Masarykova univerzita
Fakulta informatiky

Usage of evolvable circuit for statistical testing of randomness

Bachelor thesis

Martin Ukrop

Brno, spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by 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.

Martin Ukrop

Advisor: RNDr. Petr Švenda, Ph.D.

Acknowledgement

Thanks will be here.

Abstract

Abstract will be here.

Keywords

Keywords will be here.

Contents

1 Introduction

- problem description
- motivation -> EACirc
- summary of experiments to follow
- we = team of EACirc
- work and research done by me, if not stated otherwise (but consulted within the team!)
- licence of EACirc, licence of thesis text
- fithesis2 used

2 Statistical randomness testing

The goal of randomness testing is to determine, whether the data provided is *random*. The problem comes with the definition of randomness, since in truly random data, each fixed subsequence (e. g. sequence of a hundred zeroes) has the same probability of appearing. Thus, statistical metrics have been developed to asses the matter of randomness.

All the statistical randomness tests are based on mathematical properties that hold for *most* of the random sequences with a sufficient length. A simple example of such a property states that in each binary sequence the number of ones and zeroes should be approximately the same. It is crucial to be aware, that this will not hold for *all* sequences (see the example above), but the probability of randomly generating such a sequence sharply decreases with the increasing length.

Randomness testing based on statistical properties of data has both drawbacks and benefits, main of which are discussed below.

- **Speed**
  Once the tests are implemented, they do not require excessive amount of time to perform – the data is usually processed just once in a linear fashion.
- **Universality**
  Statistical tests can by applied to any binary data regardless of its origin – they perform equally well. This can be viewed both as an advantage and disadvantage, since tests cannot be effortlessly adapted to specific situations.
- **One-way design**
  The creation of new test must be preceded by the idea and analysis of some useful statistical property. This part may be very complicated and usually requires a team of skilled mathematicians.
- **Results interpretation**
  The ever-present ambiguity in statistical measurements sometimes makes the results interpretation a highly non-trivial task. It is crucial to understand what do the results indicate and what they do not. The above-mentioned finite sequence of binary zeroes fails most of the statistical randomness tests, but its generation is just as probable as any other fixed binary sequence of the same length. Put in another words, even the true random generator must produce non-random looking sequences once in a while.

In practise, statistical randomness testing is being widely used in fields where the quality of random data is crucial, such as cryptography. To ease the assessment process, several statistical randomness testing suites have been developed, some of which are discussed below.

2.1 Statistical Test Suite by NIST

Perhaps the most widely used battery of statistical tests is the Statistical Testing Suite by National Institute of Standards and Technology (STS NIST). The primary motivations for developing this test suite was the need of standardised tests for detecting non-randomness in binary (pseudo)random sequences utilized in cryptographic applications. As well as designing the tests, NIST provides their reference implementation and guidance in their use and application. [?]

The battery consists of 15 different tests, some of which can be run with several parameters. For detailed description of the tests, see the original documentation [?]. The implementation provided by NIST supports variable input data length and arbitrary number of independent data streams. The testing results provide the combined p-value of all data streams and the number of passed runs for each test according to the set significance level. Detailed setting used for the purposes of this thesis can be found in section 4.4.

2.2 Diehard battery of tests

The second (unofficial) standard of statistical randomness testing is the Diehard Battery of Tests of Randomness, developed by George Marsaglia over several years at Florida State University. [?] Although now becoming slightly outdated, they were one of the first and most-well known in the pioneering years of statistical testing of randomness. For long, the Diehard Battery of Tests was considered a golden standard along with STS NIST.

The battery consist of 12 different tests. The original implementation, documentation and test descriptions are still available, but since the code has not been revised from its creation in 1995, we chose not to use Marsaglia's original implementation.

2.3 Dieharder: A Random Number Test Suite

Dieharder, as its predecessors, aims to ease the testing of (pseudo)random generators and data for a variety purposes in research and cryptography. Developed by Robert G. Brown at the Duke University, it is designed to be as extensible as possible, allowing easy implementation of new tests and generators for testing. Most of the tests used allow for modifying the default parameters, enabling advanced users to fine-tune the testing process. According to its creators, it is intended to be the "Swiss army knife of random number test suite", or if you prefer, "the last suite you'll ever ware" for testing random numbers. [?]

After designing the testing framework, the development team gradually reimplemented and improved the original tests from the Diehard Battery of Tests of Randomness (see section 2.2), the tests from STS NIST (see section 2.1) and began to prepare and implement their own new tests. The suite now contains 31 different tests from various sources. Tests can be run selectively. The testing results provide the combined p-value for each test and a verdict of PASSED, WEAK or FAILED according to the set significance levels. Detailed settings used for the purposes of this thesis can be found in section 4.4.

