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 Pseudo-Random number generators

Random generator is everything that produces a random outcome. In the real world, it might be a coin, dice, etc. In computer science, we distinguish between two types of randomness generators: truly random number generators (TRNG) and pseudo-random number generators (PRNG). Deterministic process cannot produce true randomness – some non-determinism is required, also known as *entropy*. Interaction of computer with some physical components, such as input devices (keyboard, mouse, microphone, etc.), may be used as a source of non-determinism. However, it may be the case, those components produce less amount of entropy than computer requires; for example if an attacker is able to control those devices. Therefore we cannot rely only on this source of random data. For that reason, PRNGs were introduced. The idea behind PRNGs is to deterministically (using an algorithm) produce a seemingly random (pseudo-random) sequence based on small amount of an entropy (seed). Notice that the sequence can be generated by anyone who knows the original algorithm and the seed. [1]

2.1 Properties of PRNGs

An algorithm from PRNG is iteratively called, and each call produce output of fixed size. The PRNG keeps internal state S which fully determines value of next output. One i-th iteration consists of two steps: generation of next output X_{i+1} based on current state S_i and obtaining S_{i+1} by modification of S_i. The seed commonly represents the initial state S_0.

In some cases, the internal state S_i matches the previous output X_{i-1}. The example of such generator is shown in Figure 2.1. In this case the generator can be described by deterministic function f, which is invoked to generate i-th output using the formula $X_i = f(X_{i-1})$.

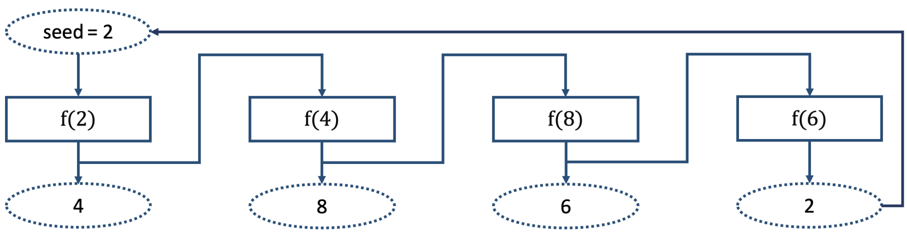

Figure 2.1: Example of a schema for the linear congruential pseudo random number generator with the function $f(X) \equiv (2 \times X \mod 10)$. *The period for value of seed 2 is four.*

Since the size of state is fixed, there are finitely many options of inputs and corresponding outputs what eventually leads to repetitions. The number of iterations before the first duplicity is called the *period*. The length of the period is also dependent on a seed. Generators are created so that the *period* is maximized for any seed. When choosing PRNG for an application the *period* should be taken into account; it must be chosen so that the period is never reached. However, this can be achieved by regular reseeding with truly random data. A simple example of a generator run with period four is shown in Figure 2.1.

In fact, modern PRNGs are periodically modifying internal state using truly random data – those are called hybrid PRNGs. The security is improved by possibility to recover from compromised state, by changing the internal state in a way the attacker cannot predict. However, hybrid PRNGs are deterministic only between two reseedings and consume more truly random data. [2]

Since whole sequence generated by PRNG can be determined from the seed, it is important, from security point of view, to keep the seed secret and generate it so that it is unpredictable and truly random.

Sequence produced by PRNG must look random. This means it should also pass all empirical tests of randomness with high probability (covered in Section 2.6). [3]

To use PRNG within sensitive applications (such as a cryptographic application), it needs to fulfill one more requirement. It must be unfeasible to compute any information about the previous/following output given that the attacker knows part of the generated sequence, including algorithm and hardware specification. Such generators are called cryptographically secure pseudo-random generators (CSPRNG). [3]

2.2 Constructions techniques

In this thesis, we distinguish between following two categories of PRNGs.

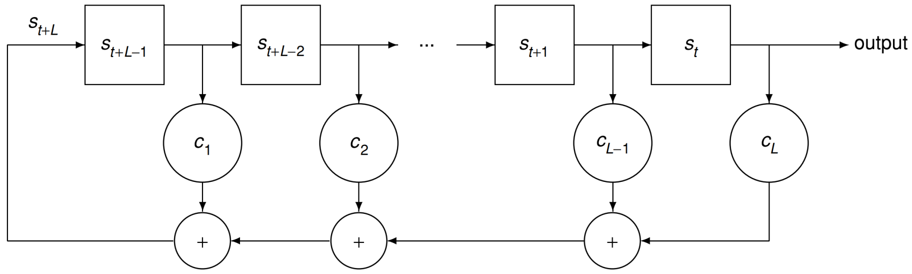

Figure 2.2: Scheme of Linear Feedback Shift Register. Variable s represents bits of the register and c is parameter of generator. [4]

Simple (pure) generators. The function f is mostly some simple mathematical formula. A commonly known generator of this type is the linear congruential generator (LCG). The formula is following. [5]

$$x_{n+1} = (a \times x_n + c) \mod m \tag{2.1}$$

Where a, c, m are fixed parameters for specific generator. Simple example of LCG with $a = 2$, $c = 0$ and $m = 10$ is shown in Figure 2.1.

