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

I'd like to thank Petr for his guidance, enthusiasm and inspiring discussions. I also owe much to my mom and brother for their continuous support. Thank you.

Further thanks goes to all my friends who had to put up with my enthusiasm and numerous research details they may have never asked for $\ddot\smile$.

Last but not least, I'd like to acknowledge the Laboratory of Security and Applied Cryptography and the National Grid Infrastructure MetaCentrum for providing access to their computing and storage facilities.

Abstract

This thesis explores a novel, automated method of creating statistical randomness tests. Tests are created as hardware-like circuits using EACirc, the framework for automatic problem solving based on genetic programming. The improvements and current capabilities of the framework are described along with a set of reference experiments. The framework is then used to assess the randomness of outputs produced by chosen eStream cipher candidates and SHA-3 hash function candidates. Success of the tests generated by EACirc is compared to standard statistical batteries (STS NIST, Dieharder). Results of one chosen case are analysed in detail.

Keywords

statistical randomness, random distinguisher, evolutionary algorithms, genetic programming, software circuit, hash function, SHA-3, stream ciphers, eStream

Contents

1 Introduction

Random data and the notion of randomness are used in in many situations (simulating and modelling complex phenomena, evenly sampling large data sets, ...), but are of utmost importance in cryptography. Random sequences serve as keys, nonces and initialization vectors in many popular cryptographic protocols. An encrypted message not looking sufficiently random might leak some information of the cipher settings and/or the key used. An insufficient randomness of keys and other parameters may cause numerous security vulnerabilities. Therefore, assessing the quality of randomness became a crucial part of cryptographers' work.

This thesis elaborates on distinguishing random and non-random data. Firstly, accounts of the most common method (testing using statistical batteries) is given. This approach has its drawbacks – since the properties and patterns looked for must be specified in advance, the creation of new tests is difficult and requires extensive mathematical skills.

In the next chapter, a novel method of distinguishing non-randomness is given. The proposed approach is based on evolutionary algorithms and utilizes the idea of software-emulated circuits. Its main benefits lie in easy automation and high potential of creating new tests, thus surpassing the disadvantages of statistical batteries. This novel method is implemented in a general problem-solving framework called *EACirc*.

Further chapters use the EACirc framework in practical tests and compare all obtained results with already existing approach using statistical batteries. The experiments are divided into three categories: control experiments checking the sanity of our implementation and used referential data, experiments examining randomness of stream cipher outputs and experiments assessing randomness of hash function outputs. A single case is chosen for a more detailed analysis.

Although most of the implementation improvements and research presented in this thesis was done by myself, EACirc is a result of a greater team[1] where problems and ideas were consulted together. Therefore, I use plural in the thesis text, even though parts not done by myself are properly attributed when mentioned.

At the beginning of 2013, we wrote a scientific paper based on part of the research presented in this thesis (up to chapter 6). The paper was accepted in unabridged version for the 10$^{\text{th}}$ International Conference on Security and Cryptography and will be published in mid-summer 2013. [ŠUM13]

The EACirc framework and results are licensed under The MIT License, copyright © 2012 Centre for Research on Cryptography and Security [Cen]. The source code and documentation is freely available from project's page on *GitHub* [Š+12].

The thesis text was typeset in LaTeX using the *fithesis2* package created by Stanislav Filipčík [Fil09]. The text of this thesis is licensed under a Creative Commons Attribution 3.0 Unported License.

1. The team consists of the following members: Milan Čermák (2012-now), Ondrej Dubovec (2011-2012), Matěj Prišťák (2011-2012), Tobiáš Smolka (2011-now), Marek Sýs (2013-now), Petr Švenda (2008-now), Martin Ukrop (2012-now).

2 Statistical randomness testing

Producing random data by computers is extremely difficult[1], as they are inherently deterministic. Yet it is crucial to be able to distinguish sufficiently random data from the non-random – it may enable us to detect badly designed cryptographic protocols, algorithms or their buggy implementations. The problem comes with the definition of randomness. In truly random data, each fixed subsequence 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 and do not hold for non-random data streams. 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 (e. g. sequence of a hundred zeroes does definitely not satisfy this property), but the probability of randomly generating such a sequence sharply decreases with the increasing length.

To ease the testing of randomness, batteries of statistical randomness tests such as STS NIST or Dieharder have been developed. The main working principle of such batteries is summarized in Figure 2.1. Firstly, the assessed statistical properties of the chosen data are computed (e. g. number of ones versus number of zeroes). Secondly, for each property, the corresponding p-value is computed (the probability of obtaining a test statistic at least as extreme as the one that was actually observed). The final verdict is based on the p-value and the chosen significance level.

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

1. *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."* [vNeu51]

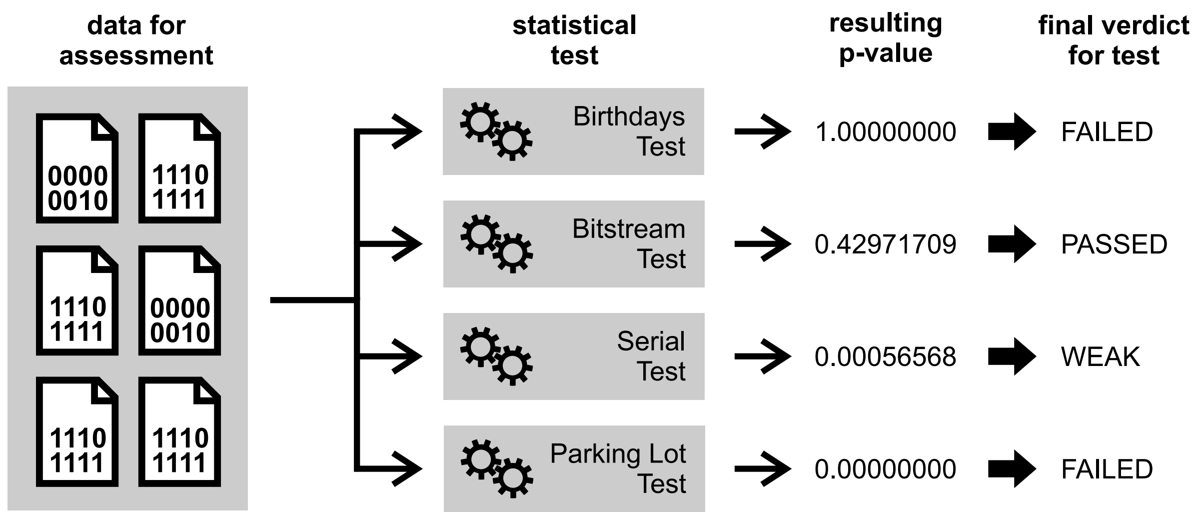

Figure 2.1: Simplified work-flow of Dieharder statistical battery.

- **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 testing process will not adapt to the stream of data with dynamically changing properties.

- **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 extensive mathematical skills.

- **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 testing suites have been developed, some of which are discussed below.

2.1 Statistical Testing 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 motivation 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. [Ran97]

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 [Ruk+10]. 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. Detailed setting used for the purposes of this thesis can be found in section 4.4.

2.2 Diehard Battery of Tests of Randomness

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. [Mar95] 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 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 of 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 the "Swiss army knife of random number test suite", or if you prefer, "the last suite you'll ever ware" for testing random numbers. [Bro04]

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 testing other properties requires beginning the testing process 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-created 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 are dynamically adapting to the testing data;
- 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* – a 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.