2.4 Drawbacks of human-designed statistical tests

Although convenient in some ways, statistical randomness testing based on human-designed tests has several important drawbacks. As mentioned above, the test creation must be preceded by an idea of mathematical property and its thorough analysis, which can be extremely time- and people-consuming. Further on, the tests are limited to one particular property and adapting them to specific situation requires beginning the process of test creation all over again.

Both of the above-mentioned problems would be resolved if tests of comparable quality could be generated automatically, without the help of human specialists. Such concept and its comparison with human-generated tests is presented in the following chapters.

3 Evolution-based randomness testing

In this chapter we try to describe a method of automatically generating statistical randomness tests. Compared to the standard (manual) way of their creation, our approach would have a couple of advantages:

- no prior knowledge of statistical properties of random data is needed;
- test creation does not require excessive human analytical labour;
- tests adapted for specific situations can be easily developed;
- atypical and/or yet unknown data properties may be used.

The main idea is to use supervised learning techniques based on evolutionary algorithms to design and further optimize a successful *distinguisher* – the test determining whether its input comes from a truly random source or not. The distinguisher will be represented as a hardware-like circuit consisting of a number of interconnected simple functions. The evolution will use the principles of genetic programming.

3.1 Basic principles of genetic programming

Genetic programming is a biologically inspired supervised learning technique. It tries to converge to optimal solution by making subtle changes to previous partial solutions, assessing their impact and propagating the perspective changes until reaching the desired success rate. The existence of partial problem solutions is therefore essential. The main flow of evolution implemented by genetic programming is as follows:

1. Firstly, a random set of partial solutions is generated. The solutions may be highly unsuccessful, but some will nonetheless be better than others. This set of solutions is called a *population*.

2. Secondly, the success of all individual solutions from the population is evaluated. The assessment is done using a so called *fitness function*. The quality of this function is crucial to the whole algorithm, as it distinguishes the better and more successful partial solutions from the worse ones.

3. A new population of solutions is created by making a *sexual crossover* of the best solutions from the previous generation. Informally put, solutions are subject to the survival of the fittest.

4. A small random change may be applied to some individuals in the new population. This *mutation* prevents the population from getting stuck in the local optimum and increases the chances of reaching a global optimum.

5. Steps 2-4 are iterated over and over, until the desired success rate of the population is achieved or the required number of generations have evolved.

The principles of evolutionary algorithms induce a couple of design limitations and disadvantages. The most important ones include:

- Only problems with a sufficient space of partial solutions are applicable, since the individuals must be assessed to determine the fittest.
- A small change in the solution should induce only a small change in the individual's fitness. If the changes were too rapid, the evolution wouldn't be able to stabilize on the better and more successful solutions.
- The evolution phase can be computationally very expensive, since making only small changes to the individuals may require higher number of generations evolved.
- It may be quite difficult to fine-tune the parameters (such as population size, mutation and crossover probabilities) to achieve the best results.

To counterweight the drawbacks, it must be noted, that evolutionary algorithms allow us to create solution not just for particular instance of the problem, but to the whole set of similar problems – we may be trying to evolve a universal solver, rather than for the solution itself. This improves the computation complexity, because after an expensive learning phase, the evolved solver may be used repeatedly on multiple instances of the problem. However, the evolution of the general solver can be trickier than it seems, since over-learning (i. e. finding the solution just to the particular instance of the problem) has to be avoided.

3.2 Using software-emulated circuits

Our goal is to create a simple circuit performing the desired task – distinguishing the random and non-random data streams. Thus, let's consider solutions in the form of of a hardware-like circuits with gates (*function nodes*) and a set of wires (*node connectors*). Each node is responsible for computation of a simple function on its inputs (e. g. binary AND operation). Circuit nodes are positioned into layers, where outputs from one layer are connected to inputs of the next. Input of the whole circuit is used as an input for the first layer and output of the last layer is considered the output of the entire circuit. Connectors may only link adjacent layers, but may cross each other (contrary to real single-layer hardware circuits). An example of such hardware-like circuit can be seen in **??**.

In the current solution design, we consider only simple nodes operating on bytes. The supported functions are:

- common bit-manipulating functions (OR, AND, XOR, NOR, NAND, ROTL, ROTR, BITSELECTOR),
- simple arithmetical functions (SUM, SUBS, ADD, MULT, DIV),
- identity function (NOP) and
- function reading specific input byte (READX).

Although it would be sufficient to restrict ourselves to a smaller set of functions (e. g. NAND only), we chose to support a wider variety of functions as an human understandability trade-off. More complex and sophisticated functions enable us to limit the circuit to significantly smaller number of layers and nodes, while retaining a comparable expressive power.