Another basic principle which is a base for lots of generators is a Linear Feedback Shift Register (LFSR). LFSRs are also suitable for hardware implementation, what is one of its advantages. The principle is based on a register which is right shifted every iteration, where rightmost bit is output. The new value for the leftmost bit is determined based on values of other chosen bits within the register. Figure 2.2 shows a schema for LFSR. The formula for computation new leftmost bit is following:

$$s_{t+L} = \sum_{i=1}^{L} c_i s_{t+L-i} \qquad \forall t \geq 0, \tag{2.2}$$

where t determines a time, s stands for register bit values, and c is a parameter which decides whether i-th bit is used for determination of the value of the new bit (left-most). L represents the bit size of the register. [4]

Makoto Matsumoto and Takuji Nishimura introduced Mersenne Twister pseudo-random generator [6] in 1998. It is an extended version of TGFSR [7] (twisted generalized feedback shift register) algorithm. Despite the long period of length $2^{19937} - 1$, Mersenne twister itself is not cryptographically secure PRNG. However, it is possible to create CSPRNG from MT, for example by modifying output by a hashing algorithm. One of the usages suitable for this generator is within the Monte Carlo simulations.

PRNGs based on cryptographic primitives. Even though cryptographic primitives, such as block ciphers or hash functions, are used for different purposes than PRNGs, they have a partly similar objective – to produce a seemingly random sequence from an input. Therefore, each cryptographic primitive can be easily used for the generation of pseudo-random sequences. The security of resulting generator should be proportional to the security of used primitive. The primitive should also guarantee the unfeasibility to compute next/previous value given a part of the sequence, which means those generators are mostly cryptographically secure. In this thesis, we rely on this fact. Therefore, we are testing cryptographic primitives directly instead of testing generators which encapsulates those primitives.

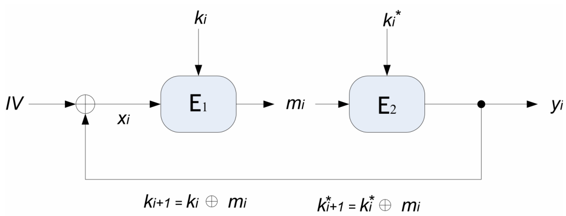

Figure 2.3: Example of PRNG created from block cipher taken from [8].

There are lots of options for creating a PRNG based on a primitive. For block ciphers, one needs to solve what to input to the cipher in each iteration (for example using counter from the value of seed, or chain iterations) and what to use as a key (either use a fixed key or determine the key from an internal state). Christophe Petit et al. studied [8] a security of a generator shown in Figure 2.3. The studied generator is using two instances of a block cipher. Keys are changed for every iteration and output from i-th iteration is used as an input to $(i+1)$-th iteration.

ANSI X9.17/ANSI X9.31 (Figure 2.4) [10] introduced standardized way how to turn block cipher into PRNG. This generator could be used with any cryptographic primitive as a provider of pseudo-randomness, however, recommended is 3-DES and AES [11].

Regarding hash functions, Russell Impagliazzo et al. proved [9] that the existence of one-way functions is necessary and sufficient for the existence of pseudo-random generators.

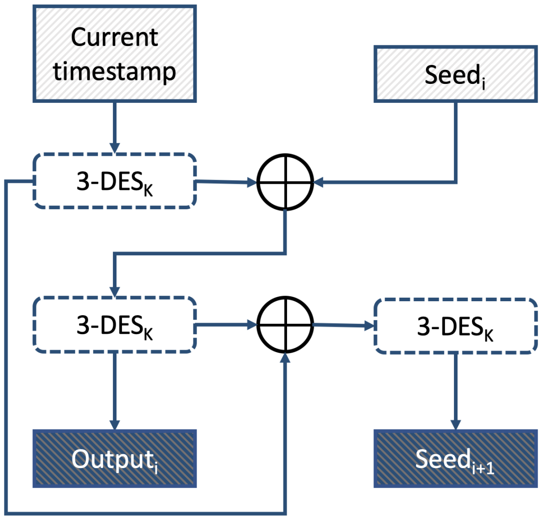

Figure 2.4: Scheme of ANSI X9.17 PRNG [10, 12].

2.3 Usage of PRNGs in cryptography

CSPRNGs in cryptography are used for several purposes, for example, for generation of keys, initialization vectors, but also for encryption and decryption within symmetric cryptography. In this section, closer look to the last usage is presented.

