MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Analysis of pseudo-random number generators based on lightweight cryptographic primitives

MASTER'S THESIS

Michal Hajas

Brno, Fall 2018

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.

Michal Hajas

Advisor: RNDr. Petr Švenda, Ph.D.

Acknowledgements

Thank all.

Computational resources were supplied by the Ministry of Education, Youth and Sports of the Czech Republic under the Projects CESNET (Project No. LM2015042) and CERIT-Scientific Cloud (Project No. LM2015085) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

We also acknowledge the support of Czech Science Foundation, the project GA16-08565S.

Abstract

Abstract to be done

Keywords

randomness testing, cryptanalysis, block functions, lightweight cryptography, pseudo-radnom number generators

Contents

1 Introduction

2 Theory

2.1 PRNGs requirements

2.1.1 Constructions techniques

SIMPLE (PURE), Stream ciphers, block/hash fncs

2.2 Typical attacks

Nextbit, prevbit, seed/state recovery

2.3 Human cryptanalysis

[1]

2.4 Distinguishers from truly random streams

One of the most important property of output from cryptographic primitive is its indistinguishability from truly random data. This is why most automatic tools for analysing cryptographic primitives are based on this fact. Those tools relies on so called empirical tests of randomness. Each tool contains several tests, where each of them has different approach of testing. Mostly they are based on some property where there is high probability that truly random generator will satisfy this property. By comparing the expected and actual results for some property we can find out so called *bias*. Higher the bias is, there is less probablility that truly random generator outputted tested sequence.

Tests are based on testing *null hypothesis*, which is mostly formulated as data being tested are random. However those tests are based only on probability, this means that even truly random data may end up with bad results. The output of each test is so called *p-value* which can be described as probability that a truly random generator produces data which are less random than the data which were tested. To interpret the result we have significance level commonly known as α. If p-value is less than significance level we say we rejected the hypothesis on significance level α. The common value of α is 0.01. [2]

There are two types of errors which may occur during interpretation, Type I and Type II. Type I means that truly random data were rejected. The probability of Type I error is significance level α. Type II error is when data from bad generator are not rejected. This is denoted by β, however it is much harder to compute value of β because there are many possible types non-randomness which may occur. [2]

The very basic and the easiest test to understand is Monobit test, which is testing uniformity of distribution of binary zeroes and ones bits within tested data. It is based on the fact, that there is high probability that the amount of binary ones and zeroes is approximately the same assuming that each sequence occurs with same probability. High-level view of this test is shown in Figure 2.1.

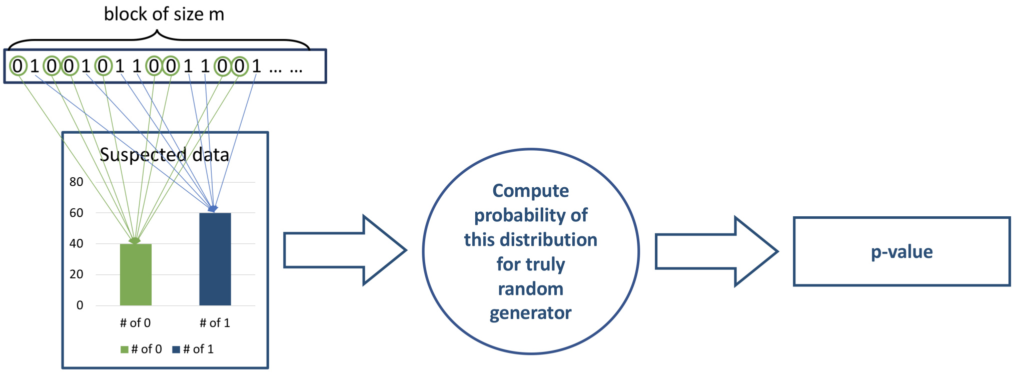

Figure 2.1: Monobit test from high level point of view.

Figure 2.2 shows high level view of whole test suite, where all tests are triggered. After evaluation of all tests, it is necessary to interpret whole run and make final decision, whether investigated data are considered truly random or not. It is quite likely that even for data with perfect properties few tests fails. Our interpretation was not based only on number of failed tests, but also on extremeness of test failures. For example if there was one failed test within whole test suite with extreme p-value (less than 10^{-7}) it is considered failure. On the other hand if there was even 3 failed tests with p-values close to α, not so extreme, it is still considered as non-failure.

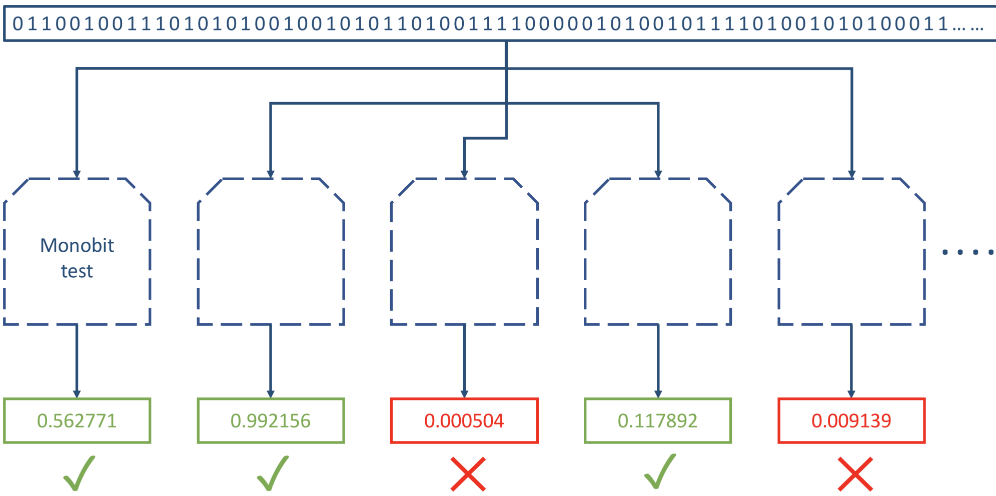

Figure 2.2: High level view of software tool run.