To some extent, the structure of a software circuit resembles artificial neural networks (deep belief belief neural networks in particular). Notable differences are in the learning mechanism and circuit dimensions (neural networks usually use very small number of layers). The function of individual nodes is different as well, since all nodes node in artificial neural networks perform the same function.

3.3 EACirc: framework for automatic problem solving

Combining the principles of genetic programming and software circuits, we developed EACirc, the framework for automatic problem solving. The initial version of EACirc was created by Petr Švenda at the Laboratory of Security and Applied Cryptography, Masaryk University [?] and was loosely based on SensorSim, locally developed application for simulation of sensor network [?]. This initial version provided the main shared functionality: evolutionary capabilities, software circuit emulation and basic fitness evaluation. Later on, the application was improved by Matej Prišťák and Ondrej Dubovec (as their master and bachelor theses, respectively).

Afterwards, the object model of the entire project was redesigned and a handful of new features was added by myself. Most of the code that was taken over was revised and refactored as necessary to ease the understanding of its function and to standardise naming and programming principles used throughout the project. Currently, the framework consist of the following main parts:

- **Evolutionary core**
  The core evolutionary features are provided by GAlib, a C++ Library of Genetic Algorithm Components developed at MIT [?]. The library, when parametrized by function callbacks (e. g. function for mutation, sexual crossover, fitness function, . . .), handles the main evolutionary actions.
- **Circuit emulator**
  The emulator simulates the behaviour of the circuit loaded from numerical representation. It plays a crucial role in fitness assessment of the population.
- **Project modules**
  These modules are responsible for generating the data used in circuit fitness assessment. Each module (*project*) corresponds to one experiment (e. g. eStream candidate ciphers testing, SHA-3 candidate functions testing, . . .). The module's main responsibility is to prepare the required number of problem–solution pairs in the form of circuit input stream (problem) and optimal circuit output (solution). These pair are called a *set of test vectors*.
- **Evaluator modules**
  Evaluator is a function responsible for yielding a numerical value of fitness, when provided with the pairs of actual and expected circuit outputs. There are multiple approaches to evaluators – the equality of expected and actual output can be based on Hamming weight, numerical value, . . .
- **Random generators**
  Since evolutionary algorithms are highly randomized, a source of randomness is

needed. To ensure the computation determinism (all experiments need to be exactly reproducible), a hierarchy of random generators was developed. To satisfy the varying needs, several generator types are implemented: true quantum random generator (based on pre-generated data), configurable biased generator and low-entropy MD5-based generator.

- **Self-tests**
  For the ease of development, EACirc provides a handful of self-tests. Running these tests ensures the consistency of seeding and data manipulation. Tests are implemented using CATCH, a C++ Automated Test Cases in Headers [?].
- **XML manipulating library**
  Most of the files produced and processed by the framework are XML-structured files. All these files are handles via TinyXML, a simple, small, minimal, C++ XML parser library [?].
- **Static checker**
  Although the static checker shares some code with the main framework, it is built as an independent application. It is designed to verify obtained results (evolved circuits) by circumventing both the genetic manipulations and circuit emulator.
- **Miscellaneous utilities**
  EACirc framework comes with an assortment of scripts, used mainly for downloading, checking and processing the results.

3.4 Current capabilities of EACirc

EACirc has a variety of other functions improving the core features of evolutionary algorithms and software circuit emulation. This section provides a short and by no means exhaustive list of them.

- **Bit-reproducibility**
  Bit-reproducibility is essential for the most research projects, since it enables replication and verification of the results. EACirc uses genetic programming, which is fundamentally a randomized algorithm. Therefore, a hierarchy of random generators with strictly defined scope of usage and seeding process was developed. This allowed us to replicate an experiment by just providing the same input files and a fixed central seed.
- **Computation recommencing**
  After reaching bit-to-bit determinism, we implemented the ability to recommence older computations. To allow for this, EACirc was made capable of saving and loading its entire internal state to a set of XML-structured files. This feature is especially useful for computation-expensive experiments – when the machine is rebooted, we can continue from last saved state instead of starting all over again.
- **Multi-format output**
  For easy reusing and analysis, the evolved circuits are output in 4 different formats:
  - binary output (useful for reloading the circuits into EACirc),
  - graph DOT output (serves as a visual aid to human analyst),
  - simple text output (application-independent export format) and

  - program output (in the form of a stand-alone C program used for static analysis).