Shannon theorized [13] that it is possible to achieve a perfect secrecy only in case a number of possible keys is greater or equal to a number of possible messages. Perfect secrecy means that there is no information about plaintext in ciphertext. The one-time pad cryptosystem achieves this property by having the key longer than message. For encryption, it uses simple bit by bit `XOR` of plaintext with a large truly random non-repeating sequence as a key. However, distribution of such key is complicated, and in case there would be schema for exchange of such key in secure way, it could be used for sending message itself. [3]

Stream ciphers were introduced with a similar principle in mind as the one-time pad. However, instead of using a long truly random sequence as the key, an output from CSPRNG is used. The ciphertext C is created from plaintext P and keystream (generated from CSPRNG) K as follows: $C_i = P_i \oplus K_i$, for each bit in plaintext. For an encrypted communication both sides need to know the key, which is nothing else than a seed for CSPRNG, to produce the same keystream (for more information see [3]). Stream ciphers are useful for long streams (such as audio or video streams), especially for their zero error propagation. This means if one bit is flipped during a data transfer over the network, only one bit is flipped in decrypted plaintext. For example, if the same situation occurs when using a block cipher in CBC mode whole block is broken after decryption.

However, block ciphers can act similarly to stream ciphers. For example, in Output Feedback mode (OFB) or Counter mode (CTR) of operation for block ciphers, the plaintext never go through the cipher itself. Cryptographic primitive, in this case, serves only as a generator of a keystream which is `XOR`-ed with plaintext, similarly like in stream ciphers. Figure 2.5 shows scheme of CTR mode.[14]

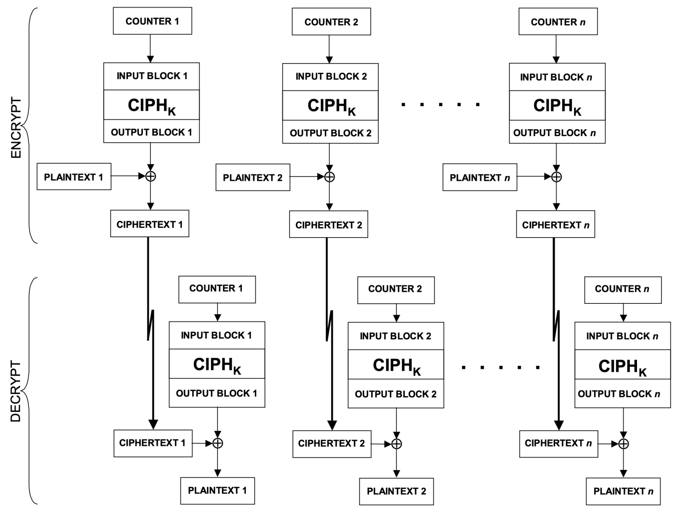

Figure 2.5: Scheme of CTR mode of operation for block ciphers. [14]

2.4 Typical attacks

In this section, three theoretical types of attacks against PRNGs are presented which were originally explained by Kelsey et al. in [12], followed by practical example of attack which were conducted on real generator. Some of those types are similar to attacks conducted within this thesis, see Section 3.4.

Direct Cryptanalytic Attack – analysis of output without any other knowledge of PRNG. For example, distinguishing between truly random data and output from PRNG, more in Section 2.6.

Input-Based Attack – an attacker is trying to take advantage of knowing (known-input) or controlling (chosen-input, replayed-input) input to be able to distinguish between outputs from PRNG and truly random generator. Those types of attacks may occur when an insufficient source of entropy is used for generating input data for PRNG such as hard drive latency (possibly observable – known input), network statistics (may be manipulated by an attacker – chosen-input/replayed-input). Examples of this type of attacks are conducted in practical part of this thesis, details explained in Section 3.4.

State Compromise Extension Attack – the assumption for this type of attack is that an attacker successfully compromised internal state S of an attacked generator. This might occur for example in a situation where an computer starts from some insecure state. The attack succeeds in case an attacker (with knowledge of S) can recover some information about the previous or the next output (either determine output values or distinguish it from truly random data).

Kesley et al. [12] also analyzed resistance of some real-world PRNGs concerning all mentioned types of attacks. The conclusion for X9.17 PRNG, shown in Figure 2.4, is following.

- Direct cryptanalysis seems to be equivalent to cryptanalysis of used block cipher (3-DES). However, they didn't prove it.

- There is a weakness concerning the replayed-input attack. By freezing the time (current time-stamp is the same for every run) it is possible to reduce difficulty to break PRNG from 2^{63} to 2^{32}.
- Compromising internal state, especially 3-DES encryption key, leads to the destruction of the security; the PRNG itself never recovers from this state. The only way how to recover is generation of whole new state including 3-DES encryption key.