2.4.1 NIST STS

NIST STS [3] is most commonly used software tool for statistical analysis. It was developed by National Institute Of Standards and Technology (NIST) and is also part of FIPS 140-2 [4] certification process. Even thought the STS test suite is most commonly used, the other testsuites have generally better results.

In this thesis we will not use original NIST implementation, instead of that we are using optimized version developed Marek Sýs and Zdeněk Říha which is approximately 50 times faster than reference implementation [5].

The tool contains 15 tests. However few of them have more variants. Therefore altogether 188 tests are executed within one run.

2.4.2 Dieharder

Dieharder [6] is an extension of Diehard test suite [7] developed by Robert G. Brown at Duke University. In newest version It contains together 31 tests. All tests from Diehard, three originates from NIST STS and the other implemented by the author or from different sources. However not all tests will be used within this thesis, with choosing which tests to run and which not we rely on project Randomness testing toolkit [8]

2.4.3 TestU01

TestU01 software tool was introduced by Pierre L'Ecuyer and Richard Simard. The aim of this tool is provision of general and extensive set of software tools for statistical testing of random number generators. It contains big amount of tests, more than any other software tool mentioned above. Tests are organized into 6 batteries of tests. Battery of tests is simply subset of tests, where each battery has different purpose or time consuption. Batteries within TestU01 are divided into two categories, three of them designed for sequences of real numbers and the other for bit sequences.

For the first category there are batteries named *SmallCrush*, *Crush* and *BigCrush*. *SmallCrush* is fastest battery, hence it is recommended to start testing with this one and continue to *Crush* only if sequence pass all tests. *BigCrush* is longest one, for the idea it consumes 1414 times more time than *SmallCrush* and 5 times more time than *Crush* to test *Mersenne Twister* [9] PRNG on a computer with AMD Athlon running at 2.4GHz. For binary sequences there are batteries *Rabbit*, *Alphabit* and *BlockAlphabit*. [10]

Besides the batteries this software tool contains also some predefined pseudo-random number generators. Paper [10] contains also results of those generators with batteries *SmallCrush*, *Crush* and *BigCrush*.

2.4.4 BoolTest

BoolTest is simple but strong testing tool developed by Marek Sýs et. al at the Centre for Research on Cryptography and Security, Masaryk University in Brno. It has slightly different approach to randomness testing. It is a generalization of Monobit test. It is based on the idea of looking for distinguisher, between truly random and tested data, in the form of boolean function. If a distinguisher is found the data are considered non-random otherwise random. The process starts with dividing sequence into blocks with size m. The boolean function is in the form $f(x_1, x_2, \cdots, x_m)$. Notice that Monobit test is specific case of this generalization with $m = 1$ and boolean function $f(x_1) = x_1$.

Computation of distinguisher success is easy process. Simply take all blocks of size m and for each compute result of boolean function where x_k is k-th bit of the block. Expected number of computations with result binary one is computed and compared with actual results with Z-score statistic [11]. This leads to resulting p-value. The more actual number distinguish from expected number, the better distinguisher we have.

Construction of distinguisher is based on assumption that stronger and more complicated distinguishers can be constructed by combination of weaker and simpler ones. The run starts with brute-force evaluation of all possible boolean functions of type $f(x_1, x_2, \cdots, x_m) = x_i$ for $\forall i \in \{1, 2, \cdots, m\}$. It is also possible to choose more complex functions in this phase, but it cannot be too complicated as it is necessary to brute-force all possible functions. Then some number of best distiguishers is chosen to second phase in which they are combined them together with XOR operation. This approach has two main advantages: it is possible to use precomputed values from first phase and it is possible to stop evaluation at any time when reasonably strong distinguisher is found. If no strong distinguisher is found tested data are considered random. [12]

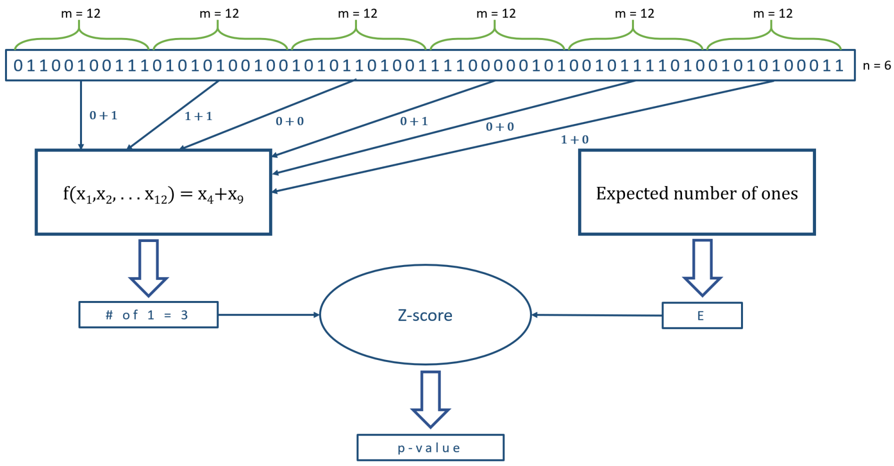

Figure 2.3: Example of evaluation with BoolTest for sequence of size 72 bits with block size 12 and number of blocks 6. Evaluated boolean function (distinguisher) is $f(x_1, x_2, ...x_{12}) = x_4 + x_9$.