  The DOT graph format can be easily displayed using the Graphviz library [?] and thus facilitates manual analysis done by humans after the computation.
  This functionality was implemented as early as the first version of EACirc.
- **Static checker for circuits in C**
  Static checker is used to verify the success of evolved circuits exported as C programs. The verification uses pre-generated test vectors and circumvents most parts of the EACirc framework, mainly the evolution and software circuit emulation. The independence of this process is of utmost importance, since it provides supporting evidence for the achieved results.
- **Modular object model**
  When redesigning the object model, the principle of modules was utilized, thus enabling integration of multiple projects and evaluators according to actual needs. This greatly improved framework's flexibility and extensibility. Currently, the following three projects (experiments) are implemented:
  - Project for distinguishing between the output of eStream candidate ciphers and random stream of data was taken from the work of Matej Prišťák. It was slightly revised to operate within the new object model and allow more detailed configuration.
  - Project for distinguishing between the output of SHA-3 candidate functions and random stream of data was inspired by the work of Ondrej Dubovec. Hash functions implementations were taken over, but the test vector generation process was reimplemented from scratch.
  - A small project for distinguishing among external binary files.

Note, that EACirc is a project beyond the scope of this thesis. Some parts were added and/or redesigned in the process, so different experiments may have incompatible configuration files and may have produced incomparable results. For further details, user and development documentation, see EACirc wiki at GitHub [?].

4 Experiment settings and output data

This chapter summarizes the configuration of EACirc used in the experiments presented in later chapters. The accounts of random data used are given and EACirc outputs are described. In most experiments, our performance is compared to traditional batteries of statistical tests (STS NIST, Dieharder) therefore settings and output description of these batteries is provided as well.

4.1 EACirc settings

Most of the general settings (evolution and circuit parameters) were taken from Matej Prišťák's thesis. The experiments supporting these parameters values were not reproduced, except for a few – for details, see chapter 5.

The evolution works with a population of 20 individuals, with a sexual crossover probability of 20 % and a mutation probability set to 5 %. In each case (if not stated otherwise), we evolve 30 000 generations with the test vector set (learning data) changing every 100^{th} generation. Thus, a total of 300 unique test vector sets is used in each run.

The circuit dimensions are limited to 5 layers with a maximum of 8 function nodes per layer. It processes up to 16 input bytes and produces 2 output bytes. Because of bad experience in previous work, using the READX function is forbidden. All other implemented functions are allowed.

Each testing set consists of 1 000 independent vectors, exactly half of which is truly random. According to research done by Matej Prišťák, the imbalance in test vectors would make the circuit learn what type is more frequent in the particular set instead of developing a deterministic distinguisher. The order of random and non-random vectors in the set is not fixed. Hence (Equation 4.1), all the results output by EACirc are based on a sample of about 2.5 MB of assessed data.

$$\Sigma = \frac{30000 \text{ generations}}{100 \, \frac{\text{generations}}{\text{test set}}} \cdot \frac{1}{2} \cdot 1000 \, \frac{\text{vector}}{\text{test set}} \cdot 16 \, \frac{\text{bytes}}{\text{vector}} \approx 2.29\,\text{MB} \tag{4.1}$$

The expected circuit output is always `0x00` (zero byte) for a non-random vector and `0xff` (full byte) for a random one. The used evaluator considers each of the output bytes separately, taking bytes with numerical interpretation lower than 128 as indicating a non-random stream and bytes higher than 127 as indicating a random stream. Hence, the decision is based only on the first bit of each output byte. Using the output of the evaluator, the fitness of the circuit is quantified as a quotient of a number of correctly predicted vectors and a total number of vectors in a set.

Experiment-specific settings (e. g. ways of generating non-random stream) are described in the appropriate chapters along with the results and their interpretation.

4.2 Random data sources

Eacirc requires a good source of randomness, since the the distinguishing process is based on comparing the assessed data with a stream of data we declare to be random. All the achieved results therefore rise and fall on the quality of this referential stream.

Fortunately, quantum physics provides randomness with inherent unpredictability based on measuring quantum effects of photons. We acquired several hundred megabytes of quantum random data from the following on-line services:

- *Quantum Random Bit Generation Service*
  provided by Ruđer Bošković Institute in Zagreb, Croatia [?] and
- *High Bit Rate Quantum Random Number Generator Service*
  provided by Humboldt University of Berlin, Germany [?].

The data from both sources have been thoroughly tested and compared, for details and results see section 5.2.

4.3 EACirc output data

The randomized nature of evolutionary algorithms calls for multiple executions of each experiment due to variation in results. For the most of the following experiments, we performed 30 independent runs. The final result presented is the average of these 30 executions.

In each run, the maximum population success rate in the generations just after the change of test vectors are examined. In our setting, this concerns the 1^{st}, 101^{st}, 201^{st}, ... and 29901^{st} generation. The presented results are of 2 types, depending on how good the found distinguishers are.

