts existing in the web interface are independent of the accounts existing on the frontend server. In the following list we summarize the roles and their privileges.

- **Anonymous user privileges**
  - Browse results of all previous experiments.
  - Create an experiment if an unique access link was shared with the user.
- **Authenticated user privileges**
  - Create experiments.
  - Change details and password of the account.
- **Administrator privileges**
  - Creating new and modifying existing users.
  - Adding administrator privileges to user accounts.
  - Adding or modifying predefined battery configurations.
  - Adding or modifying unique access links to the experiment submission. Each access link has its own expiration date and is unusable after this date.

Experiment submission

To create new experiment the user will have to fill-out following settings in the online form. The screenshot of the form is included in the Appendix I.

- **Experiment name** – Name identifying the experiment, doesn't have to be unique.
- **E-Mail** – Required only when the user is not logged in. After the computation, e-mail will be sent to the address, notifying the user about the results.
- **Binary data file** – Data that will be analysed; uploaded from the user's machine.
- **Configuration** – User can use default battery configuration, choose one of the predefined configurations or upload his own configuration file. The battery configuration files are described in Section 3.1.3.
- **Battery application** – Set of switches that defines which batteries will be included in the experiment.

Result presentation

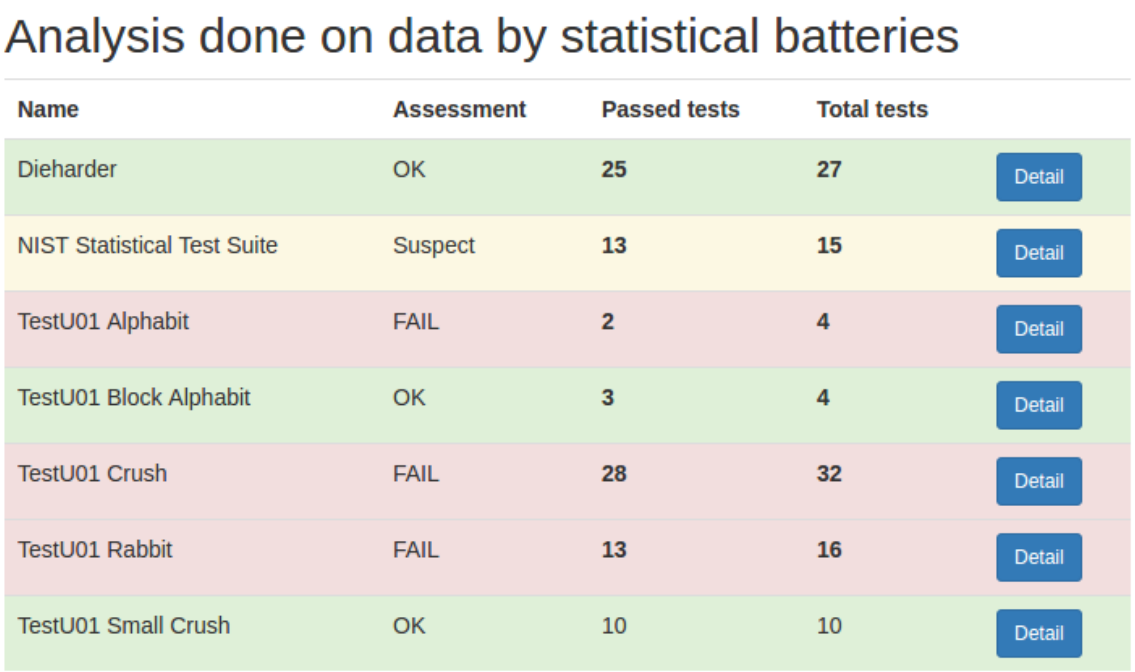

Figure 3.8: Experiment evaluation

The web interface also presents and assesses the results of the past experiments of all users. Each experiment is listed along with the batteries that were included in it. Battery results are assessed separately; the assessment is based on the proportion of passed and total number of tests included in the battery. We denote this proportion as x. Then the probability $P(x)$ of achieving such proportion when testing truly random data is calculated. The calculation process is described in Section 4.2. The example of a experiment evalution is shown in Figure 3.8. Based on the probability, the possible assessments are as follows.

- **FAIL** ($0 < P(x) < 0.001$) – The result is deemed as a failure and the source of the data should not be considered as a quality randomness generator.
- **Suspect** ($0.001 \leq P(x) < 0.01$) – The result is considered suspisious. The user should generate more data from the source and test it again. This result is possible to happen by chance even when the tested data is produced by a good source.
- **OK** ($0.01 \leq P(x) < 1$) – The result is evaluated as a good result and the data is considered random.

4 Analysing arbitrary binary data with RTT

In the chapter, we present the results of two distinct experiments and describe the methods and battery configurations used for obtaining the results.

In the first experiment, we analysed large quantities of random data with the statistical batteries implemented in RTT. The primary motivation for the experiment was to establish a baseline for further experiments – how many tests and how frequently are expected to fail with truly random sequences. Our subsequent experiments would differ from the baseline analysis only in the input data but will keep the configuration settings of the tests in the batteries the same.

In the second experiment, we analysed outputs of 15 distinct cryptographic algorithms. The specific algorithms were chosen based either on their real-world popularity and utility or their success in cryptographic competitions eSTREAM [16] and SHA-3 [17]. The chosen algorithms were AES, BLAKE, Grain, Grøstl, JH, Keccak, MD6, Rabbit, Salsa20, DES, 3-DES, Skein, SOSEMANUK, TES and RC4. We compared the results of the experiment with the results of the analysis in the EACirc framework. The EACirc framework developed at CRoCS implements an alternative approach to the randomness testing [6].

In our experiments, all statistical tests have significance level α set to 0.01. The tests will reject the null hypothesis H_0 based on this significance level.

4.1 Battery configurations in the experiments

In this section, we will present the configurations of the individual batteries that we used in the experiments mentioned above. To enable direct comparison with batteries executed as stand-alone tools instead of a part of RTT, we kept the settings as close as possible to the default settings of the individual batteries. The used RTT battery configuration file is in Appendix G.

4.1.1 Common settings

All batteries were provided with 8000 MiB[1] large data files for the analysis in each execution. The batteries were not required to process the entirety of the data; however rewinding the file and reading more bytes was not allowed. We chose the file size as a compromise between the amount of data needed by the separate tests in the batteries and our capabilities to process and store such quantities of data.

NIST Statistical Testing Suite

The settings of the tests in NIST STS was the same as the execution of the battery with default settings. The tests along with their default settings are listed in Appendix B. In a single run, each test processed 1000 data streams of size 1000000 bits each; in total, each test analysed 125 MB of data per execution.

1. The exact length of a single file was $8000 \cdot 1024^2 = 8388608000$ bytes.

Dieharder

The test settings of the Dieharder battery follow default settings as outlined in Appendix C with the single exception of the RGB Lagged Sums test. In default settings, the test would require too much data[2] for the analysis.

The RGB Lagged Sum test is executed in 33 variants with variable parameter `n={0..32}`. The number of bytes processed by the test variant scales with the parameter `n` of the variant and the number of times the variant will be repeated (p-samples option) during the battery execution. To lower the amount of the necessary bytes, we reduced the number of repetitions for each variant based on its `n`. The number of repetitions was calculated using the following function.

$$\mathrm{Rep}(n) = \min\left(100; \left\lfloor \frac{8000 \cdot 1024^2}{(n+1) \cdot 4 \cdot 10^6} \right\rfloor\right)$$

TestU01

We executed the batteries Small Crush and Crush in their default settings. The default settings of the tests in their respective batteries can be found in TestU01 documentation [12, p. 143].

The battery Big Crush is omitted from our analysis because the tests in the battery would need at least 60 GB of data; the amount of data would be infeasible both to generate and store.

The tests in Rabbit, Alphabit and Block Alphabit batteries were executed in their default settings. We limited the tests to 8000 MiB of data by the parameter `nb`.

The tests in the batteries Alphabit and Block Alphabit have settings that would set them to process only certain parts of each 32-bit block in the data. We did not use these settings; the tests analysed all bits in 32-bit blocks. Additionally, all tests from Block Alphabit are executed in 6 different variants; each test variant analyses differently reordered blocks of 32 bits [12, p. 155].

4.2 Baseline experiment

The aim of this experiment was to observe behaviour and results of the batteries in RTT when analysing truly random data. We use the results of this analysis to interpret the results of our further analyses.

We evaluate the battery analysis based on the number of failed tests in the battery. Under the assumption that we have single second-level p-value per single test, the test is considered failed when its p-value is outside of the interval $[\alpha, 1)$; null hypothesis H_0 of the test is rejected. The reason why we excluded one from the non-rejecting interval is explained in Section 2.3.1.

2. RGB Lagged Sum test with parameter `n=32` will analyse 13.2 GB of data.

One of the properties of the p-values is that for an arbitrary statistical test they will be uniformly distributed over the interval $(0, 1]$ as long as the null hypothesis is satisfied [18]. Our null hypothesis is that the tested data were the product of TRNG. Since the p-values are uniformly distributed, they will fall outside of the interval $[\alpha, 1)$ with probability equal to α; H_0 of the test will be rejected with probability α. When the null hypothesis of a test is rejected, we assess the test as failed.

When we are evaluating a battery analysis based on a count of failed tests, we must know how many tests we expect to fail when H_0 holds true. Let's have a battery with 40 independent tests and the significance level 0.01. If we don't know the expected number of failures, we must reject H_0 if we observe any number of failures. However, in this case, the probability that at least one test fails is approximately 33 % (the probability is calculated as a complement of the probability of failing exactly zero tests, see Equation 4.1); we would incorrectly reject H_0 in one-third of all battery executions.