3 Introduction of CryptoStreams tool

CryptoStreams tool is written in C++ language and is developed and maintained by team[1] at the Centre for Research on Cryptography and Security, Masaryk University [13]. The tool is used to generate a large amount of output data streams from parametrized cryptographic functions. Each stream is configurable with multiple options including output length, a structure of input plaintext, key, etc.

3.1 History

An initial implementation of CryptoStreams project was part of tool EACirc [14] which itself was a tool for automatic randomness testing based on genetic programming. At that moment it served only as a provider of testing data internally provided to its testing functionality and was not possible to use it separately outside of this project. We decided to split those two tools to provide flexible utilization with a broader range of tools in addition to EACirc. EACirc-streams project was introduced in 2017 and then in 2018 it was renamed to nowadays name, CryptoStreams.

3.2 Idea

The main idea behind CryptoStreams is an easy production of data from cryptographic primitives which are reduced in complexity, either by limiting the number of rounds or by providing them input with specific randomness properties, such as low Hamming weight, etc. The most significant advantage is that the tool contains a large number of cryptographic primitives such as block ciphers, hash functions, etc. All of them are integrated within a unified interface. CryptoStreams is also useful as an entry point for investigation of newly created cryptographic primitives, as it is built so that the addition of new primitives was as easy as possible. After obtaining data, it is possible to do any investigation over those data. For example, in this thesis, we conduct statistical analysis with seven statistical batteries of tests and also with a tool called BoolTest [12]. Notice, that addition of new analysis tool requires no additional implementation on the side of CryptoStreams.

3.3 Content of CryptoStreams

In this section, we would like to present deeper details about what this tool provides. The tool currently contains four types of cryptographic primitives: block ciphers, hash functions, stream ciphers, and pseudo-random number generators. Table 3.1 shows counts of functions of corresponding types. First cryptographic primitives which were added to CryptoStreams were candidates from SHA-3 and eStream competitions. Those additions were done by Ondrej Dubovec [15] and Matej Prišťák [16] in 2012. Within those theses were added 34 hash functions and 27 stream ciphers. Another addition was done by Martin Ukrop in his master thesis [17] regarding authenticated encryption systems from CAE-

1. The team of randomness testing involves following people: Radka Cieslarová, Michal Hajas, Dušan Klinec, Matúš Nemec, Jiří Novotný, Ľubomír Obrátil, Marek Sýs, Petr Švenda, Martin Ukrop and others.

Cryptographic primitive type	Number of functions
Block ciphers	42
Hash functions	51
Stream ciphers	27
PRNGS	6

Table 3.1: List of all types of cryptographic primitives contained within CryptoStreams with the corresponding number of functions.

SAR competition [18]. Well known block ciphers like AES, DES, etc. were added by Karel Kubíček [19] and Tamás Rózsa [20] in their theses. The tool also contains a lot of other cryptographic primitives added outside of theses or papers.

Each output is generated by so-called *streams* which are producers of data. Each call produces a chunk of data with configured size. Retrieving data from *stream* in a loop and storing them, results in a data file with desired binary data. By configuring the size of chunk and number of chunks to save it is possible to set the size of resulting file. CryptoStreams contains the following types of streams.

***Streams* outputting data with static structure** which are mostly used as an input *streams* such as plaintext or key. Those might be for example binary zero, binary one *stream* or low Hamming weight stream (small amount of ones), etc. Pseudo-random streams also belong to this category. Figure 3.1 shows example of schema of such *stream*.

The list of static streams is following: `dummy_stream`, `true_stream`, `false_stream`, `mt19937_stream`, `pcg32_stream`, `counter`, `random_start_counter`, `sac`, `sac_fixed_position`, `sac_2d_all_position` and `hw_counter`.

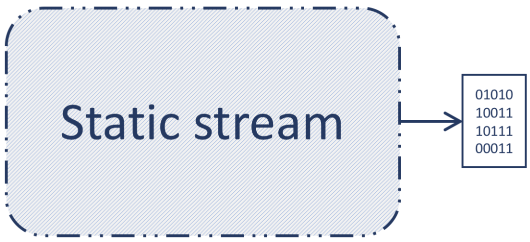

Figure 3.1: Example of one call of static stream.

Manipulating *streams* are configured with one or more inputs and manipulate them in a stream-specific way. For example, *repeating stream* is repeating one output specified number of times before generating a new chunk of data from input *stream*. Another example is *tuple stream* that is getting more *streams* as an input and for each call it returns chunk which contains data from each *stream* concatenated together. Schema of this *stream* is shown in Figure 3.2. Using tuple stream, it is possible to receive data which consists of plaintexts followed by corresponding ciphertexts.

Currently implemented streams within this category are: `single_value_stream`, `repeating_stream` and `tuple_stream`.