- If *strong distinguishers* were found, we show the average number of generations needed to reach them. For our purposes, a population of strong distinguishers has a maximum success rate in generations just after the change of test vectors over 99 % during at least 50 consecutive test vector sets (5 000 generations). We call these distinguishers strong, because of their anticipated high success rate on streams they have not been learning on so far.
- If a population of strong distinguishers was not reached during the evolution, we present the average value of maximal success rates in generations just after the change of test vectors, further averaged across all 30 runs. This average average maximum (AAM) is presented in parentheses.

4.4 Settings and output data for statistical test batteries

To compare our results with existing statistical tests, all experiments were replicated using standard batteries of statistical randomness tests (STS NIST and Dieharder). For each setting in EACirc, an external file with 250 MB of the assessed stream was created. The same stream was used for both STS NIST and Dieharder tests. For further information on STS NIST and Dieharder, see chapter 2.

STS NIST was run on 100 sub-streams, each consisting of 1 000 000 bits. This amounts to about 11.92 MB of assessed data. All 15 available test were run in all supported configurations. Some runs had problems with tests *Random Excursions* and *Random Excursions Variant* (they considered no or less than 100 sub-streams during these test), so to ensure statistical accuracy of results, these test are omitted from the results. For each test, the following results are output:

- the number of passed runs (a run is declared failed, if its p-value lies out of the interval determined by the significance level of $\alpha = 0.01$) and
- the combined p-value of all 100 runs of the test.

The result of all tests with all supported variants (162 tests in total, 2 tests excluded as mentioned above) is summarized in a cumulative score. The score assigns 1 to a test with both number of passed tests and the combined p-value within the significance interval and assigning 0 otherwise. In summary, a fraction of 162/162 denotes a random stream (all tests passed) while a value of 0/162 denotes a highly non-random stream (no test passed).

From the Dieharder suite, only the test corresponding to the original Diehard collection were used. The only exception is the *Diehard Sums Test* which was omitted, since the Dieharder community claims it has a couple of implementation bugs and thus should not be used at all. Each of the chosen tests was rune just once, but was let to process as much data as it required. Running the whole set processed about 582 MB altogether with the smallest test consuming about 3 MB and the largest one about 127 MB. Each test was labelled as PASSED, WEAK or FAILED according to the threshold interval it falls within. The value of $\tau_{weak} = 0.005$ and $\tau_{fail} = 0.000001$ were used. The result of the whole suite (20 tests in total) is again summarized in a cumulative score assigning 1 to a PASSED test, 0.5 to a WEAK test and 0 to a FAILED test.

5 Control distinguishers

- introduction – the need of reference numbers before analysis
- we need to define what does it mean "indistinguishable" in our setting
- we use quantum random data from Humboldt Universitat and Quantum random bit generator service as a standard for randomness

5.1 Looking for non-randomness in quantum random data

- trying to distinguish quantum random data from quantum random data
  => we presume to fail
- using random data from Quantum random bit generator service
- statistical batteries: data are random (Dieharder: 20/20, STS NIST: 188/188)
- evolution: no stable distinguisher found, AAM of 0.52 (differences in various runs in 3rd or 4th decimal place)
- presumption: dependence on test set size and population size
- presumption confirmed (Table 5.1), AAM decreases with smaller population and bigger test set size

		number of test vector in a set					
		200	500	1000	2000	5000	10 000
individuals in population	5	–	–	(0.509)	-	-	-
	10	–	–	(0.514)	-	-	-
	20	(0.544)	(0.527)	(0.520)	(0.514)	(0.509)	(0.506)
	50	-	-	(0.526)	-	-	-
	100	-	-	(0.530)	-	-	-

Table 5.1: Dependence of AAM on population size and test vector set size.

5.2 Distinguishing quantum random data from different sources

- distinguishing streams of quantum random data from Humboldt University and streams of quantum random data from Ruđer Bošković Institute
  – Quantum Random Bit Generator Service, Centre for Informatics and Computing, Ruđer Bošković Institute, Zagreb, Croatia
  – Quantum Random Number Generator Service, Department of Physics, Humboldt University, Berlin, Germany
- 6 files of 5 MB from each source
- fixed initial reading offsets as (0,0)
  => same data from given file in each run

- looking for distinguisher for each pair
- interpretation of results (Table 5.2):
  - data from both sources are equally random for our purposes
  - there is no single statistically different stream in these =>they can be used interchangeably

		QRBG service (Ruđer Bošković Institute, Croatia)					
		stream 1	stream 2	stream 3	stream 4	stream 5	stream 6
QRNG service (HU, Germany)	stream 1	(0.521)	(0.520)	(0.520)	(0.519)	(0.519)	(0.519)
	stream 2	(0.518)	(0.519)	(0.520)	(0.520)	(0.520)	(0.519)
	stream 3	(0.519)	(0.522)	(0.519)	(0.520)	(0.519)	(0.519)
	stream 4	(0.520)	(0.520)	(0.519)	(0.518)	(0.519)	(0.519)
	stream 5	(0.519)	(0.520)	(0.519)	(0.518)	(0.520)	(0.520)
	stream 6	(0.520)	(0.519)	(0.520)	(0.520)	(0.519)	(0.519)

Table 5.2: Distinguishing binary quantum random streams from independent sources.