In section 3.1 the basic accounts of genetic programming will be given followed by the description of software emulated circuits in section 3.2. Finally, in section 3.3 and section 3.4, the two techniques are combined and working principles and capabilities of EACirc are explained.

3.1 Basic principles of genetic programming

Genetic programming [Ban+97] 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 sufficiently large fitness landscape are applicable (where enough partial solutions exist to facilitate evolution).
- 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 improvements to the individuals may require a high 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.
- In the evolution process, an excessive amount of decisions must be taken (which individuals to breed, what mutations to perform, ...). These are usually done randomly so as not to discriminate particular changes. Randomization on the other hand complicates experiment reproducibility.

To counterweight the drawbacks, it must be noted, that evolutionary algorithms allow us to create solution not just for a particular instance of the problem, but to the whole set of similar problems – we may be trying to evolve a universal solver, rather than the solution itself. This improves the overall 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. We consider solutions in the form of hardware-like circuits with gates (*function nodes*) and a set of wires (*node connectors*). Each node is responsible for the computation of a simple function on its inputs (e. g. binary AND operation). Function 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 Figure 3.1.

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

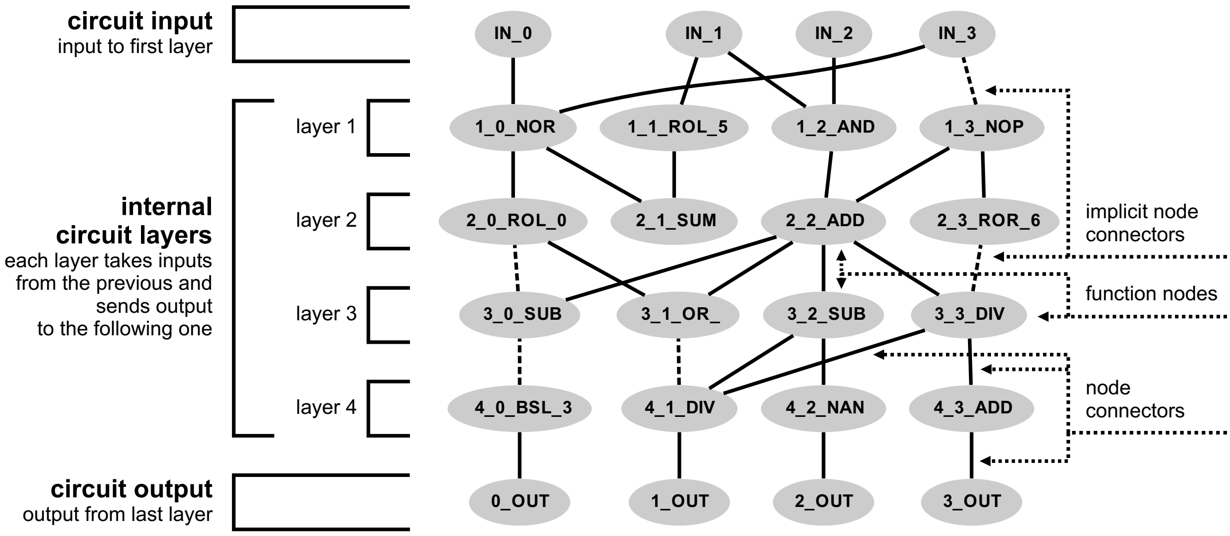

Figure 3.1: Simple example of software-emulated circuit.

- 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 a specific input byte (READX).

It would be sufficient to restrict ourselves to a smaller set of functions (e. g. NAND only), since with such subset we can express arbitrarily complex function. However, the spacial requirements rise with the function complexity. More complex functions in nodes enable us to limit the circuit to significantly smaller number of layers and nodes, while retaining a comparable expressive power. We decided to support a wider variety of functions as an human understandability trade-off.

To some extent, the structure of a software circuit resembles artificial neural networks (deep belief neural networks in particular [HOT06]). 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 in artificial neural networks usually 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 framework tries to evolve a circuit solving the given problem. The candidate population is assessed (how well does each individual solve the problem) and then cross-bred (better circuits having higher probability of surviving). The whole process repeats until the desired success rate is achieved or the required number of generations is evolved. A simplified work-flow of the evolution process can be seen in Figure 3.2.

The initial version of EACirc was created by Petr Švenda at the Laboratory of Security and Applied Cryptography, Masaryk University [Lab]. 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 [Pri12; Dub12]).

Afterwards, the object model of the entire project was redesigned and a handful of new features was added by myself. Most of the code 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 [Wal95]. 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

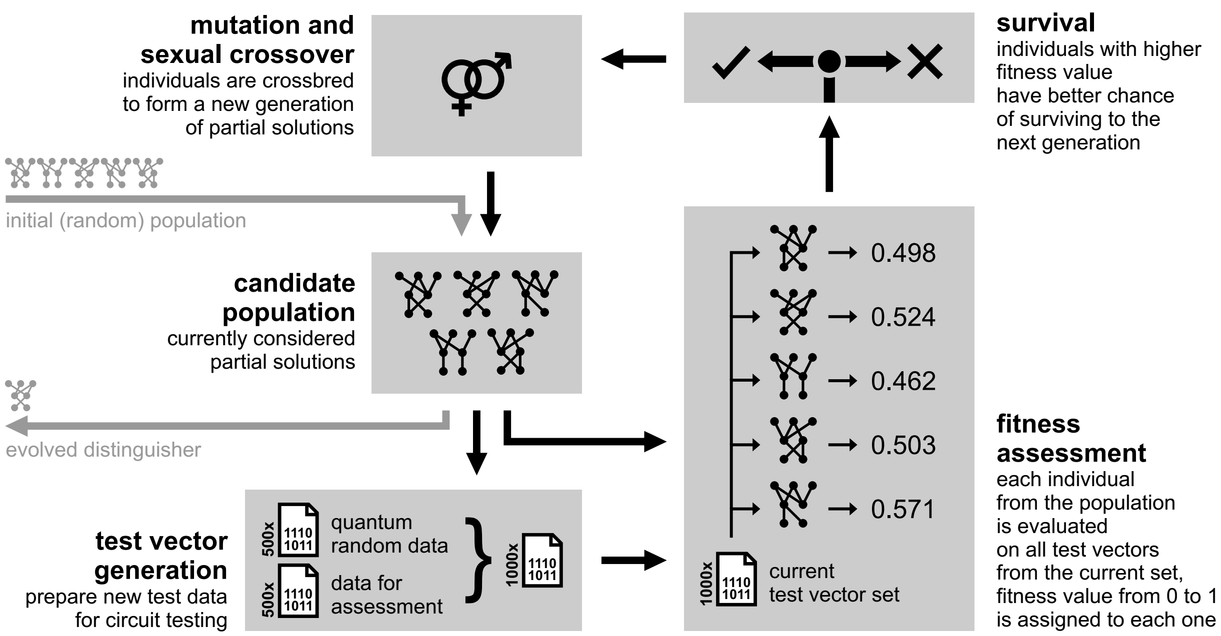

Figure 3.2: Simplified work-flow of the evolution process in EACirc.

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 pairs 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**
  Evolutionary algorithms require a source of randomness to run (determining crossovers, mutation, generating initial population, order of test vectors, ...) 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 [Nas10].

- **XML manipulating library**
  Most of the files produced and processed by the framework are XML-structured files. All these files are handled via *TinyXML*, a simple, small, minimal, C++ XML parser library [Tho00].

- **Static checker**
  The static checker is designed to verify obtained results (evolved circuits) by circumventing both the genetic manipulations and circuit emulator. Although sharing some code with the main framework, it is built as an independent application.