***Streams* based on round-reduced cryptographic primitives.** Besides the round limitation it is also possible to configure them with various types of plaintext, key and initial-

Figure 3.2: Example of one call from manipulating stream, specifically tuple stream.

ization vectors inputs. Figure 3.3 shows a scheme of such stream which uses block cipher. The scheme may be different for other cryptographic primitives, for example, hash functions do not need key or iv as an input.

Project contains 3 types of cryptoprimitves: `block_ciphers`, `stream_ciphers` and `hash_functions`. Number of functions wihitn corresponding category is shown in Table 3.1.

Figure 3.3: Example of one call of cryptographic primitive stream, where used function is block cipher.

***Streams* based on pseudo-random number generators.** Those *streams* are newly introduced as part of this thesis. It is not possible to round-reduce those types of generators; this means the only way how to weaken those generators are to provide them seed with specific randomness properties for example with low Hamming weight. As Figure 3.4 shows those streams are seeded in the beginning and then provide infinitely many output chunks. It is also possible to reseed generator after some specified number of *chunks* generated.

All functions within this category are presented in Section 3.5.

Notice that all inputs are in the form of another *stream*. Receiving *stream* is deciding how much data and when will use from *streams* on its input. For example, if receiving *stream* is a cryptographic primitive type which requests new plaintext for each chunk (e.g., from input static stream), but the key is random and fixed for the whole generation. See the figure Figure 3.5 for an example of a scheme of the whole run of CryptoStreams. The middle block is the most important one. It is *cryptographic primitive stream* and on its input it has 3 *static streams*.

Figure 3.4: Example of three calls for a new chunk with same seed from pseudo-random generator stream.

Figure 3.5: Scheme of CryptoStreams configuration from Figure 3.6. More information about specific streams within this picture can be found in Section 3.3.1

3.3.1 Configuration

All necessary configuration within one run is achieved using a `JSON` file. Example of such configuration file can be found in Figure 3.6 which will result in a file of size 8GB, generated from the AES function limited to 5 rounds. Key is generated pseudo-randomly using PCG32 [21] at the beginning of a run and then used for generation of each output chunk. *Counter stream* is used to generate plaintext blocks. Scheme of this configuration is shown in Figure 3.5.

The whole run of the generator is deterministic as it is using a pseudo-random generator with a specified seed. Experiments are therefore fully replicable using only stored `JSON` configuration. Notice that resulting file size is derived from `chunk_size` in Bytes and `chunk_count`. All possible options of configuration of CryptoStreams can be found in project documentation [22].

3.3.2 Testing of streams

Our statistical analysis relies on the fact that data which come from CryptoStreams are correct and genuinely come from cryptographic primitives. For that reason, we introduced testsuite which contains a various number of tests per individual *stream*. We have added tests only for most frequently used *streams*. Tests are also required for all newly added *streams*. Results in this thesis are based on *streams* which are tested.

```
{
  "chunk_count": 500000000,
  "chunk_size": 16,           8GB in total (500m * 16B)
  "file_name": "AES_r05_b16.bin",
  "seed": "1fe40505e131963c",
  "stream": {
    "type": "block",
    "algorithm": "AES",       Stream from block cipher AES
    "round": 5,               Number of rounds
    "block_size": 16,
    "key_size": 16,
    "iv_size": 16,
    "init_frequency": "only_once",    Key is generated
                                      only once
    "plaintext": {
      "type": "counter"       Definition of plaintext stream
    },
    "key": {
      "type": "pcg32_stream"  Definition of key stream
    },
    "iv": {
      "type": "false_stream"  Definition of iv stream
    }
  }
}
```

Figure 3.6: Example of `JSON` configuration for the tool CryptoStreams.

As testing backend we are using Google Test [2] framework. To ensure all tests are passing for each new change we use continuous integration tool called Travis CI[3].

There are two things which are important to test First one is function itself, source code which is included within CryptoStreams. The other thing is CryptoStreams superstructure which encapsulates all cryptographic primitives and functions into one interface. Each type of *streams* have different testing scenarios.

Block ciphers *streams* are tested with test vectors in both directions, encrypt and decrypt. Also, both mentioned layers are tested. All those tests are testing function only in the full number of rounds as we were not able to find test vectors for round limited version. For lightweight cryptographic primitives based *streams* we also added an *encrypt-decrypt* test, for all supported rounds which is testing whether encryption of plaintext followed by decryption results with inputted plaintext. However, this test does not work for all added functions; the reasons are summarized in Section 3.5.2. Test coverage is complete for block ciphers with all 42 functions supplied with test vectors.

Hash functions *streams* are tested with test vectors in the full number of rounds. Both low-level function and CryptoStreams superstructure are included in tests. 29 out of 51 hash functions are covered with tests.

Stream ciphers *streams* are tested similarly as block ciphers except for encrypt-decrypt test for round reduced versions. From 27 stream ciphers 15 are tested.

2. `https://github.com/google/googletest`
3. `https://travis-ci.org`

Crypto primitive type	Block ciphers	Hash functions	Stream ciphers	PRNGS
Tested/All functions	42/42	29/51	15/27	4/6

Table 3.2: List of all types of cryptographic primitives with the number of functions covered with tests.