5.3 Uncompressed audio streams

- 12 files
  - 3 quantum random files
  - 3 noise files (white, pink, Brown), generated by SoX
  - 3 noise files via intermediate mp3 compression (2 channels, 16bits, 44.1 kHz => bitrate of 128kbps)
  - noise after 128kbs mp3 compression had 480 kB
  - 3 samples of transcendental khaoblack metal music
- each file is 30sec of uncompressed WAV audio (5.3MB) (including quantum random files, wav header generated by SoX)
- evolving distinguisher for each pair
- interpretation of results (Table 5.3):
  - quantum random stream are undistinguishable (we already know)
  - generated white noise is true (undistinguishable from random)
  - pink and Brown noise are different, distinguishable (good result, since pink and Brown noise are generated by filtering white noise and are biased towards lower frequencies)
  - different types of noise can be quite successfully distinguished from one another (generally over 75%)
  - mp3 compression has small, but detectable effect on the sound (nearly undetectable by unskilled human ear, but successfully shifts the distinguisher success to cca 0.58)
  - comparing noise and mp3 compressed and decompressed noise of the same kind is difficult

  - used metal samples are nearly indistinguishable from each other on the binary level (although the differences are easily detectable by human ear)
  - used metal samples can be reliably distinguished from white noise (which is, in fact only a stream of random data) - general success over 80%. Less so from pink and Brown noise - general success around 65%.
- most of the runs have slow rising tendency in fitness
  => if more generations, the average maximum value might be slightly higher

		random streams			noise (true)			noise (via mp3)			metal music		
		random stream 1	random stream 2	random stream 3	white noise	pink noise	Brown noise	white noise (via mp3)	pink noise (via mp3)	brown noise (via mp3)	metal music (sample 1)	metal music (sample 2)	metal music (sample 3)
random	random stream 1	n/a	(0.52)	(0.52)	(0.52)	(0.80)	(0.84)	(0.59)	(0.93)	(0.89)	(0.84)	(0.87)	(0.83)
	random stream 2	(0.52)	n/a	(0.52)	(0.52)	(0.83)	(0.83)	(0.57)	(0.82)	(0.84)	(0.90)	(0.85)	(0.82)
	random stream 3	(0.52)	(0.52)	n/a	(0.52)	(0.94)	(0.91)	(0.58)	(0.83)	(0.83)	(0.89)	(0.83)	(0.85)
noise (true)	white noise (true)	(0.52)	(0.52)	(0.52)	n/a	(0.83)	(0.81)	(0.59)	(0.87)	(0.89)	(0.86)	(0.93)	(0.81)
	pink noise (true)	(0.80)	(0.83)	(0.94)	(0.83)	n/a	(0.76)	(0.86)	(0.52)	(0.76)	(0.65)	(0.65)	(0.66)
	Brown noise (true)	(0.84)	(0.83)	(0.91)	(0.81)	(0.76)	n/a	(0.86)	(0.76)	(0.56)	(0.71)	(0.69)	(0.68)
noise (mp3)	white noise (via mp3)	(0.59)	(0.57)	(0.58)	(0.59)	(0.86)	(0.86)	n/a	(0.91)	(0.83)	(0.84)	(0.80)	(0.78)
	pink noise (via mp3)	(0.93)	(0.82)	(0.83)	(0.87)	(0.52)	(0.76)	(0.91)	n/a	(0.78)	(0.63)	(0.68)	(0.70)
	Brown noise (via mp3)	(0.89)	(0.84)	(0.83)	(0.89)	(0.76)	(0.56)	(0.83)	(0.78)	n/a	(0.71)	(0.69)	(0.67)
metal music	metal music (sample 1)	(0.84)	(0.90)	(0.89)	(0.86)	(0.65)	(0.71)	(0.84)	(0.63)	(0.71)	n/a	(0.54)	(0.56)
	metal music (sample 2)	(0.87)	(0.85)	(0.83)	(0.93)	(0.65)	(0.69)	(0.80)	(0.68)	(0.69)	(0.54)	n/a	(0.53)
	metal music (sample 3)	(0.83)	(0.82)	(0.85)	(0.81)	(0.66)	(0.68)	(0.78)	(0.70)	(0.67)	(0.56)	(0.53)	n/a

Table 5.3: Distinguishing random streams and uncompressed audio (noise, compressed noise, metal music).