We can calculate the expected count of failed tests like so. Let n be the number of tests in a battery. Then we have n independent experiments that will result in either yes (failure of the test) with probability α or no (success of the test) with probability $1 - \alpha$. This situation can be modeled by using binomial distribution $\mathrm{Bin}(n, \alpha)$ [19, p. 245]. Let F_{Exp} be a random discrete variable expressing the count of failed tests in a battery with expected distribution $\mathrm{Bin}(n, \alpha)$. From the definition of the binomial distribution, the expected value of a random variable is $E(F_{Exp}) = n\alpha$. Therefore, should a battery finish with $n\alpha$ failed tests we still can't reject H_0 for the whole battery; we will treat the analysed data as random.

However, the expected number of failures is not necessarily exact, as the calculation described above treats tests as independent trials. Separate tests in a battery are neither required nor proved to be independent, which can lead to differing distributions of empirical and theoretical random variables representing the counts of failed tests in the batteries.

To observe the empirical distribution of the number of failed tests F_{Obs}, we inspected the number of failures of the individual batteries during the analysis of truly random data. In total, approximately eight terabytes of random data were downloaded from quantum random number generator service provided by Humboldt University of Berlin [20]. The data were split into 1000 blocks each of size 8000 GiB. All seven batteries included in this experiment analysed each block of data once. The sample size of 1000 trials is enough for us to compare the distributions of the two random variables F_{Exp} and F_{Obs}.

We inspected the counts of failures of individual batteries in following two scenarios.

Original second-level p-values failures

In this scenario, we assumed that all second-level p-values produced by a specific battery are independent. Therefore, we treated all statistics calculated by the battery as separate tests. For example, if we execute some test in two variants, each of those variants has two subtests[3], and each subtest will calculate two statistics, in total we will get $2^3 = 8$ second-level p-values; we consider these eight p-values as the results of eight independent tests each of whose calculated a single p-value.

3. For the difference between variant and subtest, see Section 2.1.

Corrected p-values failures
With this second scenario, we tried to mitigate the effect of dependent p-values in our set by grouping them together. We grouped the p-values that are most likely dependent; we assumed that the p-values that are most likely inter-connected are those that are produced by a single statistical test. Therefore, all p-values belonging to a single test were reduced to only one p-value, leaving us with a single result per distinct test in a statistical battery. For example, should a single test produce eight p-values, for this test we will only have information whether the test was a failure or not. For the grouping of the p-values, we used Šidák's correction [21], and we show the process in Method 4.2.1.

Method 4.2.1. For a multiset of n grouped p-values $P = \{p_1, .., p_n\}$ with significance level α and null hypothesis H_0 we will calculate the corrected significance level $\alpha_0 = 1 - (1 - \alpha)^{\frac{1}{n}}$. If $\exists p \in P : (p < \alpha_0) \vee (p = 1)$, then H_0 is rejected for $\forall p \in P$ and the whole group result is treated as a failure. We reject $p = 1$ because of the reason in Section 2.3.1.

To compare the viability of the two scenarios, we needed to compare both of the observed distributions to their respective expected distributions and see how they perform. We consider the approach more viable when it is less differing from its expected distribution; if the observed distribution is behaving as expected, we can easily calculate the expected number of test failures in a battery run. In the rest of the section, we describe the comparison process. The comparison is illustrated in Example 4.2.1.

Definition 4.2.1. Let $X = \{x_1, .., x_n\}$ be a multiset of n elements. Then we define frequency variable f_e^X as the frequency of element e in multiset X.

During this experiment each battery was executed 1000 times; for each battery we had $s = 1000$ samples of a discrete random variable F_{Obs}, expressing the count of failed tests in that battery. Random variable F_{Exp} expresses the theoretical expected count of failures in a battery with n test and the significance level α. Variable F_{Exp} follows binomial distribution $\text{Bin}(n, \alpha)$. The probability of observing x test failures (written as $F_{Exp} = x$) is defined with probability mass function of binomial distribution (Equation 4.1).

$$P(F_{Exp} = x) = \binom{n}{x} \alpha^x (1 - \alpha)^{n-x} \tag{4.1}$$

Let $X = \{x_1, .., x_s\}$ be a multiset of the samples of F_{Obs} and $D = \{d_1, .., d_k\}$ set of all k distinct values in X. Then $f_d^X, d \in D$ (see Definition 4.2.1) represents a random variable expressing the frequency of a failure count within a multiset X. The random variable f_d^X is expected to follow binomial distribution $\text{Bin}(s, p)$ where s is the number of samples and p is the probability of observing d failures ($P(F_{Exp} = d)$, Equation 4.1). The expected value of the random variable f_d^X is defined using expected value of binomial distribution (Equation 4.2).

$$E\left(f_d^X\right) = sP(F_{Exp} = d) \tag{4.2}$$

Now that we have both the observed frequency f_d^X and its expected value $E\left(f_d^X\right)$ for $d \in D$, we can assess their closeness. For this assessment, we used Pearson's chi-squared

test (χ^2) that evaluates whether two tested datasets belong to the same distribution [19, p. 219]. We calculated Pearson's cumulative test statistic χ^2 for each pair of expected and observed frequencies as shown in Equation 4.3.

$$\chi^2 = \sum_{i=1}^{k} \frac{\left(f_{d_i}^X - E\left(f_{d_i}^X\right)\right)^2}{E\left(f_{d_i}^X\right)} \tag{4.3}$$

Due to the limitation of Pearson's χ^2 test stating that the expected value must be at least 5 [19, p. 220], the frequencies that didn't satisfy this condition were summed together and treated as a single frequency. We then calculated p-value using χ^2 distribution [19, p. 116]; the p-value is calculated as $1 - \mathrm{CDF}(df, \chi^2)$, where CDF is the cumulative distribution function of the χ^2 distribution, df are the degrees of freedom (equal to $k - 1$) and χ^2 is the value of the χ^2 test statistic.

The resulting p-value expresses the probability that the measured sets belong to the same distribution. Therefore, when we are comparing the viability of the two failure counting scenarios (original versus corrected results) the more viable one will have greater p-value than the other; our point was to find out which one of our approaches followed our theoretical expectations more closely.

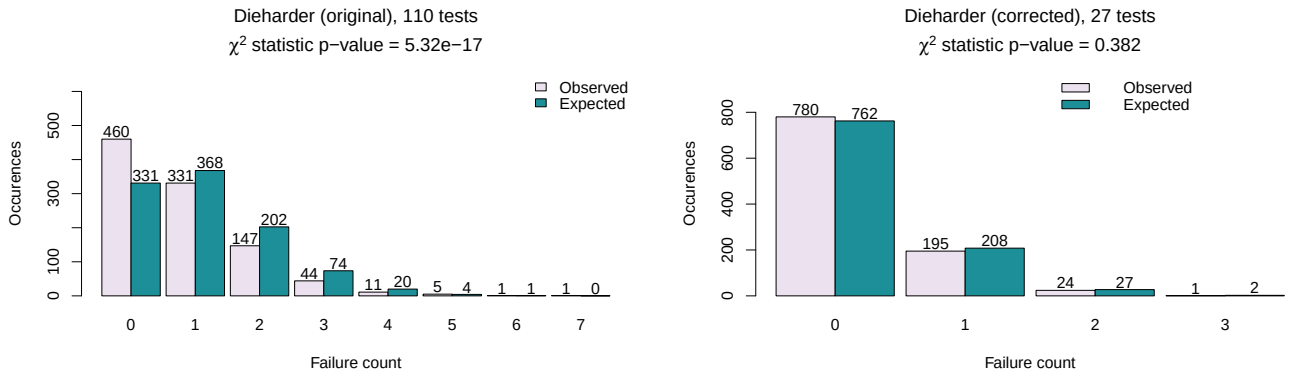

Figure 4.1: Test failure frequency distributions in Dieharder battery

Example 4.2.1. Let's have a multiset $X = \{1, 3, 4, 3, .., 0\}$, in which are stored the counts of failed tests in 1000 executions of Dieharder battery (in the first execution failed one test, in the second execution three tests, etc.); therefore we have $s = 1000$. There are $n = 110$ uncorrected tests in the Dieharder battery. In Figure 4.1 we can see the observed and expected frequencies of the failure counts for both corrected and uncorrected tests. We calculate the expected frequency for each failure count as follows (Equation 4.1, Equation 4.2); we use zero failure count as an example.

$$E(f_0^X) = sP(F = 0) = 1000\binom{110}{0}0.01^0(1 - 0.01)^{110-0} \approx 331.03$$

Based on the expected and observed frequencies in the Figure 4.1 we calculate the χ^2 statistic (Equation 4.3);

$$\begin{aligned}\chi^2 &= \sum_{i=1}^{k} \frac{\left(f_{d_i}^X - E\left(f_{d_i}^X\right)\right)^2}{E\left(f_{d_i}^X\right)} \\ &= \frac{(460-331.03)^2}{331.03} + \frac{(331-367.81)^2}{367.81} + \cdots + \frac{(7-5.12)^2}{5.12} \\ &\approx 85.71\end{aligned}$$

Note, that the frequencies for five, six and seven failures are summed together; this is caused by the limitation of Pearson's χ^2 test that is mentioned above. We calculate the final p-value of the χ^2 statistic as $1 - CDF(df, \chi^2) = 1 - CDF(5, 85.71) = 5.32 \cdot 10^{-17}$. The resulting p-value is the probability that the observed and expected frequency of the failure counts originate from the same distribution.

We can compare the χ^2 p-values shown in Figure 4.1. As the p-values suggest, in the corrected version of failure counting, the observed failure frequency is much closer to the expected distribution than in the original version.