Pseudo-random generators *streams* are hard to test, we have not found any test vectors. Nonetheless, we at least added test for linear generators by implementing an expected succession of numbers in tests and then compare whether we are getting same numbers from generator implementation. Four generators out of six included in CryptoStreams are tested. However, one test is testing only whether the generator is running without checking the correctness of output.

Other *streams*, static or manipulating, are easy to test as we know how exactly should output look like.

3.4 Conducted experiments

The experiment we conducted was based on testing of randomness provided by functions in some extreme scenario. Either by the limitation of function rounds if it is available or by providing specific input. We tested the following scenarios.

A specific type of input, mostly with some lousy randomness properties, is provided to functions and statistical analysis is conducted where results are compared with random or other specific inputs. For this thesis, we used the following types of inputs.

1. Counter *stream* is such *stream* in which each chunk is the addition of one to the previous chunk in number representation.
2. Low Hamming weight *stream* returns outputs with least count of ones it is possible. Starting with all zeroes followed with only binary one on each position, two binary ones, etc.
3. Strict avalanche criterion is a type of *stream* in which the first chunk is randomly generated, and every next call is just previous chunk with one flipped bit.

Round reduction. Almost every cryptographic function is build so that it performs a very same sequence of operations defined number of times, mostly in a loop. The number of times sequence should be performed is denoted by term *number of rounds*. For example, AES [23] has a recommended number of rounds 10, 12 or 14 based on key length as you can see in Figure 3.7. Each round in AES consists of *SubBytes, ShiftRows, MixColumns, AddRoundKey* operations. The original implementation of AES has different key scheduling phase based on the number of rounds. However, we keep key scheduling in the full number of rounds for all functions in CryptoStreams so that we avoid some undefined behavior based on uninitialized parts of the key.

Authors of each cryptographic function specify how many rounds should be used to have reasonable security and performance. This number is called *full number of rounds* and should be determined by conducting all known attacks against function limited to several numbers of rounds. Important indicator during this process is called *security margin*. Security margin denotes the rate between round vulnerable

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 3.7: Table of rounds based on key size which is taken from [23], word size is 32 bits.

to some cryptanalysis attack and the full number of rounds. For example, at the moment it is not possible to find any practical shortcut attack (any attack which is more efficient than exhaustive key search) for more than six rounds for AES with key length 128 bits. Therefore it is recommended to add four more rounds as a security margin and use AES with ten rounds [24]. Notice that *security margin* is not some general notion for the whole function, instead of that it is just kind of expression of resistance against particular cryptanalysis technique or attack.

In this thesis, we investigate security margin based on indistinguishability of output of cryptographic primitive limited to all possible rounds from a random stream.

Plaintext ciphertext stream produces pairs of input and output to function. With statistical testing performed on such output, it is possible to investigate dependency between plaintext and ciphertext. Since plaintext is part of the resulting sequence it is important to choose data with good randomness properties otherwise it may cause some interference in testing results.

3.4.1 Testing with Randomness testing toolkit

For running experiments, we are using a tool called randomness testing toolkit (RTT). It is a project which unifies more software tools for randomness testing under one interface. It provides both graphical interface (GUI) in the form of webpage and command line interface (CLI). CLI is mainly used for submission of a higher number of experiments due to possible automation, for example with python or shell scripts. Also GUI provides webpage form for submission of an experiment, but it is much easier to use CLI for automation. Besides the submission form, the webpage also provides an interpretation of results in a consistent way between all batteries. Three types of assessments OK, SUSPECT and FAIL are assigned to batteries based on the amount of failed test. The results of individual tests within batteries are evaluated with significance level $\alpha = 0.01$.

RTT contains tests from three software tools: NIST STS [3], Dieharder [6] and TestU01 [10]. We used all statistical batteries which RTT provides, except for Big Crush from TestU01 because it requires at least 60GB of data per run, which would be too demanding for resources.

3.4.2 Testing with BoolTest

TODO: Using metacentrum? Info about runtime...

3.5 Investigated pseudo-random generators

In this section, we will present all functions which were incorporated to CryptoStreams and then investigated for indistinguishability from the truly random stream.

3.5.1 Pure pseudo-random generators

The first part of this thesis is the addition to CryptoStreams and analysis of pure (do not use any cryptographic primitive to generate pseudo-randomness) pseudo-random number generators. It was not possible to round reduce those generators because the implementation does not provide it. Hence we only weakened them with seed with specific randomness properties. This component is quite small as we added together only six generators. The reason why we added such a small amount of generator is that we have not found any source of generators which would offer generators in a way it would be easy to incorporate to CryptoStream.

One of the sources of generators was a TestU01 [10] project. It contains a high number of generators, but they offer just implementations without parameters set up. So we needed to find out those parameters ourselves, and it was sometimes difficult to choose correctly. Reasons for difficulties was, for example, selecting parameters so that output was long enough to fill 8 bytes of data, absence of literature for less widely used generators, etc. We have taken over three generators and also included some basic tests. All generators are appropriately described in TestU01 user guide [25].

Linear congruential generator. Definition of this generator is shown in Formula 3.5.2. Chosen parameters are taken from [26] and values are $a = 4645906587823291368$, $c = 0$ and $m = 9223372036854775783$. We have chosen as big parameter m as possible because the generator is outputting values modulo this parameter, this means values are always less than this number. Since returning value from the generator is always 8 bytes long and the number 9223372036854775783 have few upper bits binary zeroes, also outputting value will always have those bits binary zero. For that reason, we cut those bits to not have any interference caused by too many zeroes in statistical tests. This is the main reason for adding such a small number of generators in this thesis because it would require too much configuration and testing to be sure the generators are working properly.

