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 2.3 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,

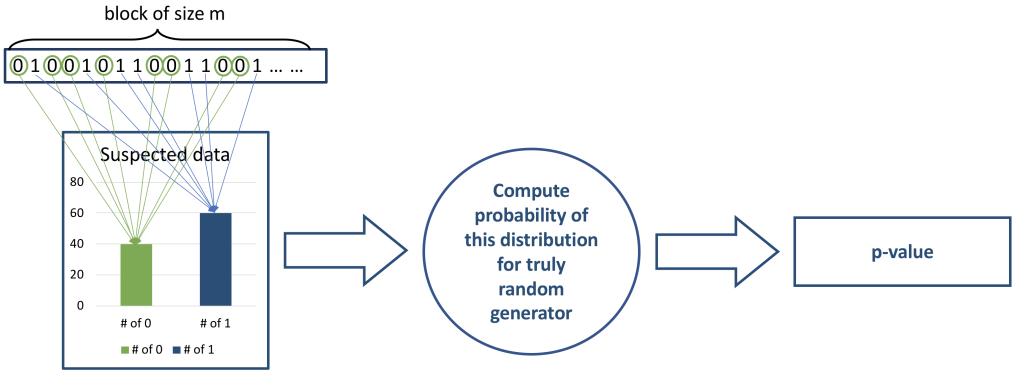

Figure 2.1: Monobit test from high level point of view.

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.

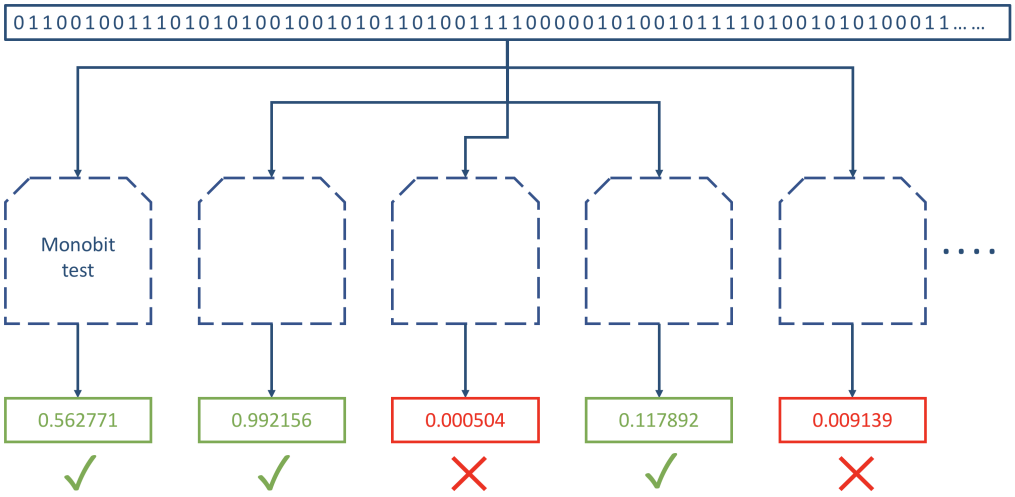

Figure 2.2: High level view of software tool run.

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 in the form of boolean function. The process starts with dividing sequence into blocks with size m. Then 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 quite easy process. Simply take all blocks and for each compute result of boolean function where x_k is k-th bit of the block. After that 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. That is

why they starts with brute-force approach, where all possible boolean function of type $f(x_1, x_2, \cdots, x_m) = x_i$ for $\forall i \in \{1, 2, \cdots, m\}$. [12]