	Original F_{Obs}		**Corrected** F_{Obs}		
Name	χ^2	$\max(F_{Obs})$	χ^2	$\max(F_{Obs})$	**Acc. bound**
Dieharder	$5.32 \cdot 10^{-17}$	7/110	0.38	3/27	3/27
NIST STS	$2.17 \cdot 10^{-2}$	7/188	$4.44 \cdot 10^{-7}$	3/15	2/15
Small Crush	0.71	3/15	0.95	2/10	2/10
Crush	$3.36 \cdot 10^{-11}$	8/186	$3.31 \cdot 10^{-3}$	3/32	3/32
Rabbit	$2.02 \cdot 10^{-5}$	6/58	$1.45 \cdot 10^{-23}$	2/16	2/16
Alphabit	$2.14 \cdot 10^{-8}$	8/33	$2.8 \cdot 10^{-7}$	2/4	1/4
Block Alphabit	$1.87 \cdot 10^{-68}$	14/198	$5.15 \cdot 10^{-47}$	3/4	1/4

Table 4.2: Summary of test failure distributions and acceptance bound of the batteries

In Table 4.2 we present comparison of the χ^2 statistic p-values between the original and the corrected results. We can see, that in general, the corrected results are closer to their expected distributions. Only in two cases (NIST STS and Rabbit), the corrected results perform worse while in the remaining batteries they offer a significant improvement. We also present the acceptance bounds for the batteries. The acceptance bound represents the maximum number of failed tests in a battery, while not rejecting the null hypothesis. Should the number of failures exceed the acceptance bound, we will reject H_0; the tested data failed to pass the battery.

However, even with the improved result interpretation, some test failure counts are still too different from their expected values. This is probably caused by currently unknown dependencies existing between the groups of tests; the accurate detection of the

dependencies is a part of our future research. To correct the inaccuracies, we decided to increase the acceptance bounds. We set the acceptance bounds in such a way that the probability of Type I error is not higher than 0.001. Therefore, we are evaluating the whole battery with the significance level $\alpha = 0.001$ and the individual tests with $\alpha = 0.01$.

The presented bounds are used for result interpretation of the subsequent experiments. If the number of failed tests will be higher than the bound, the data did not pass the battery analysis.

4.3 Analysis of well-known cryptographic algorithms

In this experiment, we focused on the analysis of well-known and used cryptographic primitives. Our motivation for this experiment was to find out how big are the security margins of conventional algorithms. Most of the tested algorithms could be reduced in the number of their rounds. We investigated how much could we reduce the rounds until we would start detecting the biases in the data.

The outputs of the functions were generated using the generator developed in CRoCS [15]. The plaintext of the functions was a binary counter; the key was randomly generated, and initialization vector was set to zeroes. Not all functions needed the key and IV.

For each particular round and function combination, we generated a data file with the size of 8000 MiB. In total, we analysed 72 distinct combinations of functions and rounds. The configurations of the batteries were the same as described in Section 4.1.

The results of this experiment were also compared to the results produced by the EACirc framework [6]. The results of the analysis of the data with EACirc are part of Karel Kubíček's diploma thesis [22].

4.3.1 Results of the analysis

The results of the analysis are shown in Table 4.3. We included only the combinations of algorithms and rounds in which the data failed to pass either EACirc or any battery in RTT. The marked cells in the table indicate failure of the tool.

The values in the battery columns express the number of failed tests in a specific battery. Each battery has defined the total number of tests in the header. In some cases, not all tests in the battery could be executed; in these cases, we specify the number of performed tests in the cell.

For each tested data file, EACirc was executed multiple times and in each execution the null hypothesis was either rejected or retained; for this experiment, EACirc was executed 1000 times for each file. The values in EACirc column display the proportion of EACirc executions that rejected the null hypothesis. When analysing random data, we expect that EACirc will reject the hypothesis in 1 % of runs. If the proportion is greater than 0.02, we can confidently consider the data as non-random.

We also provide security margins of the analysed algorithms. The security margins are based on the highest round in which the algorithm failed to pass at least one of the batteries and on the default total number of rounds in the algorithm.

Some algorithms in the table are marked by an asterisk. In highest shown rounds, some of these algorithms passed all of the batteries (e.g. RC4 and Grain); however, after closer inspection, we discovered some notable patterns in the results. These findings are summarized in Section 4.3.2.

The discovered patterns in the results were consistent failures of certain tests when analysing algorithm with fixed number of rounds (e.g. Rabbit with 4 round will not pass the `sstring_HammingIndep` test). We consider a test as consistently failing when we create multiple datasets of the same algorithm and round and the test will fail in all analyses of the datasets. The generation of the datasets can vary in the used keys or plaintexts.

For each suspicious algorithm with fixed number of rounds, we generated additional 16000 MiB of data. The additional data showed the same patterns in the results as in the original analysis, hinting the presence of a particular bias produced by an algorithm.

Algorithm	**Round**	**Total rounds**	**NIST STS** (x/15)	**Dieharder** (x/27)	**Small Crush** (x/10)	**Crush** (x/32)	**Rabbit** (x/16)	**Alphabit** (x/4)	**Block Alphabit** (x/4)	**EACirc** (proportion)	**Security margin** (rounds)	**Security margin** (proportion)
AES	3	10	8	15	5	20	5	2	4	0.182	7	0.70
BLAKE	1	16	11	11	5	18	5	2	3	0.107	15	0.93
Grain*	2	13	14	27	9/9	31/31	15	4	4	1.000	7	0.53
	6		1	0	0	3	1	0	0	0.017		
Grøstl	2	14	12	23	9	27	9	3	3	1.000	12	0.85
JH	6	42	12/13	27	10	30/31	15	4	4	1.000	36	0.85
Keccak	2	24	14	27	10	31	15	4	4	1.000	21	0.87
	3		0	1	1	11	4	0	3	0.017		
MD6*	8	104	9	19	5	16	8	2	3	0.748	94	0.90
	10		0	0	0	2	3	0	0	0.010		
Rabbit*	0	4	1	1	0	4	3	1	1	0.017	0	0
	4		0	0	0	3	3	1	1	0.009		
Salsa20	2	20	12	26	8	28	11	3	3	1.000	18	0.90
Single DES	4	16	7	22	7	26	11	4	4	0.193	11	0.68
	5		1	6	1	18	5	2	3	0.010		
Skein	3	72	12	27	10	30	13	3	4	1.000	68	0.94
	4		0	0	0	10	4	1	3	0.014		
SOSEMANUK	4	25	13/13	27	10	31/31	16	4	4	1.000	21	0.84
TEA	4	32	8	19	6	15	4	2	3	0.444	27	0.84
	5		0	3	2	4	1	0	3	0.012		
Triple DES	2	16	12	26	9	31	15	3	4	1.000	13	0.81
	3		1	4	1	4	0	1	1	0.017		
RC4* (roundless)	–	–	0	0	0	3	0	0	0	0.009	–	0

Table 4.3: Analysis of the cryptographic algorithms with RTT and EACirc

4.3.2 Notable observations

In this section, we list our observations based on the above-presented results, according to their importance.

Biased output of the Rabbit cipher
Rabbit [23] cipher is one of the winners in the eSTREAM competition in the software section. The algorithm has a total of 4 rounds. In all of its configurations, Rabbit will consistently fail tests `sstring_HammingIndep` and `sstring_PeriodsInStrings` in batteries Crush, Rabbit, Alphabit and Block Alphabit.

The tests are aiming to discover dependencies between separate bits in the data stream. One interesting detail is that the tests in Block Alphabit will fail if and only the bits in the stream are not previously re-ordered by the battery. This fact led us to believe that there indeed is some dependency between bits in the output.

Biases in the output of the Rabbit cipher were previously published in papers *On a bias of Rabbit* [24] and *Cryptanalysis of Rabbit* [25]. The authors of the latter paper claimed to be able to construct distinguisher for the cipher that would need 2^{158} bits of output. With the help of RTT, we are able to distinguish the Rabbit cipher in just 8 GB of data; however, we make strong assumption about zero initialization vector, while the authors of the paper do not assume anything.

Biased output of the RC4 cipher
RC4 [26] is a popular stream cipher widely used in WEP and TLS up until the version 1.3 when it was discontinued. Various attacks on RC4 were published, and in 2015 it was prohibited for use in TLS by RFC 7465 [27]. In its full version (no reduction of rounds or algorithm), the RC4 cipher will consistently fail tests `sknuth_SimpPoker` and `sknuth_Gap` in the Crush battery from TestU01.

There are multiple known biases and distinguishing attacks on RC4; notable publications are *A Practical Attack on Broadcast RC4* [28], *Analysis of Non-fortuitous Predictive States of the RC4 Keystream Generator* [29] and *Statistical Analysis of the Alleged RC4 Keystream Generator* [30]. Any of these biases can be causing the failure of the statistical tests.

Security margins with respect to the statistical analysis
The majority of the analysed algorithms has the security margin higher than 0.8. This means that even if we would drastically increase our computational capability and performance of the state-of-the-art statistical tools, biases in full versions of algorithms (without reduction of rounds) will not be detected.

The margins are significantly lower only for functions with particular biases detected, namely Rabbit, RC4 and Grain.

Strength of the batteries in comparison to EACirc
All of the batteries implemented in RTT perform better or equally well than EACirc. The shortcomings of EACirc can be caused by the fact that the framework analyses only 4.6 MB of data per run. EACirc is executed multiple times during the analysis, but the multiple executions only affect the precision of the obtained results, not the strength of the tool. Although in this experiment, EACirc needed 4,6 GB of data for the analysis, we can configure it to consume only 460 MB, at the cost of some precision[4].