Multiple recursive generator. Another linear generator, which is based on a very similar principle as LCG, with the difference that it combines data from more than one previous run. The definition is shown in Formula 3.5.1. Values of parameters are $k = 2$, $a_1 = 2975962250$, $a_2 = 2909704450$ and $m = 9223372036854775783$ [27]. We needed to cut upper binary zeroes too.

Xorshift generator. The generator is based on *xor* and *shift* operations [28]. We have chosen version which is created by function `uxorshift_CreateXorshift13` and requires no additional parameters, more information in [25] on page 50.

Formula 3.5.1. *Multiple recursive generator (MRG) [25]:*

$$x_n = \Big(a_1 \times x_{(n-1)} + ... + a_k \times x_{(n-k)}\Big) \mod m \tag{3.1}$$

Where k, $a_1..a_k$ and m are parameters of the generator hardcoded within CryptoStreams, X_{n-l} is an output of a run $n - l$ where n is current run and l is a number between 1 and k. The seed represents initial values of X_1 to X_k

Formula 3.5.2. *Linear congruential generator (LCG) [25]:*

$$x_{n+1} = (a \times x_n + c) \mod m \tag{3.2}$$

Where a, c, m are the parameters of the generator hardcoded within CryptoStreams, X_{n-1} is a previous value and X_0 is the seed.

The second source of pseudo-random streams is the standard library of C++. Unlike TestU01 it contains generators including parameters. It is less complicated to take over this code as it is enough to use `include` directive. The only disadvantage of this source is it contains only three pseudo-random generators. List of generators is following.

Linear congruential generator [4]. The same generator as we have taken over from TestU01. Definition is shown in Formula 3.5.2. Used parameters are $a = 48271$, $c = 0$ and $m = 2147483647$. As you can see numbers are much smaller, than those we have chosen for the generator from TestU01. The reason is that this generator outputs only 4 Bytes instead of 8 Bytes.

Mersenne Twister [5]. The generator was developed by Makoto Matsumoto and Takuji Nishimura [9]. However, it does not produce cryptographically secure random numbers [29].

Substract with carry [6]. This type of generator was introduced by George Marsaglia and Arif Zaman [30]. Formula 3.5.3 shows definition of this generator.

Formula 3.5.3. *Substract with carry [31]:*

$$x_n = (x_{n-S} - x_{n-R} - cy(n-1)) \mod M \tag{3.3}$$

Where

$$cy(n) = \begin{cases} 1, & \text{if } x_{n-S} - x_{n-R} - cy(n-1) < 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

and S,R are parameters hardcoded in CryptoStreams. x_k represents k-th output of the generator. The seed represents initial values of x_1 to x_k where $k = maxR, S$.

3.5.2 Generators based on lightweight cryptographic primitives

The other part contains block ciphers taken over from project FELICS [32] developed by Daniel Dinu and his group at the University of Luxembourg. This project is conducting a performance analysis of lightweight functions that are intended for embedded devices. We have taken over only the C++ implementation of functions, as we were not interested

4. `https://en.cppreference.com/w/cpp/numeric/random/linear_congruential_engine`
5. `https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine`
6. `https://en.cppreference.com/w/cpp/numeric/random/subtract_with_carry_engine`

in implementations optimized for other architectures. We needed to implement round reduction of functions ourselves as the project contained only full round implementation. However, provision of round reduction was mostly straightforward as functions were prepared with round reduction in mind and we only needed to replace constant in a loop with a variable which is configurable from CryptoStreams.

Besides the main loop, functions mostly contain also key scheduling, initial and final part. Key scheduling is taking care of the creation of round keys based on a provided key. Initial and final part serves for initialization and finalization of the process. We do not round-reduce any of those parts as we wanted to avoid some memory problems like uninitialized or wrongly cleaned memory.

Table 3.3 contains all investigated functions including some basic information about them. Our intention was also adding two test scenarios, for testing correctness of implementation, for each added function.

The first scenario is testing all functions with test vectors in full number of rounds. Test vectors were taken over from project FELICS.

The aim of the second test scenario is testing round reduction by doing encryption followed by decryption with function limited to all possible number of rounds. However, we were not able to make this scenario work for 6 functions, they are marked with ✗ in last column in Table 3.3. We admit the round-reduction might be broken for those functions; however we included them in the statistical testing and round-reduction behave correctly from randomness point of view. Lower rounds are not passing statistical tests, and an addition of more rounds improves statistical properties of the output. The failure of the Encrypt-Decrypt test may be caused by the fact that we do not reduce key scheduling, initial and final parts. This reason would correspond with observed behavior in which the functional part of round-reduction is broken (Encrypt-Decrypt test) while output's randomness properties are correct (statistical analysis).