Berry Schoenmakers and Andrey Sidorenko conducted practical cryptanalysis [15] against The Dual Elliptic Curve Pseudorandom Generator (DEC PRG) [16]. The algorithm is based on elliptic curves. The generator in each iteration outputs 240 least significant bits of x coordinate from point on elliptic curve. Let $\phi(r)$ denote the number of points on elliptic curve which have 240 least significant bits equal to r (size of r is 240 bits). On used elliptic curve there is close to 2^{256} points, this means the expected $\phi(r)$ for any r which is uniformly distributed is equal to $2^{256-240} = 2^{16}$.

However, Berry Schoenmakers and Andrey Sidorenko found out that for r generated by DEC PRG the value of $\phi(r)$ is slightly higher than 2^{16}. Using that knowledge it is possible to construct distinguisher which can observe this bias between output from DEC PRG and uniformly distributed data. The distinguisher works as follows: let r to be 240 bits from analyzed data and simple compute $\phi(r)$, then if value is higher than 2^{16} the output is considered an output from DEC PRG. The probability that output r from DEC PRG is distinguished by this distinguisher is approximately $p = 0.50078$. However, this probability can be improved by computing ϕ for k (k > 1) subsequent 240-bit blocks and take average of those values. The probability of success for attack with $k = 4000$ is $p = 0.548785$.

2.5 Human cryptanalysis

This section presents two cryptanalysis techniques - a linear and differential cryptanalysis. Both of them are powerful; however, they are not fully automated. The cooperation with cryptanalyst is necessary. The first was introduced by Matsui [17] in 1993 as a theoretical attack on Data Encryption Standard (DES). The other was proposed by Biham and Shamir [18] in 1991 with the same objective, to attack DES.

2.5.1 Linear cryptanalysis

It is one of the most famous known plaintext attacks against block ciphers; the assumption is that attacker knows some number of plaintext/ciphertext pairs, but he is not able to choose specific ones.

The attack is based on building a linear approximation of part of a block cipher (non-linear), which consists of bits of plaintext, ciphertext, and key (round subkey). The general form of such approximation is following:

$$\left(\bigoplus_{i \in \{1 \ldots b\}} P_i \right) \oplus \left(\bigoplus_{j \in \{1 \ldots b\}} C_j \right) = \left(\bigoplus_{k \in \{1 \ldots s\}} K_k \right) \tag{2.3}$$

where P_i, C_j and K_k denote i, j, k-th bit of plaintext, ciphertext and key respectively. $\oplus$ stands for Boolean `XOR` operator. Variables b and s are function-specific and represent block size and key size. [19]

Given such approximation and a perfect cipher, the probability that this approximation holds should be on average $p = \frac{1}{2}$. Probability p can be computed from the structure of the investigated function. For purposes of the attack we need to find an approximation which maximizes the bias $\epsilon = |p - \frac{1}{2}|$.

The linear approximation can be used to perform two types of attacks:

Obtaining one-bit information about the key. Given the approximation in the form of Equation (2.3), it is possible to obtain one-bit value of $X = \left(\bigoplus_{k \in \{1 \dots s\}} K_k\right)$ following Algorithm 1. [19]

Data: T denotes the number of results which equal to 0 when computing value of approximation for N plaintext/ciphertext pairs.

if $T > \frac{N}{2}$ **then**

$$X = \begin{cases} 0, & p > \frac{1}{2} \\ 1, & \text{otherwise} \end{cases}$$

else

$$X = \begin{cases} 1, & p > \frac{1}{2} \\ 0, & \text{otherwise} \end{cases}$$

end

Algorithm 1: Obtaining one bit information about key using the linear approximation.

Obtaining more bits of the key. For demonstrating this type of attack, we will follow the tutorial from [20] published by Howard M. Heys. He is attacking Substitution Permutation Network (SPN) defined within this paper. This function has four rounds; the size of subkey for each round and block is 16 bits. Each round consists of these operations: `mixing with subkey`, `substitution`, `permutation`. There are five subkeys because last round contains one more `mixing with subkey` at the end. Used notation is following.

- $K_{k,i}$ is an i-th bit of subkey for a k-th round. ($k \in \{1..5\}, i \in \{1..16\}$)
- $V_{j,i}$ is an i-th bit of output of a j-th round. ($j \in \{1..4\}, i \in \{1..16\}$)
- $U_{l,i}$ is an i-th bit of input to an i-th round. ($i \in \{1..4\}, i \in \{1..16\}$)

$$U_i = \begin{cases} P \bigoplus K_1, & i = 1 \\ V_{i-1} \bigoplus K_i, & \text{otherwise} \end{cases}$$

Heys is performing attack using 3-round approximation with probability either $p = \frac{15}{32}$ or $p = \frac{17}{32}$ based on specific subkeys. Key bits are omitted from the equation because the xor of all key bits results into fixed 0 or 1 bit which does not change resulting bias $\epsilon = \frac{1}{32}$. The approximation is following.