- **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 (a stand-alone C program used for static analysis).

  The DOT graph format can be easily displayed using the *Graphviz* library [Bil+88] and thus facilitates manual analysis done by humans after the computation. To further ease the human analysis, circuits can be *pruned* before export – all disconnected and unused nodes are removed.
  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. [Pri12] 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 [Dub12]. 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.
- **CUDA support**
  EACirc supports nVidia CUDA for circuit evaluation during the computation of individual's fitness. When executed on GPU instead of CPU, the evaluation runtime decreases by the coefficient of about 70.

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 are provided as well.

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. For further details, user and development documentation, see EACirc wiki at *GitHub* [Š+12].

4.1 EACirc settings

Most of the general settings (evolution and circuit parameters) were taken from Matej Prišťák's thesis [Pri12]. The experiments supporting the choice of these parameter 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 a test vector set (learning data) changing every 100^{th} generation. Thus, a total of 300 unique test vector sets are 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 suspected implementation problems, using the READX function is forbidden. All other implemented functions are allowed (see section 3.2).

Each testing set consists of 1 000 independent vectors, exactly half of which is truly random and the other half is generated from the assessed data. According to the previous research, the imbalance in test vector types complicates the learning phase, since the circuits are also trying to learn which type is more frequent in the particular set. 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 2.3 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 with 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 [Cen07] and
- *High Bit Rate Quantum Random Number Generator Service*
  provided by Humboldt University of Berlin, Germany [Nan10].

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 rates 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 is defined as having 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). Such strong distinguishers have a high probability of successfully differentiating new stream samples, since they maintained high success rate for 50 test sets.
- 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.

In some of the experiments, the results were replicated using the static checker with the evolved circuit. These sub-experiments always confirmed the qualities of evolved distinguishers and are therefore not presented explicitly in the result tables.

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 assigns 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 tests 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

Before performing the experiments themselves, we need to acquire reference results – what does it mean streams are indistinguishable from random in out context? Is the referential random data indistinguishable from random? What is the AAM value (see section 4.3) for such distinguishers?

5.1 Looking for non-randomness in quantum random data

The first control experiment tries to distinguish quantum random data from other quantum random data. We use 193 MB of data obtained from Quantum Random Bit Generator Service (for details, see section 4.2). We presume to fail at this and thus establish the randomness of the assessed data stream.

Using the standard statistical batteries confirmed out expectations – all 20 tests of Dieharder passed as well as all 162 tests of STS NIST. Running EACirc with the settings described in section 4.1 yielded the AAM value of 0.52 with runs differing in 3^{rd} or 4^{th} decimal place.

We anticipated that the difference of obtained AAM from the naïve value of 0.50 was influenced by population size and the amount of test vector in a set. This reasoning was based on the following two facts:

- AAM value is based on fitness of the currently best individual in population (it's a maximum fitness) and thus larger populations might have a better chance of getting a score above 0.50.
- With the increasing number of test vectors in a set, the probability of just guessing correctly decreases.

The performed experiments (Table 5.1) confirmed our presumptions: the AAM value decreases with decreasing population size and increasing size of test vector set. We can thus conclude that in our settings the AAM value of 0.52 corresponds to indistinguishable streams.

		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

Secondly, we want to compare quantum random data streams obtained from two different sources (for details, see section 4.2). We prepared 6 independent files of 5 MB from each source and attempted to find a distinguisher for each pair. The initial reading offset was set to 0 in each of the files so that each file produced the same stream every time.

For each pair, the computation was run just once (instead of 30 times). The results are summarized in Table 5.2 – for each pair the average of the maximum population fitness in the generations just after the test vector change is displayed (this would correspond to the AAM value, if 30 runs were performed for each pair).

The results oscillate closely around 0.52 indicating indistinguishable streams (see section 5.1). We can thus conclude that, for our purposes, both sources are equally random and equally reliable. Since no of the tested files expressed any statistically significant deviation from the others, we can use these files 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 Analysing uncompressed audio streams

The third and last of the control experiments compares the set of audio files. We considered a set of 12 files – 3 quantum random data files, 3 uncompressed audio files with white, pink and Brownian noise, the same noise files with intermediate mp3 compression and 3 samples of uncompressed black-metal music.

The quantum random data files had about 5 MB and were turned into a listenable file by adding a WAV header instructing to interpret the data as 2-channel, 16 bit/sample, 44.1 kHz PCM-encoded audio. The 30 seconds (about 5.3 MB) samples of white, pink and Brownian noise in the same audio format were generated using SoX [Bag+91]. The third subset was created from the above-mentioned generated noises by mp3 compression (bitrate of 128 kbps) and decompression back to the PCM-encoded audio. Note that after the lossy MP3 compression the files took about 480 kB each (compared to 5.3 MB of the uncompressed version). The last three were 30 seconds samples of transcendental khaoblack metal by Abbey ov Thelema [Abb12] all taken from the band's promo called *MMXII: Here & Now - At the Threshold ov End Times*.

We again attempted to develop a distinguisher for each pair of these files, again setting initial reading offset to 0 and performing just one execution per file combination. Our hypotheses included the following:

- the quantum random files will be indistinguishable from each other (assumption based on results from section 5.2),
- the noise will be very similar to quantum random data (especially the white noise), but it may be able to be told apart,
- the different noise types will be similar to each other,
- the compressed and decompressed noise will be easily distinguishable from both the uncompressed noise and quantum random data files (even though they cannot be easily differentiated by human ear),
- the provided samples of metal music will be equally easily told apart from the other files, both uncompressed and re-compressed.

The results are presented in the usual way in Table 5.3 (the part below the diagonal is mirrored to facilitate analysis). Based on these values, we accept most of the hypotheses:

- quantum random stream are indistinguishable (average maximum success rate of 0.52, see section 5.1 for details),
- generated white noise is completely indistinguishable from random data files,
- pink and Brownian noise are easily told apart from each other or the quantum random files (success rate generally over 80 %),
- mp3 compression has small, but detectable effect on the sound (although nearly undetectable by unskilled human ear, it successfully shifts the distinguisher success rate to about 0.58 when comparing with an uncompressed noise of the same kind),
- used metal samples can be reliably distinguished from white noise (general success over 80 %), less so from pink and Brownian noise (success rate only around 65 %),
- used metal samples are nearly indistinguishable from each other on the binary level (although the differences are easily detectable by human ear).

		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

Inspired by the research done by Matej Prišťák [Pri12], we analysed randomness of stream cipher outputs. The analysis used EACirc, Dieharder and STS NIST with settings described earlier (chapter 4). We only considered stream ciphers from the recent eStream competition [Eur05], since we could use the unified cipher interface (API) prescribed in the competition.

6.1 Generating stream from cipher outputs

From 34 candidates in the eStream competition, 23 were potentially usable for testing (due to renamed or updated versions, problems with compilation, ...). Out of these, we limited ourselves to only 7 (Decim, Grain, FUBUKI, Hermes, LEX, Salsa20 and TSC), since these had internal structure that allowed for a simple reduction of complexity by reducing a number of internal rounds. For all used ciphers, the implementation from the last successful phase of the competition was taken. The ciphers were tested in unlimited versions and then for all lower number of rounds until reaching indistinguishability from a random stream.