Function	Round	Block size	Key size	Encrypt-Decrypt test
Chaskey [33]	16	16	16	✓
Fantomas [34]	12	16	16	✓
HIGHT [35]	32	8	16	✗
LBlock [36]	32	8	10	✗
LEA [37]	24	16	16	✗
LED [38]	48	8	10	✓
Piccolo [39]	25	8	10	✓
PRIDE [40]	20	8	16	✗
PRINCE [41]	12	8	16	✓
RC5-20 [42]	20	8	10	✓
RECTANGLE-K80 [43]	25	8	16	✗
RECTANGLE-K128 [43]	25	8	16	✗
RoadRunneR-K80 [44]	10	8	10	✓
RoadRunneR-K128 [44]	12	8	16	✓
Robin [34]	16	16	16	✓
RobinStar [34]	16	16	16	✓
SPARX-B64 [45]	8	8	16	✓
SPARX-B128 [45]	8	16	16	✓
TWINE [46]	35	8	10	✓

Table 3.3: List of all investigated functions, where sizes are given in Bytes. Including information whether encrypt decrypt test passed.

4 Results of evaluation of statistical randomness properties

Bibliography

[1] Howard M. Heys. "A TUTORIAL ON LINEAR AND DIFFERENTIAL CRYPTANALYSIS". In: *Cryptologia* 26.3 (2002), pp. 189–221. DOI: 10.1080/0161-110291890885. eprint: https://doi.org/10.1080/0161-110291890885. URL: https://doi.org/10.1080/0161-110291890885.

[2] Lawrence E. Bassham III et al. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Tech. rep. Gaithersburg, MD, United States, 2010.

[3] National Institute for Standards and Technology. *Statistical Test Suite*. Version 2.1.1. 1997. URL: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software (visited on 05/19/2017).

[4] National Institute for Standards and Technology. *Statistical Test Suite*. Version 2.1.1. 1997. URL: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software (visited on 05/19/2017).

[5] Marek Sýs, Zdeněk Říha, and Vashek Matyáš. "Algorithm 970: Optimizing the NIST Statistical Test Suite and the Berlekamp-Massey Algorithm". In: *ACM Trans. Math. Softw.* 43.3 (Dec. 2016), 27:1–27:11. ISSN: 0098-3500. DOI: 10.1145/2988228. URL: http://doi.acm.org/10.1145/2988228.

[6] Robert G. Brown. *Dieharder: A Random Number Test Suite*. Version 3.31.1. Duke University Physics Department. 2004. URL: http://www.phy.duke.edu/~rgb/General/dieharder.php (visited on 10/07/2018).

[7] George Marsaglia. *Diehard Battery of Tests of Randomness*. Floridan State University. 1995. URL: https://web.archive.org/web/20120102192622/www.stat.fsu.edu/pub/diehard/ (visited on 10/07/2018).

[8] Ľubomír OBRÁTIL. "The automated testing of randomness with multiple statistical batteries [online]". Master's thesis. Masaryk University, Faculty of Informatics, Brno, 2017 [cit. 2018-09-29]. URL: Available%20from%20WWW%20%3Chttps://is.muni.cz/th/uepbs/%3E.

[9] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: http://doi.acm.org/10.1145/272991.272995.

[10] Pierre L'Ecuyer and Richard Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (Aug. 2007). ISSN: 0098-3500. DOI: 10.1145/1268776.1268777.

[11] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[12] Marek Sýs, Dušan Klinec, and Petr Švenda. "The Efficient Randomness Testing using Boolean Functions". In: *14th International Conference on Security and Cryptography (Secrypt'2017)*. SCITEPRESS, 2017, pp. 92–103. ISBN: 978-989-758-259-2.

[13] Petr Švenda et al. *CryptoStreams. Tool for generation of data from round-reduced cryptoprimitives*. Centre for Research on Cryptography and Security, Masaryk University. 2017. URL: https://github.com/crocs-muni/CryptoStreams (visited on 08/20/2018).

[14] Petr Švenda et al. *EACirc. Framework for automatic search for problem solving circuit via Evolutionary algorithms*. Centre for Research on Cryptography and Security, Masaryk University. 2012. URL: `https://github.com/crocs-muni/eacirc` (visited on 08/25/2018).

[15] Ondrej DUBOVEC. *Automatické hledání závislostí u kandidátních hašovacích funkcí SHA-3 [online]*. Bachelor's thesis. 2012 [cit. 2018-08-30]. URL: `Available%20from%20WWW%20%3Chttps://is.muni.cz/th/ps8b6/%3E`.

[16] Matej PRIŠŤÁK. "Automatické hledání závislostí u proudových šifer projektu eStream [online]". Master's thesis. Masaryk University, Faculty of Informatics, Brno, 2012 [cit. 2018-08-30]. URL: `Available%20from%20WWW%20%3Chttps://is.muni.cz/th/mwbpn/%3E`.

[17] Martin UKROP. "Randomness analysis in authenticated encryption systems [online]". Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno, 2016 [cit. 2018-09-02]. URL: `Dostupn%C3%A9%20z%20WWW%20%3Chttps://is.muni.cz/th/rcl3c/%3E`.

[18] CAESAR committee. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. 2013. URL: `http://competitions.cr.yp.to/caesar-call.html` (visited on 09/02/2018).

[19] Karel KUBÍČEK. "Optimisation heuristics in randomness testing [online]". Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno, 2017 [cit. 2018-09-05]. URL: `Dostupn%C3%A9%20z%20WWW%20%3Chttps://is.muni.cz/th/bhio1/%3E`.

[20] Tamás RÓZSA. *Kvalita výstupu pseudonáhodných kryptografických funkcí [online]*. Bakalářská práce. 2018 [cit. 2018-09-05]. URL: `Dostupn%C3%A9%20z%20WWW%20%3Chttps://is.muni.cz/th/mdvro/%3E`.