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 = 0 \tag{2.4}$$

Using this formula, it is possible to attack the last subkey K_5 by performing steps below.

1. Generate all possible options for subkey K_5 (2^{16}).

2. For each of N collected plaintext/ciphertext pair obtain bits of U_4 by reversing last round.
3. Apply the linear approximation (Equation (2.4)) to captured bits and plaintext.
4. For each subkey remember the number of pairs the approximation resulted with 0.
5. A subkey which ended up with a number which is most significantly different from $\frac{N}{2}$ is considered correct guess.

Using the linear cryptanalysis Matsui [21] was able to break DES in the full number of rounds using 2^{43} random plaintext/ciphertext pairs.

2.5.2 Differential cryptanalysis

It is chosen plaintext attack; an attacker owns some number of plaintext/ciphertext pairs for an fixed unknown key, and he can choose specific plaintexts.

The attack is based on dependence between differences of two plaintexts and corresponding outputs. Differences are computed using exclusive xor operation of whole block, for example difference between two plaintexts X_1 and X_2 is denoted by $\Delta X = X_1 \bigoplus X_2$. The difference between outputs Y_1 and Y_2 is expressed by $\Delta Y = Y_1 \bigoplus Y_2$. In ideal case the probability p, of getting difference ΔY for some ΔX, should be $\frac{1}{2^n}$ (differences should be equally distributed).

The idea behind the attack is looking for the pair $(\Delta X, \Delta Y)$ which maximizes the probability p of obtaining difference ΔY for plaintext ΔX. The pair $(\Delta X, \Delta Y)$ is called *differential*.

Heys in his tutorial [20] was simulating an attack using differential analysis with three round *differential* (ΔU_4 stands for a difference between inputs to fourth round). He found out that SPN for the plaintext difference $\Delta P = [0000\ 1011\ 0000\ 0000]$ results with $\Delta U_4 = [0000\ 0110\ 0000\ 0110]$ with probability $\frac{27}{1024}$. Given this approximation it is possible to obtain bits of K_5 following steps below.

1. Collect N plaintext pairs which satisfy chosen difference ΔP.
2. Generate all possible options for K_5 (2^{16}).
3. For each plaintext pair and each option of key do following steps.
   (a) Reversely compute bits of U_4 for both corresponding ciphertexts using guessed key.
   (b) For each key remember the number of times computed values give expected difference ΔU_4.
4. A key with the largest computed number is considered correct guess.

DES in full round has been broken with complexity less than 2^{55}. [18]

2.6 Distinguishers from truly random streams

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

Tests are based on testing a *null hypothesis*, which is mostly formulated as data being tested are random. Those tests are based on probability; this means that even truly random data may end up with rejected hypothesis. The output of each test is called *p-value* which can be described as a probability that a truly random generator produces data which are less random than the data which were tested. We interpret the tests with respect to a significance level which is commonly denoted as α. If the resulting *p-value* is less than the significance level we say we rejected the hypothesis on significance level α (data are probably not coming from a truly random generator). The common value of α is 0.01. [22]

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 equal to significance level α. When a tool does not reject data from a faulty generator, Type II error occurred. The probability of this error is denoted by β. However it is hard to compute the value of β because there are many possible types of non-randomness which might occur. [22]

The example of the common test often included in statistical test suites is a Monobit test, which is examining the uniformity of distribution of binary zeroes and ones bits within tested data. It is based on the fact, that there is a high probability that the amount of binary ones and zeroes is approximately the same assuming that each sequence occurs with same probability. A high-level view of this test is shown in Figure 2.6.

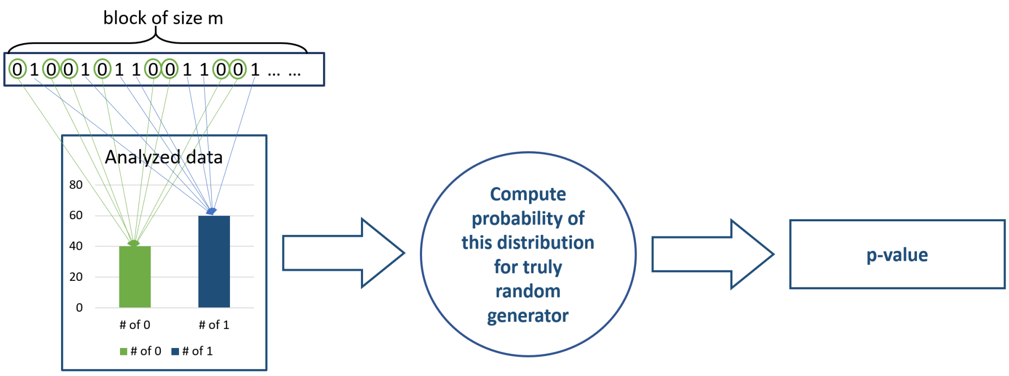

Figure 2.6: Monobit test from high level point of view.