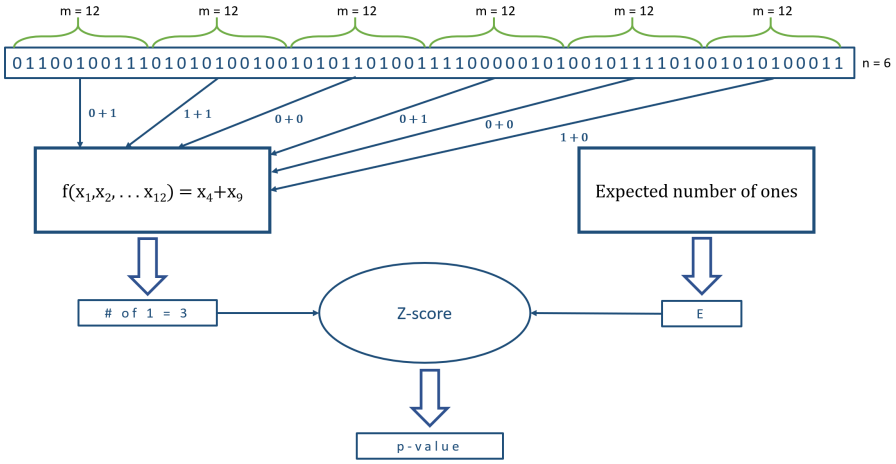

Figure 2.3: Example of BoolTest run for sequence of of size 72 bits with block size 12 and number of blocks 6.

Statistical batteries/test suite, how it works, what statistical batteries we know, comparison of results (between batteries) based on previous experiments ....

Bool test, simple, yet strong test, based on boolean function as distinguisher between truly random data and investigated data

3 Introduction of CryptoStreams tool

CryptoStreams tool is written in C++ language and is developed and maintained by team of people[1] at the Centre for Research on Cryptography and Security, Masaryk University [13]. The tool is used to generate output data streams from parametrized cryptographic functions. Each stream is configurable with resulting size and with several configuration options per individual streams such as seed, plaintext or key type.

3.1 History

Initial implementation of CryptoStreams project was part of tool EACirc [14] which is a tool for automatic randomness testing based on genetic programming. At that moment it served only as a provider of data to EACirc and was not possible to use it separately outside of this project. After some time we decided that CryptoStreams might be potentially interesting also as a separate tool. That is why EACirc-streams project was introduced in 2017 and then in 2018 EACirc sub-name was completely abandoned and project was renamed to nowadays name, CryptoStreams.

3.2 Idea

Main idea behind CryptoStreams is easy production of data from crypto-primitives which are somehow reduced in complexity, either by limiting rounds or by providing them input with bad randomness properties. The biggest advantage is easy incorporation of new functions to CryptoStreams and after this step it is trivial to obtain data which were produced by function in somehow limited scenario. After obtaining data it is possible to do any type of investigation over those data. For example in this thesis we will conduct statistical analysis with 7 statistical batteries of tests and also with tool called Booltest [12]. Notice, that addition of new analysis tool requires no additional implementation on 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 4 types of cryptographic primitives: block ciphers, hash functions, stream cipher and pseudo-random number generators. Table 3.1 shows counts of function of corresponding types. Very 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 CAESAR competition [18]. Well known block ciphers like AES, DES etc. were added by Karel Kubíček [19] and Tamás Rózsa [20] in their theses. There are also lot of other cryptographic primitives added outside of theses or papers.

---

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

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

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

***Streams* outputing data of exact 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. Random (pseudo-random) streams also belong to this category. Figure 3.1 shows example of schema of such *stream*.

Figure 3.1: Example of one call from exact stream.

Manipulating *streams* are configured with one or more inputs and manipulate them in some desired way. For example, *repeating stream* is repeating one output specified number of times before generating 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.

Figure 3.2: Example of one call from manipulating stream, specifically 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 initializa-

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

Figure 3.3: Example of one call from 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 provide them seed with bad randomness properties. 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.

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

Notice that all inputs are in the form of another *streams*. Receiving *stream* is deciding how many data and when will use from *streams* on its input. For example if receiving *stream* is cryptographic primitive type it requests new plaintext for each chunk, but key may be generated only once in the beginning of generation. For better imagination how *streams* really works, Figure 3.5 contains schema of whole run of CryptoStreams. The middle block is the most important one. It is *cryptographic primitive stream* and on its input it has 3 *exact streams*.