[21] M.E. O'Neill. *PCG, A Family of Better Random Number Generators. PCG is a simple and fast statistically good PRNG*. 2015. URL: `http://www.pcg-random.org/` (visited on 09/16/2018).

[22] Karel Kubíček et al. *EACirc – documentation wiki. Streams for data manipulation*. Centre for Research on Cryptography and Security, Masaryk University. 2018. URL: `https://github.com/crocs-muni/CryptoStreams/wiki/Streams` (visited on 09/11/2018).

[23] *Federal Information Processing Standards Publication (FIPS 197). Advanced Encryption Standard (AES)*. 2001.

[24] Joan Daemen and Vincent Rijmen. "AES proposal: Rijndael". In: (1999).

[25] Pierre L'Ecuyer and Richard Simard. *TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators*. 2007. URL: `http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf`.

[26] Pierre L'Ecuyer. "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure". In: *Math. Comput.* 68.225 (Jan. 1999), pp. 249–260. ISSN: 0025-5718. DOI: `10.1090/S0025-5718-99-00996-5`. URL: `http://dx.doi.org/10.1090/S0025-5718-99-00996-5`.

[27] Pierre L'Ecuyer, François Blouin, and Raymond Couture. "A Search for Good Multiple Recursive Random Number Generators". In: *ACM Trans. Model. Comput. Simul.* 3.2 (Apr. 1993), pp. 87–98. ISSN: 1049-3301. DOI: `10.1145/169702.169698`. URL: `http://doi.acm.org/10.1145/169702.169698`.

[28] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 008.i14 (2003). URL: `https://EconPapers.repec.org/RePEc:jss:jstsof:v:008:i14`.

[29] Krister Sune Jakobsson. *Theory, methods and tools for statistical testing of pseudo and quantum random number generators*. 2014.

[30] George Marsaglia and Arif Zaman. "A New Class of Random Number Generators". In: *Ann. Appl. Probab.* 1.3 (Aug. 1991), pp. 462–480. DOI: `10.1214/aoap/1177005878`. URL: `https://doi.org/10.1214/aoap/1177005878`.

[31] Wikipedia contributors. *Subtract with carry — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Subtract_with_carry&oldid=719857744`. [Online; accessed 28-September-2018]. 2016.

[32] Daniel Dinu et al. "Felics–fair evaluation of lightweight cryptographic systems". In: *NIST Workshop on Lightweight Cryptography*. Vol. 128. 2015.

[33] Nicky Mouha et al. *Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers*. Cryptology ePrint Archive, Report 2014/386. `https://eprint.iacr.org/2014/386`. 2014.

[34] Vincent Grosso et al. "LS-designs: Bitslice encryption for efficient masked software implementations". In: *International Workshop on Fast Software Encryption*. Springer. 2014, pp. 18–37.

[35] Deukjo Hong et al. "HIGHT: A New Block Cipher Suitable for Low-Resource Device". In: *Cryptographic Hardware and Embedded Systems - CHES 2006*. Ed. by Louis Goubin and Mitsuru Matsui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 46–59. ISBN: 978-3-540-46561-4.

[36] Wenling Wu and Lei Zhang. "LBlock: A Lightweight Block Cipher". In: *Applied Cryptography and Network Security*. Ed. by Javier Lopez and Gene Tsudik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 327–344. ISBN: 978-3-642-21554-4.

[37] Deukjo Hong et al. "LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors". In: *WISA*. 2013.

[38] Jian Guo et al. "The LED Block Cipher". In: *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*. CHES'11. Nara, Japan: Springer-Verlag, 2011, pp. 326–341. ISBN: 978-3-642-23950-2. URL: `http://dl.acm.org/citation.cfm?id=2044928.2044958`.

[39] Kyoji Shibutani et al. "Piccolo: An Ultra-Lightweight Blockcipher". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 342–357. ISBN: 978-3-642-23951-9.

[40] Martin R. Albrecht et al. "Block Ciphers – Focus on the Linear Layer (feat. PRIDE)". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 57–76. ISBN: 978-3-662-44371-2.

[41] Julia Borghoff et al. "PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 208–225. ISBN: 978-3-642-34961-4.

[42] Ronald L. Rivest. "The RC5 encryption algorithm". In: *Fast Software Encryption*. Ed. by Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 86–96. ISBN: 978-3-540-47809-6.

[43] WenTao Zhang et al. “RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms”. In: *Science China Information Sciences* 58.12 (Dec. 2015), pp. 1–15. ISSN: 1869-1919. DOI: 10.1007/s11432-015-5459-7. URL: https://doi.org/10.1007/s11432-015-5459-7.

[44] Adnan Baysal and Sühap Şahin. “RoadRunneR: A Small and Fast Bitslice Block Cipher for Low Cost 8-Bit Processors”. In: *Lightweight Cryptography for Security and Privacy*. Ed. by Tim Güneysu, Gregor Leander, and Amir Moradi. Cham: Springer International Publishing, 2016, pp. 58–76. ISBN: 978-3-319-29078-2.

[45] Daniel Dinu et al. “Design Strategies for ARX with Provable Bounds: Sparx and LAX”. In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 484–513. ISBN: 978-3-662-53887-6.

[46] T Suzaki et al. “TWINE: A Lightweight, Versatile Block Cipher”. In: (Jan. 2011).

A Glossary