Figure 2.7 shows a high-level view of the whole test suite, where all tests are triggered. After evaluation of all tests, it is necessary to interpret the entire run of the battery of tests and make a final decision, whether investigated data are considered genuinely random or not. It is likely that even data with perfect properties will fail some of the tests due to Type I error. Our interpretation was not based only on the number of failed tests, but also on extremeness of test failures. For example, if there was one failed test within the whole test suite with an extreme p-value (less than 10^{-7}), it is considered failure. However, if there were three failed tests with p-values close to α (not so extreme), it is still viewed as non-failure.

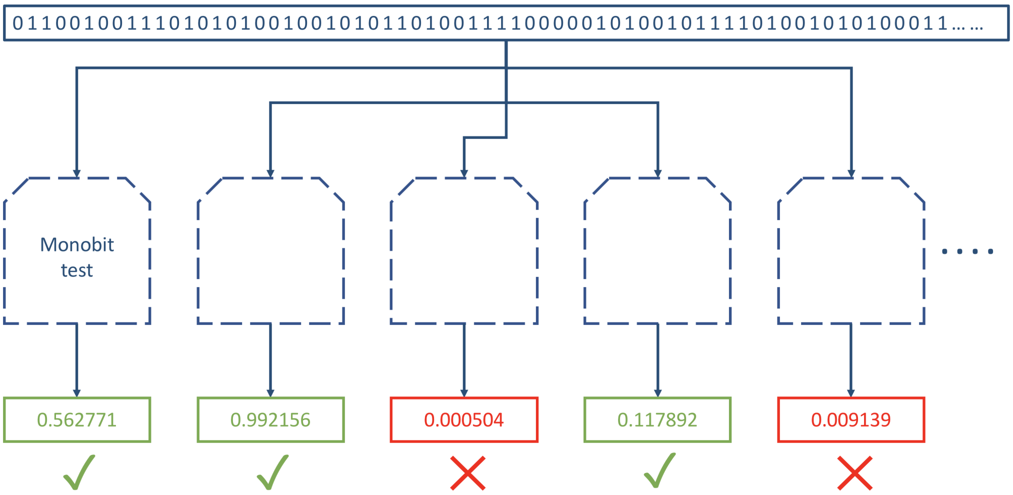

Figure 2.7: High level view of software tool run.

2.6.1 NIST STS

NIST STS [23] is the 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 [24] certification process. Even though the STS test suite is most commonly used, some of the other test suites generally have better results.

In this thesis we will not use original NIST STS implementation, instead of that we are using optimized version developed Marek Sýs and Zdeněk Říha which is approximately 50 times faster than reference implementation [25].

The tool contains 15 tests. However several of them have more variants. In total, 188 tests are executed within one run.

2.6.2 Dieharder

Dieharder [26] is an extension of Diehard test suite [27] developed by Robert G. Brown at Duke University. In the newest version, it contains together 31 tests. All tests from Diehard, three originates from NIST STS and the others are implemented by the author or come from different sources. However, not all tests will be used within this thesis. For choosing what tests to run we rely on project Randomness Testing Toolkit [28].

2.6.3 TestU01

Pierre L'Ecuyer and Richard Simard introduced TestU01 software tool. The aim of this tool is the provision of the general and extensive set of software tools for statistical testing of random number generators. It contains a significant number of tests, more than any other software tool we mentioned. Tests are organized into six batteries of tests. The battery of tests is a subset of tests, where each battery has a different purpose or time consumption. 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, it consumes 1414 times more time than *SmallCrush* and 5 times more time than *Crush* to test *Mersenne Twister* [6] PRNG on a computer with AMD Athlon running at 2.4GHz. For binary sequences there are batteries *Rabbit*, *Alphabit* and *BlockAlphabit*. [29]

Besides the batteries this software tool contains also some predefined pseudo-random number generators. Paper [29] contains also results of those generators with batteries *SmallCrush*, *Crush* and *BigCrush*.

2.6.4 BoolTest

BoolTest is simple but a strong testing tool developed by Marek Sýs et al. at the Centre for Research on Cryptography and Security, Masaryk University in Brno. It has a slightly different approach to randomness testing. The tool is based on a generalization of Monobit test. The main idea is 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. The process starts with a division of 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 success of distinguisher requires calculation of the value of the Boolean function for each block so that x_k is k-th bit of the block. Expected number of computations which results with binary one is statistically computed and compared with actual results using Z-score statistics [30]. This produces p-value.