4. This is a very simplified view. For the details about EACirc see [22].

In many situations we aren't able to retrieve 8000 MiB (e.g. slow hardware generators in smartcards); in these cases it might be viable to extract just 460 MB and analyse the data with EACirc. Therefore we have a trade-off between the needed time and resources and strength of the tool.

The batteries themselves have differing strength. While batteries NIST STS and Small Crush have more or less same bias detection capabilities as EACirc, the remaining batteries perform better and can detect subtler biases. The remaining batteries are listed here, along with the number of times they detected a bias when other batteries failed to do so: Alphabit (2 times), Dieharder (3 times), Block Alphabit (5 times), Rabbit (6 times), Crush (8 times).

Biased output of round reduced Grain function
The Grain cipher [31] is one of the winners of the eSTREAM competition in the hardware section. In 3, 4, 5 and 6 rounds (out of 13), the test will consistently fail tests `smarsa_MatrixRank` and `scomp_LinearComp` in batteries Crush and Rabbit. Bias in 6 rounds does not break the security of the cipher, but due to its existence, the security margin is significantly lower (around 0.5) than in other well-known functions.

Biased output of round reduced MD6 hash function
The MD6 hash function [32] was a competitor in the SHA-3 competition but did not advance to the second round of the competition. In 9 and 10 rounds out of 94, the algorithm will consistently fail tests `smarsa_MartixRank` and `sspectral_Fourier3` in batteries Crush and Rabbit. While this hints us with the presence of a specific bias in the function output it does not break the security of the function; in further rounds, we can no longer detect any bias, and the output appears as truly random.

5 Validation of outputs of Dieharder

The goal of the research presented in this chapter was to analyse the validity of the results of the Dieharder battery[1]. To obtain the results for later analysis, we processed eight terabytes of truly random data with each test included in the Dieharder battery. We used the same set of data as in the experiment described in Section 4.2.

5.1 Goals of the research

As stated in Section 4.2, the p-values produced by an arbitrary statistical test are uniformly distributed over the interval $(0, 1]$ if the null hypothesis holds true. This assumption should also hold for the first-level p-values of the individual tests in the battery.

Dieharder uses this assumption to process the results further. In the standard settings, the user of the battery will not be presented the first-level p-values of the tests as a result. Each test in Dieharder will be repeated multiple times resulting in multiple first-level p-values; these p-values will be then tested for uniformity by the Kolmogorov-Smirnov (K-S) test [19, p. 473]. Should the K-S test fail, the assumption of the uniformity is rejected, and the test is deemed failed; the user should consider the tested data non-random.

The aim of this research was to verify the assumption mentioned above; that the analysis of data produced by TRNG yields uniform results.

5.2 Settings of the experiment

We analysed outputs of the tests executed with their default settings; the used parameters can be found in Appendix C. The tests marked as not used were omitted from our analysis.

We inspected the uniformity of first-level p-values of all subtests and test variants present in default Dieharder run. For example, if a single test is executed in two variants and both variants have two subtests[2] then the test will produce four sets of first-level p-values; we treat these sets independently as if four separate tests produced them. In this chapter, we are referring to these low-level tests as to atomic tests.

During the default run, the battery executes 110 atomic tests, resulting in 110 individual p-value sets. Each atomic test process the eight terabytes of data from start to finish to extract the maximum possible amount of p-values. The final size of the p-value set varies, as each atomic test processed a different amount of data in a single execution. Each p-value in the set represents a single execution of an atomic test.

1. The research is an extended version of the project for the course PA018 Advanced Topics in Information Technology Security, Autumn 2016.
2. For the difference between variant and subtest, see Section 2.1.

5.3 Testing the uniformity of p-values

We inspected the uniformity of the p-value sets using the proportion statistic and Pearson's chi-squared (χ^2) goodness-of-fit test [19, p. 219].

5.3.1 Proportion test

The proportion statistic is evaluated based on the number of observed test failures within the p-value sets. We consider a test failed if its p-value is lesser than the significance level $\alpha = 0.001$. Let F be a discrete random variable expressing the count of failed tests within a p-value set with n elements; the p-values in the set are samples of F. According to our previous assumption, the p-values should be uniformly distributed on the interval $(0, 1]$, and therefore the probability of a single test failure is α. Then the random variable F should follow binomial distribution $\text{Bin}(n, \alpha)$ [19, p. 245]; the expected value of the variable is $E(F) = n\alpha$. The probability of observing x failures within the p-value set is calculated using the probability mass function of binomial distribution shown in Equation 5.1.

$$P(F = x) = \binom{x}{n} \alpha^x (1 - \alpha)^{n-x} \tag{5.1}$$

If we observe x failures, we can calculate the probability of observing equal or more extreme count of failures under the assumption that F follows the distribution $\text{Bin}(n, \alpha)$. The more extreme result is a result more distant from the expected value. The probability is calculated as shown in Equation 5.3.

$$D = |E(F) - x| \tag{5.2}$$

$$P(|E(F) - F| \geq D) = 1 - \left(\sum_{k=\lfloor E(F)-D \rfloor + 1}^{\lceil E(F)+D \rceil - 1} P(F = x) \right) \tag{5.3}$$

The p-value of the proportion statistic is the probability described above. If the p-value is lesser than 0.001, we reject the hypothesis that F follows the binomial distribution $\text{Bin}(n, \alpha)$ and consider the inspected p-value set non-uniform.

5.3.2 Pearson's chi-squared test

The Pearson's chi-squared (χ^2) test evaluates the distribution of the p-values on the entire interval $(0, 1]$. First, the set of p-values P is split into several bins; in this testing we used $k = 1000$ bins. The individual bins in the set $B = \{b_1, .., b_k\}$ have values as follows; the bin $b_i \in B$ has value equal to the count of p-values $p \in P$ such that $\left(\frac{i-1}{k} \leq p < \frac{i}{k}\right)$. For example, the value of the bin b_i will be equal to the count of p-values greater or equal to zero and lesser than 0.001. The expected value of b_i is calculated as $\frac{n}{k}$. The Pearson's chi-

squared test statistic χ^2 is calculated based on the values of the bins and their expected values as shown in Equation 5.4.

$$\chi^2 = \sum_{i=1}^{k} \frac{\left(b_i - \frac{n}{k}\right)^2}{\frac{n}{k}} \tag{5.4}$$

The p-value of the χ^2 test statistic is calculated as $1 - \mathrm{CDF}(df, \chi^2)$; the calculation is described in Section 4.2. If the calculated statistic p-value is lesser than 0.001, we reject the hypothesis that the p-values from the set P are uniformly distributed.

5.4 Results of the testing

The resulting p-values of the uniformity testing are presented in Table 5.2. We included only the atomic tests that failed one or both uniformity tests presented in Section 5.3. For each atomic test, we show the total count of extracted p-values, the expected and observed count of failures within the set and p-values of both uniformity statistics we calculated. The cells that contain p-values lesser than 0.001 are marked.

In Figure 5.1 we visualise distributions of p-value sets of four chosen atomic tests. The Diehard Minimum Distance (2d Circe) test was included because its p-values are uniformly distributed, and the graph can be used as a baseline for comparison with other visualised sets. The histograms also contain the p-values of the calculated χ^2 statistics.

From the presented results we can see that out of 110 atomic tests, 39 produced non-uniform p-values. This result is surprising since prior to the experiment, we presumed that all tests produce uniform results. The discovery is important in the context of further processing of p-values by the Dieharder battery. The battery assumes that if the null hypothesis holds, the first-level p-values are uniformly distributed; the uniformity of the set is tested by the K-S test. However, the fact that some p-value sets are significantly and consistently non-uniform breaks this assumption.

This significant non-uniformity of results might not be apparent in basic usage of Dieharder for two reasons. The first potential reason is that, during normal usage, only small sets of p-values (usually 100) are tested for the uniformity; in a limited number of p-values, the non-uniform patterns are harder to detect.

The second possible reason is that, in default Dieharder, a test is considered failed only if its second-level p-value is lesser than 10^{-6}. If we assume that everything is functioning properly, we should see a failure of a test once in a million of executions. Due to the non-uniformity of the underlying p-values, the actual number of failures can be higher (10 or even 100 failures) but still too small to notice in the common use-case. Additionally, the small significance level introduces high type II error; the tests in Dieharder have lowered sensitivity and might not reject the null hypothesis for non-random data.