As opposed to previous research, we considered three modes with respect to the frequency of cipher initialization and key change:

- The cipher initialization is performed just once (at the beginning of the computation) and the key is fixed for all the generated test vectors and sets. Even when the sets change, new test vectors are generated using the same key.
- Every test set is generated using different key. All test vectors in a particular set are generated with the same key. The cipher is reinitialized when the key is changed, i. e. at set change.
- Every test vector is generated using a different key and a freshly initialized cipher.

For every setting, each mode was considered separately. Keys were generated randomly and initialization vectors and plaintexts were fixed to binary zeroes.

6.2 Results interpretation

Before coming to the results, two things must be noted:

- The internal structure of LEX function responds specifically to the limitation of number of rounds. LEX prepares 4 output bytes during every round (other bytes default to binary zero). Limitation of internal rounds therefore only limits the number of output bytes, not their strength/randomness.
- During the first 8 rounds, TSC only fills internal memory structures and thus produces no output. The output of these rounds is therefore a default stream of binary zeroes. Such stream causes 4 of the Dieharder tests to get stuck – the score from the remaining 16 tests is presented with an asterisk (*).

The results of the described distinguisher experiments can be seen in Table 6.3 to Table 6.7. The value of n indicates average number of generations needed to reach a population of strong distinguishers while the number in parentheses expresses the AAM value in case such a generation has not been found. For detailed description of the data meanings see section 4.3. We also remind the reader that the value of 0.52 indicates the stream is indistinguishable from random (for reasoning, see section 5.1).

In summary, the results indicate that in this case, EACirc performs more or less the same as standard statistical batteries. Although it did not always find a population of strong distinguishers, it found a population with significantly better success rate than random guessing in most of the cases (Decim being the most prominent exception). Dieharder sometimes performed better than STS NIST, but it has to be taken into consideration that it is newer and made decision based on a much larger data sample. In general, both statistical batteries processed longer stream than EACirc (for detailed numbers see chapter 4). Regarding the matters of speed, EACirc had a comparably longer learning phase, but usually provided a distinguisher working working faster than statistical batteries.

# 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. For the notes on fields marked with asterisk, see section 6.2.

7 Analysis of Salsa20 output stream

In this chapter we analyse one selected case from the previous experiment in a more detailed manner. We study the dependence of distinguisher success rate on the number of generations already computed. Further attention is paid to the evolved circuit and the statistical properties it uses to draw the final verdict (random vs. non-random).

7.1 Distinguisher success rate

The general relationship between fitness value and the number of evolved generations in evolutionary algorithms is very specific – a typical example can be seen in Figure 7.1. This jaw-like curve represents the the success rate of a circuit trying to distinguish two independent random streams. The success rate rises, during the period when the test vector set remains unchanged (100 generations in our setting) and then suddenly drops after the set change. As can be easily seen in the graph, the behaviour repeats almost periodically. However, notice that the success rate does never exceed the value of 0.55.

This is caused by the circuit *over-learning* on a specific test vector set (circuits are learning to distinguish this particular set instead of general characteristics of the streams). On one hand, this phenomenon can be easily suppressed by changing the test vectors more frequently or increasing the number of vecors in a set. On the other, higher test set change frequency of more vectors would increase computational complexity (generating a new set of test vectors is by far the most computationally-intensive phase at the moment). Also, changing the test vectors too fast may hamper the ability of evolution to discover dependency in the input bits. Therefore a reasonable trade-off is used.

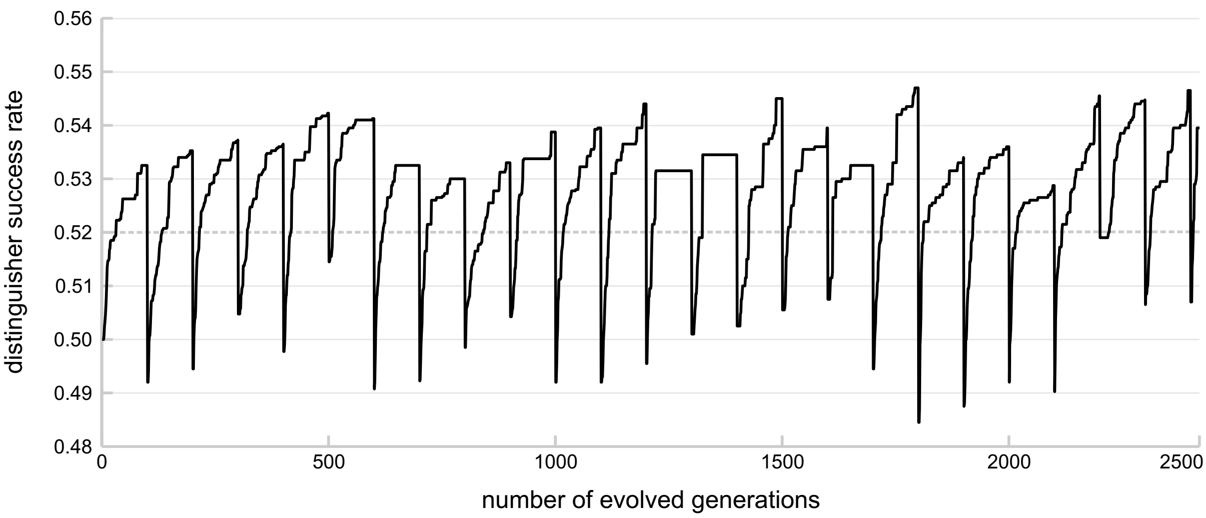

Figure 7.1: Circuit success rate for control experiment trying to distinguish two streams of quantum random data (note the shifted scale on y-axis). The dotted line represents the value of 0.52 (stream indistinguishable from random).

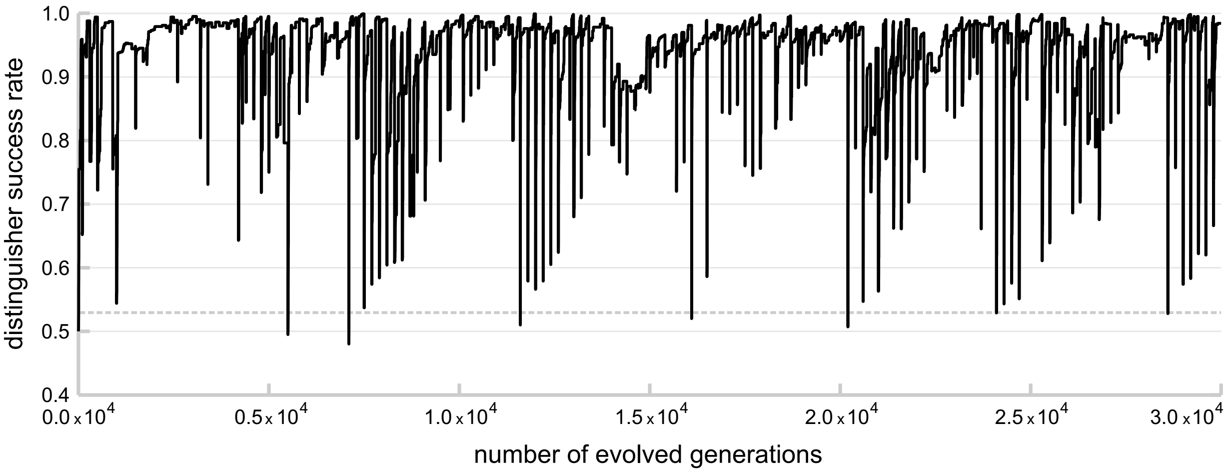

Figure 7.2: Circuit success rate for distinguishing Salsa20 limited to 2 rounds from quantum random data (note the shifted scale on y-axis). The dotted line represents the value of 0.52 (stream indistinguishable from random).