Construction of distinguisher is based on the assumption that a combination of weaker and simpler Boolean functions may lead to stronger distinguisher. The process starts with brute-force evaluation of all possible Boolean functions of type $f(x_1, x_2, \cdots, x_m) = x_i$ for $\forall i \in \{1, 2, \cdots, m\}$ with possibility to choose more complex functions in this phase. However, it cannot be too complicated as it is necessary to brute-force all possible functions. The second phase captures some number of best distinguishers which are then combined with XOR operation. There are two ways how to optimize this approach: by pre-computation of values from the first phase, and by stopping evaluation at any time when reasonably strong distinguisher is found. If no strong distinguisher is found tested data are considered random. [31]

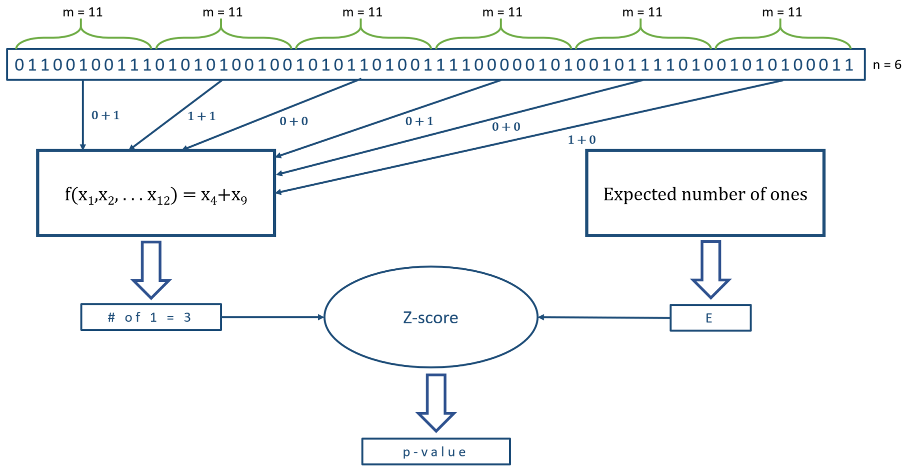

Figure 2.8: Example of evaluation with BoolTest for sequence of size 66 bits with block size 11 and number of blocks 6. Evaluated Boolean function (distinguisher) is $f(x_1, x_2, ...x_{11}) = x_4 + x_9$.

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 [32]. 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 [33] 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 prim-

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.

itives 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 [31]. 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 [34] and Matej Prišťák [35] 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 [36] regarding authenticated encryption systems from CAESAR competition [37]. Well known block ciphers like AES, DES, etc. were added by Karel Kubíček [38] and Tamás Rózsa [39] 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`.

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

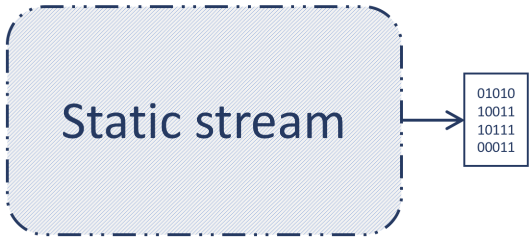

Figure 3.1: Example of one call of static stream.

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`.

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 initialization 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.