During the experiment, we made several observations. The most obvious one is that the Pearson's χ^2 test is much more viable for testing the uniformity than the proportion test. The χ^2 test evaluates the entire interval between 0 and 1, while the proportion test inspects only the beginning of the interval. The superiority of the χ^2 test is supported by

Test name	P-value count	Failure counts		P-values	
		Expected	Observed	Fail prop.	χ^2 unif.
Diehard tests					
Birthday	54613333	54613.3	60109	$< 1 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
32x32 Binary Rank	1638399	1638.4	1634	0.931	$2 \cdot 10^{-29}$
6x8 Binary Rank	3495253	3495.3	4911	$< 1 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
Bitstream	7999938	8000	7951	0.587	$< 1 \cdot 10^{-323}$
Count the 1s (stream)	32767487	32767.5	44779	$1.9 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
Count the 1s (byte)	1638399	1638.4	2249	$< 1 \cdot 10^{-8}$	$3.1 \cdot 10^{-67}$
Parking Lot	87381333	87381.3	93227	$< 1 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
Min Distance (3d sphere)	174762666	174762.7	176065	0.002	$4.7 \cdot 10^{-28}$
Runs - subtest 1	20971519	20971.5	35308	$< 1 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
Runs - subtest 2	20971519	20971.5	35246	$< 1 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
Craps - subtest 1	838860	838.9	825	0.641	$< 1 \cdot 10^{-323}$
NIST STS tests					
Monobit	20971519	20971.5	20742	0.113	$< 1 \cdot 10^{-323}$
Runs	20971519	20971.5	21256	0.050	$< 1 \cdot 10^{-323}$
Serial - Subtest 1	20970519	20970.5	20740	0.111	$< 1 \cdot 10^{-323}$
Serial - Subtest 2	20970519	20970.5	21131	0.270	$3.7 \cdot 10^{-228}$
Serial - Subtest 4	20970519	20970.5	21125	0.289	$8.7 \cdot 10^{-192}$
Serial - Subtest 23	20970519	20970.5	20971	1.000	$9.8 \cdot 10^{-5}$
Serial - Subtest 24	20970519	20970.5	20930	0.780	0.0002
Serial - Subtest 25	20970519	20970.5	21125	0.289	$4.7 \cdot 10^{-15}$
Serial - Subtest 26	20970519	20970.5	20968	0.986	$1 \cdot 10^{-7}$
Serial - Subtest 27	20970519	20970.5	21124	0.292	$6.6 \cdot 10^{-81}$
Serial - Subtest 28	20970519	20970.5	21005	0.817	$2.8 \cdot 10^{-15}$
Serial - Subtest 29	20970519	20970.5	20990	0.898	$3.1 \cdot 10^{-228}$
Serial - Subtest 30	20970519	20970.5	21029	0.691	$1 \cdot 10^{-67}$
RGB tests					
Bit Distribution (n=2)	5242866	5242.9	5516	$1.7 \cdot 10^{-4}$	0.3641
Bit Distribution (n=8)	1310719	1310.7	1286	0.498	$3.2 \cdot 10^{-11}$
Bit Distribution (n=9)	1165083	1165.1	1301	$7.4 \cdot 10^{-5}$	$< 1 \cdot 10^{-323}$
Bit Distribution (n=10)	1048575	1048.6	1050	0.988	$< 1 \cdot 10^{-323}$
Bit Distribution (n=11)	953250	953.3	1041	0.005	$< 1 \cdot 10^{-323}$
Bit Distribution (n=12)	873812	873.8	895	0.488	$< 1 \cdot 10^{-323}$
Generalized Min Distance (n=4)	52428799	52428.8	51953	0.038	$< 1 \cdot 10^{-323}$
Generalized Min Distance (n=5)	41943039	41943	41561	0.062	$< 1 \cdot 10^{-323}$
Permutations (n=2)	10485759	10485.8	10341	0.158	$< 1 \cdot 10^{-323}$
Permutations (n=3)	6990506	6990.5	6974	0.843	$1.8 \cdot 10^{-5}$
Permutations (n=4)	5242879	5242.9	5247	0.961	$2.7 \cdot 10^{-19}$
Permutations (n=5)	4194303	4194.3	4241	0.473	$9.1 \cdot 10^{-114}$
Kolmogorov-Smirnov Test	209715199	209715.2	202696	$3.8 \cdot 10^{-8}$	$< 1 \cdot 10^{-323}$
DAB tests					
DCT	163839	163.8	162	0.907	$2.9 \cdot 10^{-40}$
Monobit 2	32263	32.3	28	0.537	$1.2 \cdot 10^{-132}$

Table 5.2: Atomic tests with significantly non-uniform p-value sets

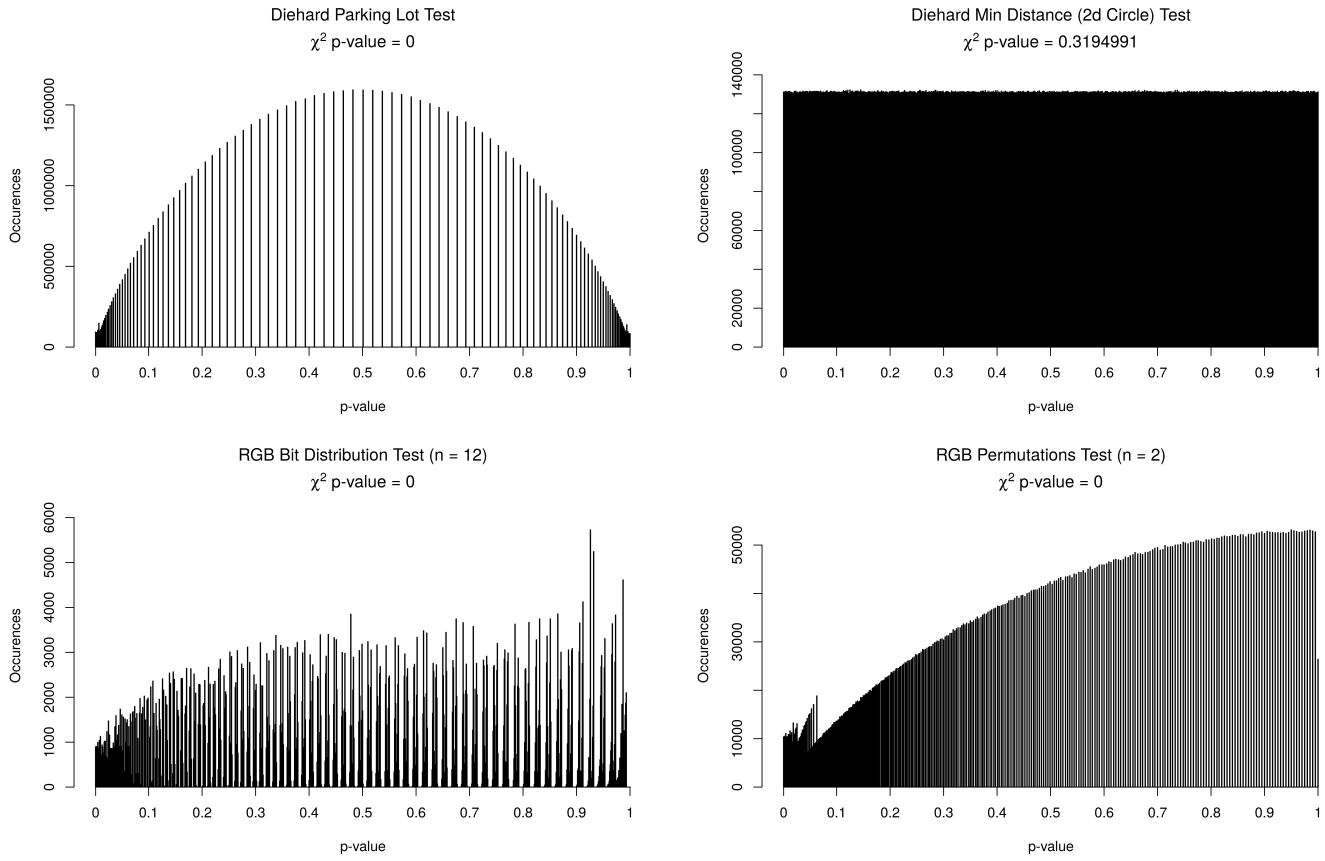

Figure 5.1: Histograms of p-values of chosen atomic tests. Only the Min Distance (2d Circle) test produces uniform p-values.

the fact, that only in a single case, the proportion test detected non-uniformity when the χ^2 test did not.

Our further observation was that even within a single test, its variants and subtests could produce both uniform and non-uniform results. Specifically, some subtests of the STS Serial test produced non-uniform p-values while the remaining subtests produced valid results. Similarly, the uniformity of results of the RGB Bit Distribution test depends on its parameter `n`. This behaviour can be noticed in multiple various tests.

The results of this experiment inspired our further research that aims to inspect the non-uniformity of the atomic tests more deeply. Another goal of the further work is to find out the exact impact of the non-uniformity on the second-level p-values.

6 Summary

The aim of this work was to explore and assess current state-of-the-art approaches to the statistical testing of randomness. Our further goal was to design and implement an interface to these statistical tools and to unify them into a single toolkit.

We presented an overview of the statistical tools used for randomness testing; to observe the unexpected behaviour of the tools we executed them with border-case inputs. When we run the batteries with extremely non-random data (e.g. stream of zeroes), we observed various undocumented situations such as frozen battery execution or invalid test results. Based on the findings, we made adjustments in our toolkit design.

With respect to the design and the initial requirements of the project, we developed Randomness Testing Toolkit [7], or RTT for short. RTT implements an interface for the three presented tools: NIST Statistical Testing Suite [1], Dieharder [2] and TestU01 [3]. It is possible to extend the framework to include additional statistical tools in the future.

The RTT framework consists of multiple parts intended for various purposes. Currently, the framework can be used on user's local machine, but it can also be deployed on multiple servers and accessed through the web or console interface. The web interface also eases the result interpretation for the user.

To establish a baseline for the interpretation of the results of the statistical tools, we used RTT to analyse eight terabytes of binary data; the data were generated using quantum random number generator [20]. All batteries analysed the data in their default settings. We used the established baseline to interpret our subsequent experiments.

In further experiments, we analysed 15 distinct popular and well-known cryptographic algorithms such as AES, Keccak or RC4. The results of the analysis were compared to the results produced by the EACirc framework; EACirc, a tool developed at CRoCS, implements an alternative approach to the statistical testing [6]. From the comparison of the results, we concluded that RTT could detect subtler biases present in the data better that EACirc. However, there is a trade-off between the two approaches as the EACirc framework needs a much lesser amount of data and resources for the analysis.

The notable result of the experiment was that RTT is able to distinguish the keystreams of the Rabbit [23] and RC4 [26] stream ciphers from the random data. The Rabbit cipher is one of the winners of eSTREAM [16] cryptographic competition and RC4 is an algorithm commonly used in TLS before the version 1.3. The only restriction on the ciphers is that we need at least 8 GB of keystream generated with a single arbitrary key and the initialization vector must be set to zero.