In Figure 7.2 we see similar relationship for circuit distinguishing Salsa20 cipher limited to 2 rounds. The over-learning tendency (repeating continual rise and sudden drop) is partly present as well, but in contrast to the previous case the circuits success rate reaches much higher values. Even if not evolving a strong distinguisher, this would be a sufficient evidence for non-randomness of Salsa20 output stream.

We can further notice that after initial fluctuations the circuit success rate show another periodic behaviour about every 4 000 generations. The circuit stabilises at distinguishing the Salsa20 output and then suddenly drops back to about a success of random guessing. It than gets better again and after about 4 000 generations (equivalent to about 450 KB of data) drops again. This behaviour is specific to Salsa20 and its source probably comes the the cipher design. A detailed analysis will be the part of our future work.

7.2 Evolved circuits

Other type of detailed study of Salsa20 limited to 2 rounds included the evolved distinguishers. An example interpretation of one such circuit is demonstrated on the circuit shown in Figure 7.3. We took an evolved distinguisher circuit, pruned it (removing all nodes not participating in computing the final fitness), generated 1 000 000 random input sequences for the circuit and inspected the distribution of values coming from every node. From the distribution we concluded the following:

- The node 1_2_DIV located divides the value of IN_2 by IN_7, IN_12 and IN_14 resulting in a byte with very low Hamming weight.
- The node 1_3_OR makes logical disjunction of 7 different input bytes resulting in a byte with very high Hamming weight.
- The result of node 2_2_NOR is a negation 1_2_DIV. Its Hamming weight should

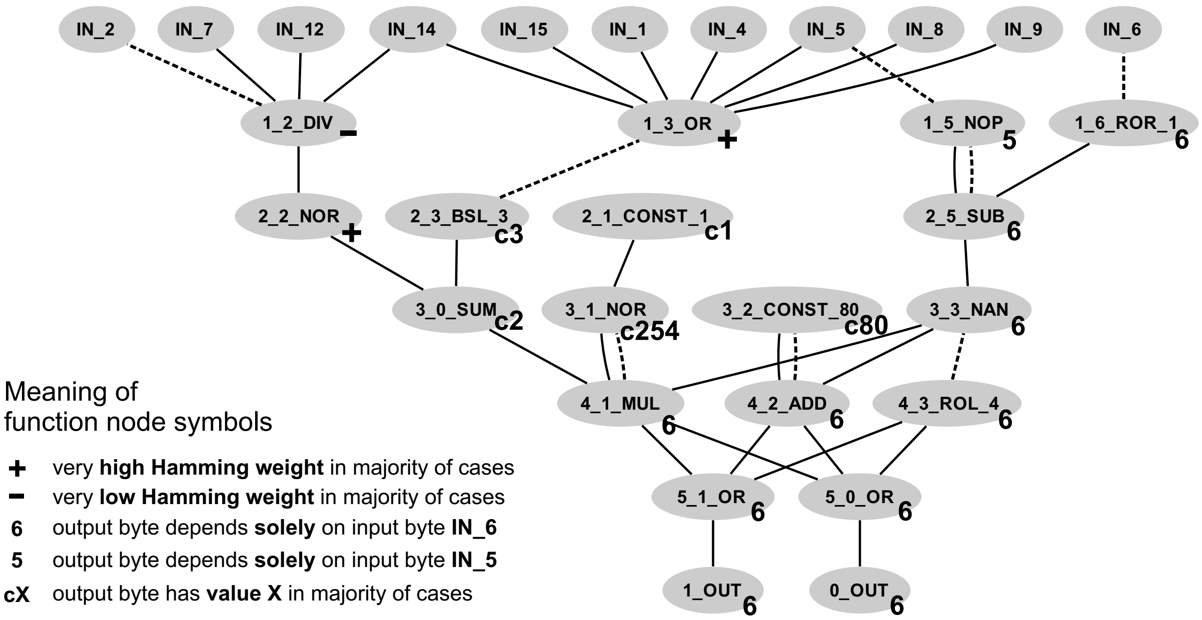

Figure 7.3: Analysis of a distinguisher evolved for Salsa20 limited to 2 rounds (pruned version of the circuit is displayed).

therefore be generally very high (in 99.97% of the tested 1 000 000 input sequences it produced binary one).

- The node 2_3_bsl_3 results in the last three bits of 1_3_or and thus most likely outputs binary representation of 3 (98.43% of the tested cases).
- The output of 3_0_sum has low Hamming weight (due to overflow in addition), usually being a binary representation of 2 (98.40% of tested cases).
- Nodes 2_1_const_1, 3_1_nor and 3_2_const_80 produce constant output.
- The node 1_5_nop takes in_5 without change and the node 1_6_ror_1 is solely dependent on input byte in_6.
- The node 2_5_sub computes 1_5_nop - 1_5_nop - 1_6_ror_1 resulting in value solely dependent on in_6.

Following the reasoning the the above list, we can conclude that both output bytes (1_out and 2_out) depend solely on 7^{th} input byte (in_6). Circuits evolved in parallel runs exhibited very similar behaviour – in may of them, the output bytes (and thus the final verdict) depended only on the 7^{th} input byte. It is difficult to tell what is the exact form of this weakness, but it draws out attention to the ever-mentioned byte 7. It definitely implies a possible design flaw in Salsa20 limited to 2 rounds influencing the randomness of every 7^{th} output byte.

After analysing the output of internal nodes, we concentrated on the circuit output and its final verdict (random vs. non-random). We used circuit from Figure 7.3, where both output byte produce the same value (this happened in most of the parallel runs). We made statistics of all values output by the circuit for random data stream and Salsa20

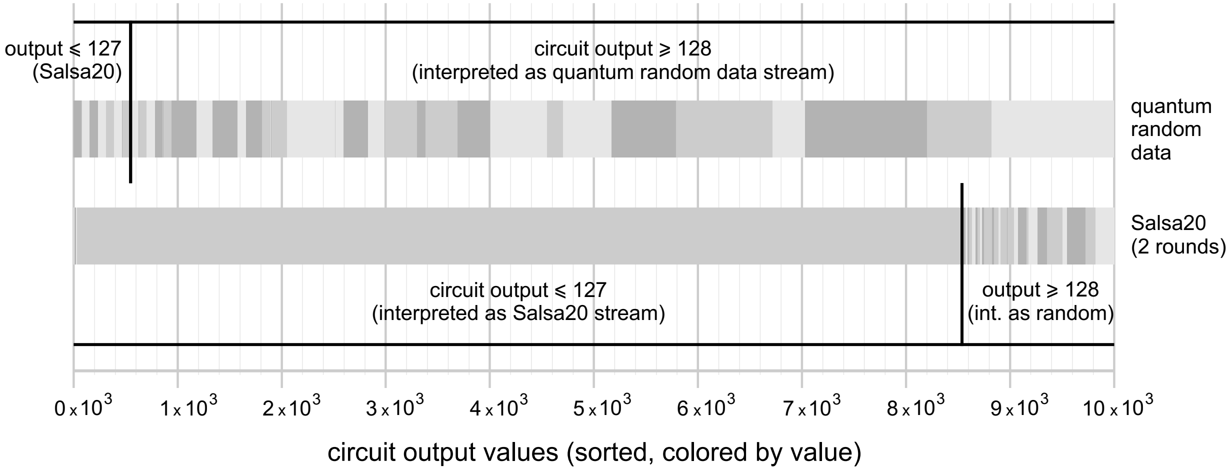

Figure 7.4: Frequency of circuit output values for the circuit displayed in Figure 7.3 for 2 streams. Evaluator interprets values less than 128 as stream originating from the cipher, values above or equal to 128 as stream originating from random source.