3.3.1 Configuration

All necessary configuration within one run is achieved by `JSON` file. Example of such configuration file can be found in Figure 3.6 which will result in file of size 8GB, which contains output from function CHASKEY limited to 5 rounds. Key is generated pseudo-randomly using PCG32 [21] in the beginning of run and then used for generation of each

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

output chunk. *Counter stream* is used as a plaintext. Schema of this configuration is shown in Figure 3.5.

Whole run of generator is deterministic as it is using pseudo-random generator. That is why configuration contains also seed. All values which are generated pseudo-randomly are based on this seed, in other words if you run CryptoStreams twice with the same configuration resulting files will be equal. Notice that resulting file size is derived from `chunk_size` in Bytes and `chunk_count`. All possible options how to configure CryptoStreams can be found in project documentation [22].

```
{
  "chunk_count": 500000000,
  "chunk_size": 16,
  "file_name": "AES_r05_b16.bin",
  "seed": "1fe40505e131963c",
  "stream": {
    "type": "block",
    "algorithm": "AES",
    "round": 5,
    "block_size": 16,
    "key_size": 16,
    "iv_size": 16,
    "init_frequency": "only_once",
    "plaintext": {
      "type": "counter"
    },
    "key": {
      "type": "pcg32_stream"
    },
    "iv": {
      "type": "false_stream"
    }
  }
}
```

8GB in total (500m * 16B)

Stream from block cipher AES

Number of rounds

Key is generated only once

Definition of plaintext stream

Definition of key stream

Definition of iv stream

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

3.3.2 Testing of streams

Our statistical analysis relies on the fact that data which comes from CryptoStreams are correct and truly comes from cryptographic primitives. That is why it is very important to have proof that our implementation of each cryptographic primitive is correct one. To have this proof we introduced testsuite which contains various number of tests per individual *stream*. Since CryptoStreams contains huge amount of *streams*, we have not added tests for all of them, instead of that we added tests for those which are used most frequently. It is also required to have tests in order to add new *stream* into CryptoStreams. All results in this thesis are based on *streams* which are properly tested.

Regarding implementation details we are using Google Test [2] framework as testing backend. Wee are using tool Travis CI[3] for continuous integration. It means new changes are not approved unless it passes all tests including newly added tests.

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 full number of rounds as we were not able to find test vectors for round limited version. For lightweight cryptographic primitives based *streams* we added also *encrypt-decrypt* test for all supported rounds. This test 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. Coverage with tests is very good for block ciphers, all 42 functions are tested with test vectors.

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

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.

Pseudo-random generators *streams* are quite hard to test, we have not found any test vectors. Nonetheless, we at least added test for linear generators, we tried to implement succession of numbers in tests and then compare whether we are getting same numbers from generator implementation. 4 tests out of 6 included in CryptoStreams are tested, however one test is testing only functionality without checking whether output is correct.

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

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 number of functions covered with tests.

3.4 Conducted experiments

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

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

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

Round reduction. Almost each cryptographic function is build so that it performs very same sequence of operations multiple times, mostly in a loop. Number of times sequence should be performed is denoted by term *number of rounds*. For example, AES [23] has 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.

	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 taken from [23]*, word size is 32 bits.*

Creators of each cryptographic function specify how many rounds should be used in order 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 number of rounds. Important indicator during this process is called *security margin*. Security margin denotes the rate between round vulnerable to some cryptanalysis attack and the full number of rounds. For example authors of AES were unable to find any practical shortcut attack (any attack which is more efficient than exhaustive key search) for more than 6 rounds for AES with key length 128 bits. That is why they recommend to use 10 where 4 is security margin [24]. Notice that *security margin* is not some general notion for 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 indistinguishably of output of cryptographic primitive limited to all possible rounds from random stream.

Plaintext ciphertext stream produce pairs of input and output to function. With statistical testing performed on such output, it is possible to investigate dependency between plaintext and ciphertext. However we need to be careful with deciding what input we choose since it is part of output. It must have perfect randomness properties otherwise it may cause some interference in testing result.