We also inspected the validity of the results produced by the Dieharder battery tests. The validity evaluation was based on the assumption that the test results should be uniformly distributed over the interval $(0, 1]$ when random data were analysed. We processed eight terabytes of random data with each test in the Dieharder battery; this way we obtained 110 sets of results while each set was assumed to be uniformly distributed. We found that 39 (out of 110) sets broke this assumption and were non-uniform. This is significantly more than we originally anticipated as the Dieharder battery relies on the uniformity during post-processing of the results.

6.1 Future work

The future development and research can follow up on this thesis in multiple different directions.

Further development of RTT
The one possibility of extending RTT is to add more statistical batteries into the interface. Viable candidates for addition are the PractRand battery [33] and the EACirc framework. The other possible extension of RTT is the implementation of the result visualisation directly in the web interface; the visualisation would make the interpretation of the data analysis even easier for the user.

Study of the dependencies between the statistical tests
As stated in Section 4.2, we lack a deep understanding of dependency between the individual tests in the statistical batteries. For NIST STS, the research was done in [34]; however, no such research exists for Dieharder or TestU01. The advanced knowledge of the dependencies would allow us to further improve the accuracy of the battery results interpretation.

Impact of Dieharder non-uniformity on overall battery results
In Chapter 5 we discovered that a large number of tests in Dieharder produce non-uniform results. Further research on this topic is necessary as we know neither what is causing the non-uniformity nor the exact impact of the broken assumptions on the overall results of Dieharder.

Bibliography

1. NATIONAL INSTITUTE FOR STANDARDS AND TECHNOLOGY. *Statistical Test Suite* [online]. 1997. Version 2.1.1 [visited on 2017-05-19]. Available from: http://csrc.nist.gov/groups/ST/toolkit/rng/index.html.
2. BROWN, Robert G. *Dieharder: A Random Number Test Suite* [online]. 2004. Version 3.31.1 [visited on 2017-05-19]. Available from: http://www.phy.duke.edu/~rgb/General/dieharder.php.
3. L'ECUYER, Pierre; SIMARD, Richard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Transactions on Mathematical Software*. 2007, vol. 33, no. 4. Available from DOI: 10.1145/1268776.1268777.
4. CAELLI, William et al. *Crypt X Package Documentation* [online]. 1992 [visited on 2017-05-19]. Available from: https://web.archive.org/web/20060819041438/http://www.isi.qut.edu.au/resources/cryptx/.
5. MARSAGLIA, George. *Diehard Battery of Tests of Randomness* [online]. 1995 [visited on 2017-05-19]. Available from: https://web.archive.org/web/20120102192622/www.stat.fsu.edu/pub/diehard/.
6. ŠVENDA, Petr; KUBÍČEK, Karel; SÝS, Marek, et al. *EACirc: Framework for automatic search for problem solving circuit via evolutionary algorithms* [online]. 2012. Version 4.2 [visited on 2017-05-19]. Available from: https://github.com/crocs-muni/EACirc.
7. OBRÁTIL, Ľubomír; ŠVENDA, Petr. *Randomness Testing Toolkit* [online]. 2015. Version 1.0 [visited on 2017-05-19]. Available from: https://github.com/crocs-muni/randomness-testing-toolkit.
8. SHUTTLEWORTH, Martyn. *Type I Error – Type II Error* [online]. 2008 [visited on 2017-05-19]. Available from: https://explorable.com/type-i-error.
9. RUKHIN, Andrew et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications* [online]. National Institute of Standards and Technology (NIST), 2000 [visited on 2017-05-19]. Available from: http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf. Technical report.
10. SÝS, Marek; ŘÍHA, Zdeněk; MATYAS, Vashek. Algorithm 970: Optimizing the NIST Statistical Test Suite and the Berlekamp-Massey Algorithm. *ACM Transactions on Mathematicsl Software*. 2016, vol. 43, no. 3, pp. 27:1–27:11. ISSN 0098-3500. Available from DOI: 10.1145/2988228.
11. BROWN, Robert G. *dieharder(1) – Linux man page* [online]. Version 3.31.1 [visited on 2017-05-19]. Available from: https://linux.die.net/man/1/dieharder.
12. L'ECUYER, Pierre; SIMARD, Richard. *TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators* [online]. 2013. Version 1.2.3 [visited on 2017-05-19]. Available from: http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf.
13. NATIONAL INSTITUTE FOR STANDARDS AND TECHNOLOGY. *FIPS 140-2: Security Requirements For Cryptographic Modules* [online]. 2001 [visited on 2017-05-19]. Available from: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf.

14. MCGRATH, Roland. *chroot(1) – Linux man page* [online] [visited on 2017-05-15]. Available from: https://linux.die.net/man/1/chroot.

15. KUBÍČEK, Karel; HAJAS, Michal. *EACirc Generator* [online]. 2017. Version 1.0 [visited on 2017-05-19]. Available from: https://github.com/crocs-muni/eacirc-streams.

16. EUROPEAN NETWORK OF EXCELLENCE FOR CRYPTOLOGY. *eStream project: Call for Stream Cipher Primitives* [online]. 2005 [visited on 2017-05-19]. Available from: http://www.ecrypt.eu.org/stream/call/.

17. NATIONAL INSTITUTE FOR STANDARDS AND TECHNOLOGY. *SHA-3: Cryptographic hash algorithm competition* [online]. 2007 [visited on 2017-05-19]. Available from: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

18. MURDOCH, Duncan J; TSAI, Yu-Ling; ADCOCK, James. P-Values are Random Variables. *The American Statistician*. 2008, vol. 62, no. 3, pp. 242–245. Available from DOI: 10.1198/000313008X332421.

19. SHESKIN, David J. *Handbook of parametric and nonparametric statistical procedures*. 3rd ed. CRC Press, 2003. ISBN 978-1-58488-440-8.

20. NANO-OPTICS GROUP AND PICOQUANT GMBH. *High Bit Rate Quantum Random Number Generator Service* [online]. 2010 [visited on 2017-05-19]. Available from: http://qrng.physik.hu-berlin.de/.

21. ŠIDÁK, Zbyněk. Rectangular Confidence Regions for the Means of Multivariate Normal Distributions. *Journal of the American Statistical Association*. 1967, vol. 62, no. 318, pp. 626–633. Available from DOI: 10.1080/01621459.1967.10482935.

22. KUBÍČEK, Karel. *Optimisation heuristics in randomness testing*. submitted in 2017. Available also from: https://is.muni.cz/th/408351/fi_m/. master thesis. Faculty of Informatics, Masaryk University.

23. WIKIPEDIA. *Rabbit (cipher) — Wikipedia, The Free Encyclopedia* [online]. 2017 [visited on 2017-05-19]. Available from: https://en.wikipedia.org/wiki/Rabbit_(cipher).

24. AUMASSON, Jean-Philippe. *On a bias of Rabbit* [eSTREAM, ECRYPT Stream Cipher Project, Report 2007/033 (SASC 2007)]. 2007 [visited on 2017-05-19]. Available from: http://www.ecrypt.eu.org/stream.

25. LU, Yi; WANG, Huaxiong; LING, San. Cryptanalysis of Rabbit. In: *Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings*. Ed. by WU, Tzong-Chen; LEI, Chin-Laung; RIJMEN, Vincent; LEE, Der-Tsai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 204–214. ISBN 978-3-540-85886-7. Available from DOI: 10.1007/978-3-540-85886-7_14.

26. WIKIPEDIA. *RC4 — Wikipedia, The Free Encyclopedia* [online]. 2017 [visited on 2017-05-19]. Available from: https://en.wikipedia.org/wiki/RC4.

27. POPOV, Andrei. *Prohibiting RC4 Cipher Suites* [Internet Requests for Comments]. 2015. Available from DOI: 10.17487/RFC7465. RFC.

28. MANTIN, Itsik; SHAMIR, Adi. A Practical Attack on Broadcast RC4. In: *Fast Software Encryption: 8th International Workshop, FSE 2001 Yokohama, Japan, April 2–4, 2001 Revised Papers*. Ed. by MATSUI, Mitsuru. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 152–164. ISBN 978-3-540-45473-1. Available from DOI: 10.1007/3-540-45473-X_13.

29. PAUL, Souradyuti; PRENEEL, Bart. Analysis of Non-fortuitous Predictive States of the RC4 Keystream Generator. In: *Progress in Cryptology - INDOCRYPT 2003: 4th International Conference on Cryptology in India, New Delhi, India, December 8-10, 2003. Proceedings*. Ed. by JOHANSSON, Thomas; MAITRA, Subhamoy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 52–67. ISBN 978-3-540-24582-7. Available from DOI: 10.1007/978-3-540-24582-7_4.

30. FLUHRER, Scott R.; MCGREW, David A. Statistical Analysis of the Alleged RC4 Keystream Generator. In: *Fast Software Encryption: 7th International Workshop, FSE 2000 New York, NY, USA, April 10–12, 2000 Proceedings*. Ed. by GOOS, Gerhard; HARTMANIS, Juris; LEEUWEN, Jan van; SCHNEIER, Bruce. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 19–30. ISBN 978-3-540-44706-1. Available from DOI: 10.1007/3-540-44706-7_2.

31. WIKIPEDIA. *Grain (cipher) — Wikipedia, The Free Encyclopedia* [online]. 2016 [visited on 2017-05-19]. Available from: https://en.wikipedia.org/wiki/Grain_(cipher).

32. WIKIPEDIA. *MD6 — Wikipedia, The Free Encyclopedia* [online]. 2017 [visited on 2017-05-19]. Available from: https://en.wikipedia.org/wiki/MD6.