(limited to 2 rounds) output stream. In both cases, 1 000 000 input sequences were used. The frequencies of output values are plotted in Figure 7.4 in form of a cumulative bar chart. Each coloured block represents a single value, its width proportional to the number of times this value was output.

The evaluator used interpreted the circuit outputs as follows:

- if the numerical value of the output byte was less or equal to 127, the stream was concluded to originate from Salsa20;
- otherwise (numerical value of the output byte equal to or greater than 128) it was concluded to be from a random source.

Two important conclusions can be drawn from the data in Figure 7.4. Firstly, the circuit succeeds in distinguishing Salsa20 limited to 2 rounds most of the times, but not always. Secondly, while the distribution of output values in case of random stream is more or less even, in case of Salsa20 the value of 126 was far more frequent (85.02%). From the latter fact, we could possibly backtrack through the circuit to establish the exact bits in the 7th input byte causing the stream to be distinguishable from random, if such analysis was required.

8 Distinguishing hash outputs from random stream

Similar experiments as done in chapter 6 were performed on candidate hash functions from SHA-3 competition [Nat07] (inspired by research done by Ondrej Dubovec [Dub12]). As in eStream ciphers, we utilized the unified hash function interface (API) prescribed in the competition. We analysed the randomness of streams produced as a concatenation of hash digests. The usual settings for EACirc, Dieharder and STS NIST were used (see chapter 4 for details).

8.1 Generating stream from hash function outputs

From 64 hash functions that entered the competition, 51 were selected to the first round. Out of these, 42 were potentially usable for testing (due to source code size, speed and compilation problems). The implementations (taken from the last successful phase of the competition) and modifications limiting the number of rounds in these functions were taken over from previous work [Dub12] and revised. In the end, 18 most promising candidates were chosen: ARIRANG, Aurora, Blake, Cheetah, CubeHash, DCH, Dynamic SHA, Dynamic SHA2, ECHO, Grøstl, Hamsi, JH, Lesamnta, Luffa, MD6, SIMD, Tangle, and Twister. These were the candidates fulfilling the following two requirements:

- the hash functions could be effortlessly limited in complexity by decreasing the number of internal rounds and
- while the full (unlimited) version produced a random-looking output, their most limited version did not.

All these hash function were subsequently tested unlimited and then for all lower number of rounds until reaching indistinguishability from a random stream.

As opposed to the work we were inspired by, we generated continuous output stream by hashing a simple 4-byte counter starting from a randomly generated value. We obtained a 256-bit digest, which we cut in half to produce 2 independent test vector inputs of 16 bytes each. In case of generating a continuous stream (for the purposes of Dieharder and STS NIST), we concatenated the digests.

8.2 Determining optimal set change frequency

In the previous work [Dub12], Ondrej Dubovec claims that optimal test set change frequency is once per 10 generations. However, not enough evidence supporting this hypothesis is provided. Since in other experiments in this thesis we changed test vector set once in 100 generations (a setting taken over from Matej Prišťák's work [Pri12]), we decided to re-test the optimality of this setting.

We performed reference computation on Dynamic SHA limited to 5 rounds. The AAM values for the usual 30 000 generations along with the estimate runtime are displayed in the first two rows of Table 8.1. From these results it seems that decreasing the test set use period increases the success rate, which we considered slightly counter-intuitive (since the circuits work with the same vectors only for a very limited time).

It must noted, however, that circuits in these experiments had extremely different amounts of data – only 30 sets of test vectors in case of changing the set once in 1 000 generations compared to astounding 6 000 different test sets when changing every 5th generation. To even out these differences, we re-run the experiments while using exactly 300 unique test sets in each case. The results can be seen in the bottom two lines of the same table. In this case we see a completely reversed behaviour.

All in all, we decided to keep the setting of changing the test set every 100th generation. Changing the test set more frequently would result in the generation of more test vectors, which is computationally very expensive. The chosen frequency is a acceptable trade-off between circuit success rate and required run time. Furthermore, we retain settings similar to the previous experiments, which facilitates the comparison of the results.

	change frequency for test vector set							
	5	10	20	50	100	200	500	1000
30 000 gen.	(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: The AAM and estimate run-time for different test vector change frequencies.

8.3 Results interpretation

The results of the above-introduced experiments are summarized in Table 8.2 to Table 8.17 in the usual way. The value of n indicates average number of generations needed to reach a population of strong distinguishers while the number in parentheses expresses the AAM value in case such a generation has not been found. For detailed description of the data meanings see section 4.3. We also remind the reader that the value of 0.52 indicates the stream is indistinguishable from random (for reasoning, see section 5.1).

Notice the asterisk in Dieharder results in Table 8.7 and Table 8.16. In these cases, the hash functions produced no output at all (output stream defaulted to binary zero). This caused 4 Dieharder tests to get stuck, effectively reducing the number of tests to 16 (situation similar to Table 6.7, see section 6.2).

In summary, the results indicate that in this case, EACirc performs slightly worse than standard statistical batteries. Although in most of the cases it either found a population of strong distinguishers or a statistically significant variation from a neutral success rate of 0.52, it can be seen that it often failed in the last round successfully distinguished by statistical batteries. Once again, when interpreting these results, we must be aware of the imbalance of test data available to statistical batteries and EACirc (for detailed numbers see chapter 4).

Another observation worth noting is the consistency of the results within the 30 runs of the same experiment. Previously (mainly chapter 6), all the results within an experiment were consistent (all 30 runs reached more or less the same results). The computations presented in this chapter display the variations characteristic to evolutionary algorithms – only some of the runs are successful (the randomized evolution in the other just did not succeed). If only some of the parallel runs reached the population of strong distinguishers, their number is denoted in curly brackets after the average generation. It may be interesting to consider a larger amount of runs in border cases, where statistical batteries were successful but EACirc was not.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 694$
1	0.0	0	$n = 707$
2	0.0	0	$n = 467$
3	0.0	0	$n = 1071$
4	20.0	161	(0.52)

Table 8.2: Random distinguishers for ARIRANG output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	1	$n = 5614$
1	0.0	1	$n = 4101$ {1}
2	0.5	132	$n = 13201$ {1}
3	0.5	132	(0.52)
4	20.0	160	(0.52)
17	19.5	161	(0.52)

Table 8.3: Random distinguishers for Aurora output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 474$
1	0.0	0	(0.52)
2	20.0	162	(0.52)
14	20.0	159	(0.52)

Table 8.4: Random distinguishers for Blake output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	1	$n = 181$
1	0.0	1	$n = 574$
2	0.0	0	$n = 708$
3	0.0	0	$n = 14659$ {12}
4	0.0	1	$n = 16870$ {10}
5	0.0	1	(0.52)
6	20.0	161	(0.52)
16	20.0	162	(0.52)

Table 8.5: Random distinguishers for Cheetah output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 104$
1	0.0	0	(0.52)
2	20.0	161	(0.52)
8	20.0	162	(0.52)

Table 8.6: Random distinguishers for CubeHash output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0*	0	$n = 104$
1	0.0*	0	$n = 17260$ {5}
2	19.5	162	(0.52)
4	20.0	162	(0.52)