6 Distinguishing cipher outputs from random stream

- introduction, idea, running EACirc along with statistical batteries
- stream ciphers from eStream competition

6.1 Stream ciphers used

- ciphers except for ?? (why??)
- from last phase
- those that could be limited in rounds are tested in weaker variant as well
- differences from Metej Pristak thesis

6.2 Generating binary stream from stream ciphers

- cipher modes (iv+key initialization frequency)
- case of LEX (not weakening the cipher, only making shorter output)
- case of TSC (producing binary stream of 0 for 1-8 rounds) => problems in 3 Dieharder tests

6.3 Results interpretation

- ???
- more or less as statistical batteries
- dieharder better in some case than STS-NIST (is newer and some tests are redesigned)
- statistical tests has much more input data compared to EACirc
- using evolved distinguisher is quick

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	0.0	0	$n = 2681$	0.0	0	(0.85)	0.0	5	$n = 1431$
2	0.5	0	(0.54)	1.0	0	(0.54)	15.5	146	(0.52)
3	1.0	0	(0.53)	1.0	0	(0.53)	15.0	160	(0.52)
4	3.5	79	(0.52)	3.0	78	(0.52)	20.0	160	(0.52)
5	4.5	79	(0.52)	3.5	91	(0.52)	17.5	161	(0.52)
6	19.0	158	(0.52)	19.0	159	(0.52)	18.0	162	(0.52)
7	18.5	162	(0.52)	19.0	161	(0.52)	20.0	161	(0.52)
8	20.0	162	(0.52)	20.0	159	(0.52)	19.0	161	(0.52)

Table 6.1: Random distinguishers for Decim ciphertext.

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	20.0	162	(0.52)	20.0	161	(0.52)	18.0	162	(0.52)
4	20.0	162	(0.52)	20.0	162	(0.52)	20.0	162	(0.52)

Table 6.2: Random distinguishers for FUBUKI ciphertext.

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	0.0	0	$n = 221$	0.0	0	(0.67)	18.5	162	(0.52)
2	0.0	0	$n = 471$	0.5	0	(0.66)	20.0	162	(0.52)
3	19.5	160	(0.52)	20.0	162	(0.52)	20.0	162	(0.52)
13	20.0	162	(0.52)	20.0	161	(0.52)	19.5	162	(0.52)

Table 6.3: Random distinguishers for Grain ciphertext.

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	20.0	162	(0.52)	20.0	162	(0.52)	20.0	162	(0.52)
10	20.0	160	(0.52)	20.0	162	(0.52)	20.0	162	(0.52)

Table 6.4: Random distinguishers for Hermes ciphertext.

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	0.0	0	$n = 148$	0.0	0	$n = 7274$	3.0	1	$n = 154$
2	4.0	1	$n = 221$	4.0	1	$n = 304$	3.5	1	$n = 254$
3	0.5	1	$n = 378$	3.5	1	$n = 491$	4.0	1	$n = 361$
4	20.0	162	(0.52)	19.5	162	(0.52)	20.0	161	(0.52)
10	19.5	162	(0.52)	19.5	160	(0.52)	20.0	160	(0.52)

Table 6.5: Random distinguishers for LEX ciphertext.

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	5.5	1	(0.87)	8.5	1	(0.67)	17.5	161	(0.52)
2	5.5	1	(0.87)	7.0	1	(0.67)	19.5	162	(0.52)
3	20.0	162	(0.52)	20.0	162	(0.52)	19.5	161	(0.52)
12	20.0	162	(0.52)	19.5	161	(0.52)	19.0	161	(0.52)

Table 6.6: Random distinguishers for Salsa20 ciphertext.

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1–8	0.0*	0	$n = 104$	0.0*	0	$n = 101$	0.0*	0	$n = 104$
9	1.0	1	$n = 234$	1.5	1	$n = 491$	2.0	1	$n = 121$
10	2.0	13	$n = 188$	3.0	13	$n = 218$	3.0	12	$n = 158$
11	10.0	157	(0.52)	11.5	157	(0.52)	14.0	159	(0.52)
12	16.0	162	(0.52)	17.0	161	(0.52)	17.5	162	(0.52)
13	20.0	162	(0.52)	20.0	162	(0.52)	19.0	162	(0.52)
32	20.0	161	(0.52)	20.0	162	(0.52)	20.0	161	(0.52)

Table 6.7: Random distinguishers for TSC-4 ciphertext.