33. DOTY-HUMPHREY, Chris. *PractRand* [online]. 2010. Version 0.92 [visited on 2017-05-19]. Available from: http://pracrand.sourceforge.net.

34. SÝS, Marek; ŘÍHA, Zdeněk; MATYÁŠ, Václav; MÁRTON, Kinga; SUCIU, Alin. On the Interpretation of Results from the NIST Statistical Test Suite. *Romanian Journal of Information Science and Technology*. 2015, vol. 18, no. 1, pp. 18–32.

A Data attachment

TODO!!!

B NIST STS Tests

Common parameters of all tests

- **Stream size** – 1000000
- **Stream count** – 1000

ID	Test name	Block size	Subtests
1	Frequency (Monobit)	–	1
2	Frequency within a Block	128	1
3	Cumulative Sums (Cusums)	–	2
4	Runs	–	1
5	Longest Run of Ones in a Block	–	1
6	Random Binary Matrix Rank	–	1
7	Discrete Fourier Transform (Spectral)	–	1
8	Non-overlapping (Aperiodic) Template Matching	9	148
9	Overlapping (Periodic) Template Matching	9	1
10	Maurer's "Universal Statistical"	–	1
11	Approximate Entropy	10	1
12	Random Excursions	–	8
13	Random Excursions Variant	–	18
14	Serial	16	2
15	Linear Complexity	500	1

C Dieharder Tests

Note: Tests labeled as not used are excluded from our experiments; they are flagged as suspicious or not working in Dieharder documentation. Tests with plus sign next to the stream size will process variable amount of data on each run. The value is only orientational.

ID	Test name	Stream size (byte)	P-samples	Arguments
	Diehard tests			
0	Birthdays	153 600	100	–
1	OPERM5	4 000 020	100	–
2	32×32 Binary Rank	5 120 000	100	–
3	6×8 Binary Rank	2 400 000	100	–
4	Bitstream	1 048 584	100	–
5	OPSO **(Not used)**	–	–	–
6	OQSO **(Not used)**	–	–	–
7	DNA **(Not used)**	–	–	–
8	Count the 1s (stream)	256 004	100	–
9	Count the 1s (byte)	5 120 000	100	–
10	Parking Lot	96 000	100	–
11	Minimum Distance (2D Circle)	64 000	100	–
12	Minimum Distance (3D Sphere)	48 000	100	–
13	Squeeze	10 000 000+	100	–
14	Sums **(Not used)**	–	–	–
15	Runs	400 000	100	–
16	Craps	10 000 000+	100	–
	Marsaglia and Tsang tests			
17	GCD	80 000 000	100	–
	STS tests			
100	Monobit	400 000	100	–
101	Runs	400 000	100	–
102	Serial (Generalized)	400 000	100	–
	RGB tests			
200	Bit Distribution	n×800 000 + 4	100	`-n {1..12}`
201	Generalized Minimum Distance	n×40 000	100	`-t 10000 -n {2..5}`
202	Permutations	n×400 000	100	`-n {2..5}`
203	Lagged Sum	(n+1)×4 000 000	100	`-n {0..32}`
204	Kolmogorov-Smirnov	40 000	1000	–
	DAB tests			
205	Byte Distribution	614 400 000	1	–
206	DCT	51 200 000	1	–
207	Fill Tree	500 000 000+	1	–
208	Fill Tree 2	200 000 000+	1	–
209	Monobit 2	260 000 000	1	–

D Overview of options in toolkit settings

The tags that are listed in the top-level description must contain JSON object with key-value pairs with options and their values. For complete example of the JSON see Appendix E.

toolkit-settings/logger

- **dir-prefix** *Optional* – If set, value of this tag will become prefix of all log files locations. Can be used when all logger directories should have the same parent directories.
- **run-log-dir** – Sets location of the main log file.
- **<battery>-dir** – Value **<battery>** is replaced by values dieharder, nist-sts, tu01-smallcrush, tu01-crush, tu01-bigcrush, tu01-rabbit, tu01-alphabit and tu01-blockalphabit. All of these tags are mandatory and their values set locations of files that will contain raw outputs of the respective executed batteries.

toolkit-settings/result-storage/file *Optional*
For details about the file storage see Section 3.2.1.

- **main-file** – Sets path of the file with final results of the program run. If the file already exists, results will be added to it, if not, new result file will be created.
- **dir-prefix** – If set, all directory values will have this prefix.
- **<battery>-dir** – Value **<battery>** can be replaced by values dieharder, nist-sts, tu01-smallcrush, tu01-crush, tu01-bigcrush, tu01-rabbit, tu01-alphabit and tu01-blockalphabit. The tags set locations of files with detailed results of the respective battery.

toolkit-settings/result-storage/mysql-db *Optional*
For details about the MySQL database storage see Section 3.2.2.

- **address** Address of the MySQL server with created RTT database.
- **port** Port on which the MySQL server is accessible.
- **name** Name of the database scheme with RTT tables.
- **credentials-file** Path to the file that contains login information for the database. The example credentials file is shown in Figure D.1.

toolkit-settings/binaries

- **<battery>** – Value **<battery>** is replaced by nist-sts, dieharder and testu01. All of the tags are mandatory. The values of the tags sets the locations of the executables of the respective batteries.

toolkit-settings/miscelaneous/nist-sts

- **main-result-dir** – Sets the directory where NIST STS stores its result files.

toolkit-settings/execution

- **max-parallel-tests** Sets maximum number of concurrently running test processes. The higher value may cause the analysis to finish faster but will cause higher strain on the system.
- **test-timeout-seconds** Sets time period after which the running tests will be considered stuck and will be killed.

```
{
  "credentials": {
    "username": "jane_doe",
    "password": "password"
  }
}
```

Figure D.1: Example of MySQL database credentials file

E Sample file with RTT settings

```
{
  "toolkit-settings": {
    "logger": {
      "dir-prefix": "rtt-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": "rtt-results/table.txt",
        "dir-prefix": "rtt-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": {
        "address": "127.0.0.1",
        "port": "3306",
        "name": "rtt",
        "credentials-file": "credentials.json"
      }
    },
    "binaries": {
      "nist-sts": "/home/rtt-statistical-batteries/nist-sts",
      "dieharder": "/home/rtt-statistical-batteries/dieharder",
      "testu01": "/home/rtt-statistical-batteries/testu01"
    },
    "miscelaneous": {
      "nist-sts": {
        "main-result-dir": "experiments/AlgorithmTesting/"
      }
    },
    "execution": {
      "max-parallel-tests": 8,
      "test-timeout-seconds": 3600
    }
  }
}
```

F Guide to the settings in the battery configuration

In following list we describe common as well as battery specific tags that can be present in battery configuration file. Each tag is listed with its description and list of its allowed parent tags. For example of complete configuration see Appendix G.

Common settings

`randomness-testing-toolkit`
Root tag of the configuration file.

`<battery>-settings`
Possible parent tags: `randomness-testing-toolkit`
<battery> can be dieharder, nist-sts, tu01-smallcrush, tu01-crush, tu01-bigcrush, tu01-rabbit, tu01-alphabit and tu01-blockalphabit. These tags contain settings for respective batteries.

`defaults`
Possible parent tags: `<battery>-settings`
Options inside this tag are considered as the default options for the battery specified by the parent tag.

If certain test is executed in the battery and needs to have some option defined but it does not, the option falls back to the value defined in the `defaults` tag.

`test-ids`
Possible parent tags: `defaults`
The option defines which tests will be executed in the specific battery by default. The default tests are executed only if no test was specified through command-line arguments.

Value of this tag must be array of strings. Each string can contain either a single number or range of numbers. For example, should the value of the tag be `["3", "10-13", "15"]`, then the default IDs of the tests in the battery will be 3, 10, 11, 12, 13, 15.

`test-specific-settings`
Possible parent tags: `<battery>-settings`
This tag can contain list of single or several JSON objects. Each object defines options that are specific to a single test and the test's variants in the battery specified by the parent tag. Each of the objects must contain tag `test-id` for assigning the options to the test.

If the execution of a test requires a certain option that is not defined in the test object, the value of the option is taken from the tag `defaults`.

`test-id`
Possible parent tags: `test-specific-settings`
The tag must be present in each object that is in `test-specific-settings` object list. The value of the tag is used for assigning the options in the test object to actual test from the battery.

`variants`
Possible parent tags: `test-specific-settings`
The tag can be present in the test object in `test-specific-settings`. The tag can contain

list of single or multiple JSON objects. Each of the variant objects can have defined additional options for test variants. The options that will be undefined by the variant object will be taken from the test object or the `defaults` tag. The variants are tied to the test which is specified by the `test-id` tag in the test object. For example, should the test object contain `variant` tag with list of three objects, then three variants of the test will be executed.

NIST STS settings

`stream-size`
Possible parent tags: `defaults, test-specific-settings, variants`
Sets bit size of single stream of data that will be used for testing.

`stream-count`
Possible parent tags: `defaults, test-specific-settings, variants`
Sets number of streams of data that will be used for testing. E.g. if this value will be 100, then the test will be repeated 100 times, each time with different input data of length `stream-size`.

`block-length`
Possible parent tags: `defaults, test-specific-settings, variants`
Some tests in NIST STS analyse the data streams in blocks. This option sets the bit size of the blocks. For details on which tests accept this parameter, see Appendix B.

Dieharder settings

`psamples`
Possible parent tags: `defaults, test-specific-settings, variants`
Sets how many times will be the test repeated before final statistic is calculated.