Table 8.7: Random distinguishers for DCH output. For the notes on fields marked with asterisk, see section 8.3.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 484$
1	0.0	0	$n = 2337$
2	0.0	1	$n = 1773$
3	0.0	1	$n = 12731$ {10}
4	0.0	18	(0.74)
5	0.5	18	(0.61)
6	3.0	16	(0.59)
7	3.0	17	(0.59)
8	20.0	162	(0.52)
16	20.0	160	(0.52)

Table 8.8: Random distinguishers for Dynamic SHA output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	1.0	1	$n = 15886$ {13}
2	1.0	1	(0.74)
3	0.0	1	(0.75)
4	0.0	1	(0.57)
5	3.5	1	(0.60)
6	3.5	1	(0.60)
7	4.0	2	(0.61)
8	4.0	2	(0.60)
9	3.5	5	(0.61)
10	3.5	5	(0.61)
11	11.5	46	(0.52)
12	11.5	46	(0.52)
13	20.0	161	(0.52)
17	20.0	161	(0.52)

Table 8.9: Random distinguishers for Dynamic SHA2 output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
1	9.0	24	$n = 10501$ {3}
2	9.0	24	(0.52)
3	20.0	161	(0.52)
8	20.0	161	(0.52)

Table 8.10: Random distinguishers for ECHO output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 6285$ {25}
1	0.0	0	(0.58)
2	12.5	52	(0.58)
3	12.5	52	(0.52)
4	20.0	162	(0.52)
10	20.0	162	(0.52)

Table 8.11: Random distinguishers for Grøstl output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	2.5	1	$n = 10376$ {24}
1	2.5	1	(0.52)
2	19.5	161	(0.52)
3	20.0	162	(0.52)

Table 8.12: Random distinguishers for Hamsi output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 581$
1	0.0	0	$n = 4397$
2	0.0	1	$n = 5984$
3	0.0	1	$n = 3674$
4	0.0	1	$n = 1748$
5	0.0	3	$n = 784$
6	0.0	3	$n = 5040$ {28}
7	20.0	161	(0.52)
42	20.0	162	(0.52)

Table 8.13: Random distinguishers for JH output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 791$
1	0.0	0	$n = 568$
2	0.0	0	$n = 504$
3	0.0	0	(0.52)
4	20.0	162	(0.52)
32	20.0	162	(0.52)

Table 8.14: Random distinguishers for Lesamnta output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 604$
1	0.0	0	$n = 1073$ {29}
2	0.0	1	$n = 2074$
3	0.0	1	$n = 3735$ {29}
4	0.0	4	(0.75)
5	0.0	3	(0.75)
6	0.0	10	(0.74)
7	6.0	11	(0.74)
8	20.0	161	(0.52)

Table 8.15: Random distinguishers for Luffa output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0*	0	$n = 101$
1	0.0*	0	$n = 1281$
2	0.0	0	$n = 1084$
3	0.0	0	$n = 631$
4	0.0	0	$n = 781$
5	0.0	0	$n = 13020$ {21}
6	0.0	1	(0.88)
7	0.0	1	(0.65)
8	17.5	18	(0.53)
9	17.5	18	(0.52)
10	20.0	160	(0.52)
104	20.0	162	(0.52)

Table 8.16: Random distinguishers for MD6 output. For the notes on fields marked with asterisk, see section 8.3.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 474$
1	0.0	0	$n = 718$
2	0.0	0	$n = 524$
3	0.0	0	$n = 1247$
4	0.0	0	$n = 1334$
5	0.0	0	$n = 411$
6	0.0	0	$n = 524$
7	0.0	0	(0.52)
8	20.0	161	(0.52)
9	20.0	162	(0.52)

Table 8.17: Random distinguishers for Twister output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	1	$n = 4697$ {28}
1	0.0	1	(0.52)
2	19.5	162	(0.52)
4	19.5	161	(0.52)

Table 8.18: Random distinguishers for SIMD output.

# of rounds	Dieharder (x/20)	Sts Nist (x/162)	EACirc (AAM)
0	0.0	0	$n = 714$
1	0.0	0	$n = 6868$
2	0.0	1	$n = 2518$
3	0.0	1	(0.85)
4	1.0	2	(0.84)
5	1.0	2	(0.80)
10	3.5	4	(0.64)
11	3.0	4	(0.63)
12	3.0	4	(0.64)
13	4.0	4	(0.64)
14	4.0	4	(0.64)
15	3.0	5	(0.64)
16	3.0	5	(0.64)
17	4.5	27	(0.60)
18	4.5	27	(0.60)
19	6.0	36	(0.60)
20	5.5	39	(0.60)
21	10.5	91	(0.54)
22	10.5	90	(0.54)
23	19.0	161	(0.52)
24	20.0	161	(0.52)
80	20.0	161	(0.52)

Table 8.19: Random distinguishers for Tangle output.

9 Conclusions and future work

This work explored automated methods of creating statistical randomness tests. Tests were created as hardware-like circuits using EACirc, framework for automatic problem solving based on genetic programming principles. The complete codes of EACirc were revised and significantly improved and support for multi-platform compilation was added. The computation is now perfectly deterministic enabling recommencing of older runs and easy experiment replication. The object model of EACirc was enhanced to allow for better adaptability and effortless integration of new projects and evaluators. For independent verification of achieved results, static checker was developed. I summary, several thousands of lines of code and documentation were created.

Capabilities of the framework were checked by numerous reference experiments. The assumed behaviour when trying to distinguish two sets of quantum random data (even from different sources) was confirmed. A set of uncompressed audio files was confronted with various types of noise and random data.

After performing these control experiments, cryptographically interesting applications of randomness were investigated. The randomness of 7 different eStream cipher outputs (produced in three different modes) was assessed. The evaluation was done both using the proposed automated method (EACirc) and utilising standard statistical batteries (Dieharder, STS NIST) and the results were compared. An analogical set of experiments was performed on 18 SHA-3 candidate hash functions. EACirc results for Salsa20 were thoroughly analysed, demonstrating the usage of information provided by EACirc.

Part of the research was summarized in a scientific paper that was accepted for the 10^{th} International Conference on Security and Cryptography. [ŠUM13]

9.1 Conclusions based on experimental data

Based on our experience and the experimental results obtained, we can draw several conclusions concerning the proposed method of automatic randomness test generation:

- **Success rate**
  From the point of success rate, the proposed method is generally comparable to the standard sets of statistical batteries. Results sometimes differ in border cases, in favour of statistical batteries. These border cases should be the goal of further research as the differences may be caused by improper settings and/or insufficient computation time. The difference may also lie in unequal input sample lengths (see below).
- **Amount of data used**
  In general, smaller data sample was provided to EACirc (at most, we used about 2.5 MB) than to statistical batteries (about 12 MB in case of STS NIST and more than 200 MB in case of Dieharder). Note that some test may provide indication of failure even when less data is available.

- **Atypical approach**
  The proposed method uses a significantly different approach to detect non-randomness compared to statistical batteries. It does not require prior knowledge of specific data properties – instead, it tries to deduce these properties by itself. Therefore, possibilities of using yet unknown data properties arises. This, however, was not conclusively proven, since we have done a wide analysis instead of concentrating on the best possible performance for a particular function.
- **Limited input scope**
  Since the distinguisher circuits only process small parts of the input at a time, this approach may by unable to detect non-randomness present in the global scale. Enabling the circuit to process longer inputs would alleviate this drawback.
- **Speed and complexity**
  The proposed evolution-based approach has a very slow (and computation-intensive) learning phase compared to the use of statistical batteries. Nevertheless, when a working distinguisher is found, assessing further data is very fast.
- **Dynamically adapting distinguishers**
  While tests from standard statistical batteries look for a predefined evidence of non-randomness, distinguishers evolved by EACirc dynamically adapt to the data stream. Thus, if a data stream changes its properties, the test will evolve accordingly (predefined statistical tests never change).
- **Results interpretation**
  On one hand, dynamically adapting tests present a huge disadvantage when interpreting their results – it may be very difficult for humans to analyse, on what data properties is the distinguisher basing its verdict. On the other hand, statistical tests only inform of the data's global characteristics (e. g. there are much more ones than zeroes), while the distinguisher circuits may be a little more specific (e. g. every 4^{th} byte has a higher Hamming weight than it should).