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

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 3.5: 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 [40] 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 [41].

```
{
  "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 the tool CryptoStreams.

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.

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.

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.

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 to block ciphers except for encrypt-decrypt test for round reduced versions. From 27 stream ciphers 15 are tested.

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 [42] 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.

	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 [42], word size is 32 bits.

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 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 [43]. 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 [23], Dieharder [26] and TestU01 [29]. 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 [29] 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 [5].

Linear congruential generator. Definition of this generator is shown in Equation (2.1). Chosen parameters are taken from [44] 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. It is based on following formula.

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

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.

Chosen parameters are following: $k = 2$, $a_1 = 2975962250$, $a_2 = 2909704450$ and $m = 9223372036854775783$ [45]. We are cutting upper binary zeroes similarly like in LCG.

Xorshift generator. The generator is based on *xor* and *shift* operations [46]. We have chosen version which is created by function `uxorshift_CreateXorshift13` and requires no additional parameters, more information in [5] on page 50.

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. The definition is shown in Equation (2.1). 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 [6]. However, it does not produce cryptographically secure random numbers [1].

Substract with carry [6]. This type of generator was introduced by George Marsaglia and Arif Zaman [47]. The definition is following:

$$x_n = (x_{n-S} - x_{n-R} - cy(n-1)) \mod M \tag{3.2}$$

Where

$$cy(n) = \begin{cases} 1, & \text{if } x_{n-S} - x_{n-R} - cy(n-1) < 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

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 $k = maxR, S$.

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`

3.5.2 Generators based on lightweight cryptographic primitives

The other part contains block ciphers taken over from project FELICS [48] 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 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 second test scenario is verifying the expected functionality of round-reduction by doing encryption followed by decryption (Encrypt-Decrypt test) for all rounds provided by tested function. This test is not passing for 6 functions out of 19 (failing marked with ✗ in last column in Table 3.3). The possible explanation of failure is missing round-reduction of key scheduling, which we intentionally do not perform. Despite the failure of the test we observed expected behavior concerning detectable bias by statistical testing – more rounds results in less bias.

Function	Round	Block size	Key size	Encrypt-Decrypt test
Chaskey [49]	16	16	16	✓
Fantomas [50]	12	16	16	✓
HIGHT [51]	32	8	16	✗
LBlock [52]	32	8	10	✗
LEA [53]	24	16	16	✗
LED [54]	48	8	10	✓
Piccolo [55]	25	8	10	✓
PRIDE [56]	20	8	16	✗
PRINCE [57]	12	8	16	✓
RC5-20 [58]	20	8	10	✓
RECTANGLE-K80 [59]	25	8	16	✗
RECTANGLE-K128 [59]	25	8	16	✗
RoadRunneR-K80 [60]	10	8	10	✓
RoadRunneR-K128 [60]	12	8	16	✓
Robin [50]	16	16	16	✓
RobinStar [50]	16	16	16	✓
SPARX-B64 [61]	8	8	16	✓
SPARX-B128 [61]	8	16	16	✓
TWINE [62]	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] Krister Sune Jakobsson. *Theory, methods and tools for statistical testing of pseudo and quantum random number generators*. 2014.

[2] Jan Krhovjak. “Cryptographic random and pseudorandom data generators”. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2009.

[3] Bruce Schneier et al. *Applied cryptography-protocols, algorithms, and source code in C*. 1996.

[4] Anne Canteaut. “Linear Feedback Shift Register”. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 355–358. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_235. URL: https://doi.org/10.1007/0-387-23483-7_235.

[5] 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.

[6] 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.

[7] Makoto Matsumoto and Yoshiharu Kurita. “Twisted GFSR generators”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 2.3 (1992), pp. 179–194.

[8] Christophe Petit et al. “A Block Cipher Based Pseudo Random Number Generator Secure Against Side-channel Key Recovery”. In: *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’08. Tokyo, Japan: ACM, 2008, pp. 56–65. ISBN: 978-1-59593-979-1. DOI: 10.1145/1368310.1368322. URL: http://doi.acm.org/10.1145/1368310.1368322.

[9] R. Impagliazzo, L. A. Levin, and M. Luby. “Pseudo-random Generation from One-way Functions”. In: *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*. STOC ’89. Seattle, Washington, USA: ACM, 1989, pp. 12–24. ISBN: 0-89791-307-8. DOI: 10.1145/73007.73009. URL: http://doi.acm.org/10.1145/73007.73009.

[10] ANSI X9. 17. *American National Standard for Financial Institution Key Management*. 1985.

[11] Sharon S. Keller. “NIST-Recommended Random Number Generator Based on”. In: *ANSI X9.31. Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms*. 2005.

[12] John Kelsey et al. “Cryptanalytic attacks on pseudorandom number generators”. In: *International Workshop on Fast Software Encryption*. Springer. 1998, pp. 168–188.

[13] C. E. Shannon. “Communication theory of secrecy systems”. In: *The Bell System Technical Journal* 28.4 (Oct. 1949), pp. 656–715. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[14] Morris J. Dworkin. *SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. Tech. rep. Gaithersburg, MD, United States, 2001.

[15] Berry Schoenmakers and Andrey Sidorenko. “Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator.” In: *IACR Cryptology ePrint Archive* 2006 (Jan. 2006), p. 190.

[16] Elaine B Barker and John Michael Kelsey. *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2007.

[17] Mitsuru Matsui. "Linear cryptanalysis method for DES cipher". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1993, pp. 386–397.

[18] Eli Biham and Adi Shamir. "Differential cryptanalysis of DES-like cryptosystems". In: *Journal of Cryptology* 4.1 (Jan. 1991), pp. 3–72. ISSN: 1432-1378. DOI: 10.1007/BF00630563. URL: https://doi.org/10.1007/BF00630563.

[19] Pascal Junod. *Linear cryptanalysis of DES*. Tech. rep. 2000.

[20] 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.

[21] Mitsuru Matsui. "The First Experimental Cryptanalysis of the Data Encryption Standard". In: *Advances in Cryptology — CRYPTO '94*. Ed. by Yvo G. Desmedt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 1–11. ISBN: 978-3-540-48658-9.

[22] 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.

[23] 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).

[24] 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).

[25] 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.

[26] 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).

[27] 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).

[28] 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.

[29] 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.

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

[31] 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.

[32] 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).

[33] 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).

[34] 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.

[35] 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.

[36] 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.

[37] 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).

[38] 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.

[39] 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.

[40] 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).

[41] 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).

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

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

[44] 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.

[45] 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`.

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

[47] 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`.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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`.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

A Glossary