`arguments`
Possible parent tags: `defaults, test-specific-settings, variants`
Allows you to pass additional arguments to the Dieharder binary. Format is expected as single string of argument-value pairs separated by spaces (e.g. `"-x 2048 -y 30"`). For summary of arguments accepted by Dieharder, see its documentation.

TestU01 settings

`repetitions`
Possible parent tags: `defaults, test-specific-settings, variants`
Sets how many times will be a test repeated. Setting this option to more that 1 will produce several multiple of the test.

`bit-nb`
Possible parent tags: `defaults, test-specific-settings, variants`
This option is valid only for rabbit, alphabit and block alphabit batteries. Sets how many bytes can be processed by a test. The test will not process more bytes.

bit-r
Possible parent tags: `defaults, test-specific-settings, variants`
This option is valid only for alphabit and block alphabit batteries. Sets the offset in each group of 32 bits. For example, if set to r then first r bits from each 32 bits will be ignored. Only the rest of the bits in each 32-bit block will be processed.

bit-s
Possible parent tags: `defaults, test-specific-settings, variants`
This option is valid only for alphabit and block alphabit batteries. Sets how many bits from each 32-bit block will be processed, starting from `bit-r`. For example, if we set $r = 5, s = 10$, then only 10 bits will be processed in each 32-bit block, starting from the sixth bit.

bit-w
Possible parent tags: `defaults, test-specific-settings, variants`
This option is valid only for block alphabit battery. The value must be lesser or equal to `bit-s`, otherwise the option will be ignored. The option sets the reordering of the bits that will happen before processing. For details about the reordering see TestU01 documentation (TODO cite).

parameters
Possible parent tags: `defaults, test-specific-settings, variants`
This option is valid only for small crush, crush and big crush batteries. The value of the tag must be JSON object that will contain key-value pairs that will represent test's parameters and their values. All parameters of the test must be specified. Each test has different settable parameters, the parameters for each test in respective batteries can be found in TestU01 documentation [12, p. 143]. Example of the `parameters` tag for test smars_BirthdaySpacings is shown in Figure F.1.

```
{
  "parameters": {
    "N": "1",
    "n": "10000",
    "r": "0",
    "d": "1000000",
    "t": "2",
    "p": "1"
  }
}
```

Figure F.1: Parameters setting for smarsa_BirthdaySpacings test

G Example of battery configuration

Following battery configuration was used in both experiments described in Chapter 4.

```
{
  "randomness-testing-toolkit":{
    "dieharder-settings":{
      "defaults":{
        "test-ids":["0-4", "8-13", "15-17", "100-102", "200-209"],
        "psamples":100
      },
      "test-specific-settings":[
        {
          "test-id":200,
          "variants":[
            {"arguments":"-n 1"},{"arguments":"-n 2"},{"arguments":"-n 3"},
            {"arguments":"-n 4"},{"arguments":"-n 5"},{"arguments":"-n 6"},
            {"arguments":"-n 7"},{"arguments":"-n 8"},{"arguments":"-n 9"},
            {"arguments":"-n 10"},{"arguments":"-n 11"},{"arguments":"-n 12"}
          ]
        },
        {
          "test-id":201,
          "psamples":1000,
          "variants":[
            {"arguments":"-n 2 -t 10000"},{"arguments":"-n 3 -t 10000"},
            {"arguments":"-n 4 -t 10000"},{"arguments":"-n 5 -t 10000"}
          ]
        },
        {
          "test-id":202,
          "variants":[
            {"arguments":"-n 2"},{"arguments":"-n 3"},
            {"arguments":"-n 4"},{"arguments":"-n 5"}
          ]
        },
        {
          "test-id":203,
          "variants":[
            {"arguments":"-n 0"},{"arguments":"-n 1"},{"arguments":"-n 2"},
            {"arguments":"-n 3"},{"arguments":"-n 4"},{"arguments":"-n 5"},
            {"arguments":"-n 6"},{"arguments":"-n 7"},{"arguments":"-n 8"},
            {"arguments":"-n 9"},{"arguments":"-n 10"},{"arguments":"-n 11"},
            {"arguments":"-n 12"},{"arguments":"-n 13"},{"arguments":"-n 14"},
            {"arguments":"-n 15"},{"arguments":"-n 16"},{"arguments":"-n 17"},
            {"arguments":"-n 18"},{"arguments":"-n 19"},
            {"arguments":"-n 20", "psamples":99},{"arguments":"-n 21", "psamples":95},
            {"arguments":"-n 22", "psamples":91},{"arguments":"-n 23", "psamples":87},
            {"arguments":"-n 24", "psamples":83},{"arguments":"-n 25", "psamples":80},
            {"arguments":"-n 26", "psamples":77},{"arguments":"-n 27", "psamples":74},
            {"arguments":"-n 28", "psamples":72},{"arguments":"-n 29", "psamples":69},
            {"arguments":"-n 30", "psamples":67},{"arguments":"-n 31", "psamples":65},
            {"arguments":"-n 32", "psamples":63}
          ]
        },
        {
          "test-id":204,"psamples":1000
        },
        {
          "test-id":205,"psamples":1
        },
        {
          "test-id":206,"psamples":1
        },
        {
```

```
62          "test-id":207,"psamples":1
63        },
64        {
65          "test-id":208,"psamples":1
66        },
67        {
68          "test-id":209,"psamples":1
69        }
70      ]
71    },
72
73    "nist-sts-settings":{
74      "defaults":{
75        "test-ids":["1-15"],
76        "stream-size":"1000000",
77        "stream-count":"1000"
78      }
79    },
80
81    "tu01-smallcrush-settings":{
82      "defaults":{
83        "test-ids":["1-10"],
84        "repetitions":1
85      }
86    },
87
88    "tu01-crush-settings":{
89      "defaults":{
90        "test-ids":["1-96"],
91        "repetitions":1
92      }
93    },
94
95    "tu01-rabbit-settings":{
96      "defaults":{
97        "test-ids":["1-26"],
98        "repetitions":1,
99        "bit-nb":"8388608000"
100     }
101   },
102
103   "tu01-alphabit-settings":{
104     "defaults":{
105       "test-ids":["1-9"],
106       "repetitions":1,
107       "bit-nb":"8388608000",
108       "bit-r":"0",
109       "bit-s":"32"
110     }
111   },
112
113   "tu01-blockalphabit-settings":{
114     "defaults":{
115       "test-ids":[
116         "1-9"
117       ],
118       "repetitions":1,
119       "bit-nb":"8388608000",
120       "bit-r":"0",
121       "bit-s":"32"
122     },
123     "test-specific-settings":[
124       {
125         "test-id":1,
126         "variants":[
127           {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
128           {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
129         ]
130       },
```

```
        {
          "test-id":2,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":3,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":4,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":5,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"},{"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":6,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":7,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":8,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        },
        {
          "test-id":9,
          "variants":[
            {"bit-w":"1"}, {"bit-w":"2"}, {"bit-w":"4"},
            {"bit-w":"8"}, {"bit-w":"16"}, {"bit-w":"32"}
          ]
        }
      ]
    }
  }
}
```

H Example of file storage report file

```
***** Randomness Testing Toolkit data stream analysis report *****
Date:    05-04-2017
File:    data.bin
Battery: Dieharder

Alpha:   0.01
Epsilon: 1e-08

Passed/Total tests: 1/1
Battery errors:
Battery warnings:

------------------------------------------------------------
Diehard Runs Test test results:
  Result: Passed
  Test partial alpha: 0.00250943
  Variant 1:
    User settings:
      P-sample count: 10
    ************
    Subtest 1:
      Kolmogorov-Smirnov statistic p-value: 0.78507772 Passed
      p-values:
        0.05899849 0.14303914 0.25813353 0.26205674 0.29277325
        0.35258302 0.56733382 0.65239775 0.72661662 0.97024989
      ============
    #############
    Subtest 2:
      Kolmogorov-Smirnov statistic p-value: 0.98126677 Passed
      p-values:
        0.06708958 0.13845018 0.24143043 0.30884227 0.34883845
        0.45856887 0.50099963 0.72226328 0.79368168 0.94420838
      ============
    #############
  ^^^^^^^^^^^^^
  Variant 2:
    User settings:
      P-sample count: 20
    ************
    Subtest 1:
      Kolmogorov-Smirnov statistic p-value: 0.94522158 Passed
      p-values:
        0.02636030 0.05899849 0.07020394 0.11462969 0.14303914
        0.25813353 0.26205674 0.29277325 0.35258302 0.42201984
        0.49409172 0.50249791 0.50852031 0.56733382 0.65239775
        0.72661662 0.87379533 0.94805962 0.95109797 0.97024989
      ============
    #############
    Subtest 2:
      Kolmogorov-Smirnov statistic p-value: 0.97098651 Passed
      p-values:
        0.05940416 0.06708958 0.10176118 0.13845018 0.18591525
        0.20171335 0.24143043 0.30884227 0.34883845 0.45856887
        0.47711566 0.50099963 0.51679868 0.57821548 0.72226328
        0.73270941 0.79368168 0.91155189 0.94420838 0.99216956
      ============
    #############
  ^^^^^^^^^^^^^
------------------------------------------------------------
```

I Experiment submission online form

General

Experiment name

E-Mail

Binary data that will be analysed

Choose File No file chosen

The data is deleted after the analysis.

Configuration

Use default configuration (recommended)

Choose configuration

Configuration file (advanced user)

Choose File No file chosen

For configuration file description read our wiki.

Battery application

Switch all

NIST Statistical Testing Suite

Dieharder

TestU01 Small Crush

TestU01 Crush

TestU01 Rabbit

TestU01 Alphabit

TestU01 Block Alphabit

TestU01 Big Crush (requires at least 20GB of data, 60GB for full run)

Submit

You will be notified by email when the experiment finishes.