7 Analysis of Salsa20 output stream

- learns current vectors quicker than other ciphers
- the case of six

8 Distinguishing hash outputs from random stream

- introduction, idea
- hash function candidates from SHA-3

8.1 Hash functions used

- except for 2 (?? source code size, compilation)
- from last phase
- those that could be limited in rounds are tested in weaker variant as well
- differences from Ondrej Dubovec Bc thesis

8.2 Generating binary stream from hash functions

- length set to 256b
- hashing 4 byte counters starting from random value (in fact, cutting each hash in half)

8.3 Determining optimal set change frequency

- previously,we used change every 100 generations
- 100 was taken from Matej Pristak's thesis
- Ondrej proposes 10 as best, however, data is not provided
- interpretation of results (Table 8.1):
  - ???

	change frequency for test vector set							
	5	10	20	50	100	200	500	1000
30 000 g.	(0.614)	(0.614)	(0.607)	(0.602)	(0.599)	(0.598)	(0.591)	(0.582)
run-time	70 m.	52 m.	42 m.	37 m.	32 m.	28 m.	23 m.	20 m.
300 sets	(0.567)	(0.583)	(0.585)	(0.589)	(0.599)	(0.608)	(0.617)	(0.618)
run-time	4 m.	6 m.	9 m.	19 m.	32 m.	57 m.	115 m.	220 m.

Table 8.1: Determining optimal change frequency for test vector set.

8.4 Results interpretation

- ???

	number of rounds						
	0	1	2	3	4	5	full
ARIRANG	$n = 694$	$n = 707$	$n = 467$	$n =$ 1071	(full)	–	(0.52)
Aurora	$n =$ 5614	(0.75)	(0.78) !!!	(0.52)	–	–	(0.52)
Blake	$n = 474$	(0.52)	–	–	–	–	(0.52)
Blue Midnight Wish	(0.52)	–	–	–	–	–	(0.52)
Cheetah	$n = 181$	$n = 574$	$n = 708$	(0.90) !!!	(0.86)!!!	(0.52)	(0.52)
CHI	(0.52)	–	–	–	–	–	(0.52)
CRUNCH	$n = 104$	$n = 534$	$n = 954$	10-$n =$ 1327	17- $n = 774$	34- (0.52)	(0.52)
CubeHash	$n = 104$	(0.52)	–	–	–	–	(0.52)
DCH	$n = 104$	(0.73) !!!	(0.52)	–	–	–	(0.52)
Dynamic SHA	$n = 484$	$n =$ 2337	$n =$ 1773 !!!	(0.95) !!!	(0.74)	(0.61)	(0.52)
Dynamic SHA	from 6 ->	(0.59)					
Dynamic SHA2	–	(0.94) !!!	(0.74)	(0.75)	(0.57)	(0.60)	(0.52)
Dynamic SHA2	from 6 ->						

Table 8.2: Random distinguishers for SHA-3 candidate functions.

	number of rounds					
	0	1	2	3	4	5	full
ECHO	–	(0.73) !!!					(0.52)
ESSENCE	8-(0.52)						(0.52)
Fugue	(0.52)						(0.52)
Grøstl	$n =$ 8651 !!!						(0.52)
Hamsi	$n =$ 12408 !!!						(0.52)
JH	$n = 581$						(0.52)
Lesamnta	$n = 791$						(0.52)
Luffa	$n = 604$						(0.52)
MD6	$n = 101$						(0.52)
Sarmal	(0.52)						(0.52)
SHAvite-3	(0.52)						(0.52)
SIMD	$n =$ 5428						(0.52)
Tangle	$n = 714$						(0.52)
Twister	$n = 474$						(0.52)
Vortex	$n = 104$						$n =$ 1257
WaMM	$n =$ 1171						(0.52)
Waterfall	(0.52)						(0.52)

Table 8.3: Random distinguishers for SHA-3 candidate functions.

9 Conclusions and future work

9.1 Conclusions based on experimental data

- summary of what we did
- control distinguishers (random-random, hr-de, audio)
- estream (round limited ciphers)
- analysis of Salsa20
- sha3 (round limited hash functions)

- different approach than statistical batteries -> possibly new things
- dynamically adapting distinguisher - both advantage and disadvantage
- comparable to statistical tests, however smaller inputs
- speed: slow learning (more computational power needed), fast distinguishing
- problem with interpreting results

9.2 Proposed future work

- deep analyses instead of wide
- possibilities of longer input
  - READX
  - memory circuit
- tools for interpreting results
  - histogram of outputs in nodes
- fixing functions in layers