3.4.1 Testing with Randomness testing toolkit

For running experiments we are using 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 higher amount of experiments, since it is possible to automate it, 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 submission the webpage provides also interpretation of results in a consistent way between all batteries. 3 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 truly random stream.

3.5.1 Pure pseudo-random generators

First part of this thesis is 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 do not provide it. Hence we only weakened them with seed with bad randomness properties. This component is quite small as we added together only six generators. The reason why we added so 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.

First source of generators was TesU01 [10] project. It contains big amount of generators, but they offer just implementation without parameters set up. So we needed to find out

those parameters ourselves and it was quite hard to choose correctly. We have taken over 3 generators and also included some basic tests. All generators are properly 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 $m = 92233720368547775783$. We tried to choose as big parameter m as possible because generator is returning values modulo this parameter, this means outputting values are always less than this number. Since returning value from generator is always 8 Bytes long and number 92233720368547775783 have few upper bits binary zeroes, also outputting value will always have those bits binary zero. This is why we needed to cut those bits in order to not have some interference caused by this in statistical tests. This is the main reason, why we have not added more generators in this thesis because it requires too much configuration and testing to be sure the generators are working properly.

Multiple recursive generator. Another linear generator, which is based on 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 $m = 92233720368547775783$ [27]. Also 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]:*

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

Where k, a_1..a_k and m are parameters of generator hardcoded within CryptoStreams, X_{n-l} is output of run $n - l$ where n is current run and l is number between 1 and k. 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 parameters of generator hardcoded within CryptoStreams, X_{n-1} is previous value and X_0 is seed.

The second source of streams is standard library of C++. Unlike TestU01 it contains generators including parameters. It is much less difficult to take over this code as it is enough to simply use `include` directive. The only disadvantage of this source is it contains only 3 pseudo-random generators. List of generators is following.

Linear congruential generator [4]**.** Very 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 generator from TestU01. The reason is that this generators outputs only 4 Bytes instead of 8 Bytes.

Mersenne twister [5]**.** The generator is 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. *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 generator. 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 University of Luxembourg. This project is conducting performance analysis of lightweight functions that are intended for embedded devices. We have taken over only C++ implementation of functions, as we were not interested in implementations optimized for other architectures. Also we needed to implement round reduction of functions ourselves as the project contained only full round implementation. However, provision of round reduction was not hard, mostly functions were prepared with round reduction in mind and we needed to only replace constant in loop with 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 key based on a encryption key. Initial and final part serves for initialization and finalization of 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 each added function. First scenario is testing correct implementation with test vectors. FELICS project contains test vectors for each function, so we used those. 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 each function. Function without this test are marked with *no* in last column in Table 3.3. The main reason why we were not able to make this test work is that FELICS project is not build to support it and we were not able to modify it in a way it would work. This may be caused also by non reducing of key scheduling, initial and final parts. However, the most important is that the test is passing in full round version of function and it passes for all functions. We do not consider round reduction broken in case the test does not pass.

Function	Round	Block size	Key size	Encrypt Decrypt test
Chaskey [33]	16	16	16	yes
Fantomas [34]	12	16	16	yes
HIGHT [35]	32	8	16	no
LBlock [36]	32	8	10	no
LEA [37]	24	16	16	no
LED [38]	48	8	10	yes
Piccolo [39]	25	8	10	yes
PRIDE [40]	20	8	16	no
PRINCE [41]	12	8	16	yes
RC5-20 [42]	20	8	10	yes
RECTANGLE-K80 [43]	25	8	16	no
RECTANGLE-K128 [43]	25	8	16	no
RoadRunneR-K80 [44]	10	8	10	yes
RoadRunneR-K128 [44]	12	8	16	yes
Robin [34]	16	16	16	yes
RobinStar [34]	16	16	16	yes
SPARX-B64 [45]	8	8	16	yes
SPARX-B128 [45]	8	16	16	yes
TWINE [46]	35	8	10	yes

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