9.2 Proposed future work

As could be seen in chapter 6 and chapter 8, EACirc still falls behind standard statistical batteries in some cases. In the future work we will concentrate on those border cases were EACirc is outperformed.

Another primary goal for us will be enabling the circuit to process longer inputs and thus detect more global interdependencies. We consider several scenarios of achieving this. To name but a few, we may re-implement the READX function that reads arbitrary input byte. Other method would be to implement a kind of *memory* for the circuit, which would enable the transfer of information when processing longer inputs.

Another interesting idea is to explore the range of functions allowed in the circuit nodes. On one hand, we may allow more complex data processing in a single node – sequences extracted from the byte-code of the analysed stream cipher/hash function may be used. On the other hand, we may limit the range of allowed functions to but a few, e. g. only and, or and not, as such a small set is sufficient to express arbitrarily complex function if given enough space.

Furthermore, we plan to perform deeper analysis of the obtained results with respect to the tested stream ciphers and hash functions. For this, new tools for interpreting the results will have to be developed (e. g. statistical analyser of the node outputs in evolved circuits).

Bibliography

[Abb12] Abbey ov Thelema, *MMXII: Here & Now - At the Threshold ov End Times*, 2012. [Online]. Available: http://bandzone.cz/abbeyovthelema (visited on 05/04/2013).

[Bag+91] C. Bagwell *et al.* (1991). SoX, the Swiss Army knife of sound processing programs, [Online]. Available: http://sox.sourceforge.net/ (visited on 05/04/2013).

[Ban+97] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, "Genetic Programming: An Introduction, On the Automatic Evolution of Computer Programs and Its Applications", 1997.

[Bil+88] A. Bilgin, D. Caldwell, J. Ellson, E. Gansner, Y. Hu, S. North, *et al.* (1988). Graphviz, Graph Visualization Software, AT&T Research, [Online]. Available: http://www.graphviz.org/ (visited on 05/04/2013).

[Bro04] R. G. Brown. (2004). Dieharder, A Random Number Test Suite. version Version 3.31.1, Duke University Physics Department, [Online]. Available: http://www.phy.duke.edu/~rgb/General/dieharder.php (visited on 05/03/2013).

[Cen] Centre for Research on Cryptography and Security. [Online]. Available: http://www.fi.muni.cz/crocs (visited on 05/16/2013).

[Cen07] Centre for Informatics and Computing. (2007). Quantum Random Bit Generator Service, Ruđer Bošković Institute, Zagreb, [Online]. Available: http://random.irb.hr/index.php (visited on 05/03/2013).

[Dub12] O. Dubovec, "Automated search for dependencies in SHA-3 hash function candidates", bachelor thesis, Faculty of Informatics Masaryk University, 2012. [Online]. Available: http://is.muni.cz/th/324866/fi_b_a2/ (visited on 05/04/2013).

[Eur05] European Network of Excellence for Cryptology. (2005). eStream project, Call for stream cipher primitives, [Online]. Available: http://www.ecrypt.eu.org/stream/call/ (visited on 05/04/2013).

[Fil09] S. Filipčík, "LaTeX Thesis Style", bachelor thesis, Faculty of Informatics Masaryk University, 2009. [Online]. Available: http://is.muni.cz/th/173173/fi_b/ (visited on 05/04/2013).

[HOT06] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets", *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[Lab] Laboratory of Security and Applied Cryptography, Faculty of Informatics Masaryk University. [Online]. Available: http://www.fi.muni.cz/research/laboratories/labak/ (visited on 05/07/2013).

[Mar95] G. Marsaglia. (1995). Diehard battery of tests of randomness, Floridan State University, [Online]. Available: http://www.stat.fsu.edu/pub/diehard/ (visited on 05/03/2013).

[Nan10] Nano-Optics groups (Department of Physics) and PicoQuant GmbH. (2010). High bit rate quantum random number generator service, Humboldt University of Berlin, [Online]. Available: http://qrng.physik.hu-berlin.de/ (visited on 05/03/2013).

[Nas10] P. Nash. (2010). CATCH, C++ Automated Test Cases in Headers, [Online]. Available: http://github.com/philsquared/Catch (visited on 05/04/2013).

[Nat07] National Institute for Standards and Technology. (2007). SHA-3, Cryptographic hash algorithm competition, [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html (visited on 05/04/2013).

[Pri12] M. Prišťák, "Automated search for dependencies in eStream stream ciphers", master thesis, Faculty of Informatics Masaryk University, 2012. [Online]. Available: http://is.muni.cz/th/172546/fi_m/ (visited on 05/04/2013).

[Ran97] Random Number Generation Technical Working Group. (1997). Statistical test suite. version Version 2.1.1, National Institute for Standards and Technology, [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/index.html (visited on 05/03/2013).

[Ruk+10] A. Rukhin *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", *NIST Special Publication 800-22rev1a*, 2010. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf.

[Š+12] P. Švenda, M. Ukrop, M. Prišťák, *et al.* (2012). Eacirc, Framework for autmatic search for problem solving circuit via evolutionary algorithms, Laboratory of Security and Applied Cryptography, Masaryk University, [Online]. Available: http://github.com/petrs/EACirc (visited on 05/04/2013).

[ŠM13] P. Švenda and V. Matyáš, "On the origin of yet another channel", presented at the Twenty-first International Workshop on Security Protocols (Mar. 19, 2013), accepted for publishing, Faculty of Informatics Masaryk University, Springer, 2013. [Online]. Available: http://spw.stca.herts.ac.uk/ (visited on 05/04/2013).

[ŠUM13] P. Švenda, M. Ukrop, and V. Matyáš, "Towards cryptographic function distinguishers with evolutionary circuits", presented at the 10^{th} International Conference on Security and Cryptography, Laboratory of Security and Applied Cryptography, Masaryk University, 2013. [Online]. Available: http://www.secrypt.icete.org/?y=2013 (visited on 05/18/2013).

[Tho00] L. Thomason. (2000). TinyXML, simple, small, C++ XML parser, [Online]. Available: http://www.grinninglizard.com/tinyxml/ (visited on 05/04/2013).

[vNeu51] J. von Neumann, "Various techniques used in connection with random digits", *Applied Mathematics Series*, no. 12, pp. 36–38, 1951.

[Wal95] M. Wall. (1995). GAlib, A C++ Library of Genetic Algorithm Components, Massachusetts Institute of Technology, [Online]. Available: http://lancet.mit.edu/ga/ (visited on 05/04/2013).

A Data attachment

The data attachment contains source codes and most experimental results organized in the following structure:

- `eacirc`
  source codes of EACirc (copy of entire project repository with master commit `4b92ed9e36` from 2013-05-19)
- `eacirc-wiki`
  project's wiki-based documentation (copy of entire wiki repository with master commit `2817031fac` from 2013-05-05)
- `data-estream`
  experimental results related to distinguishing eStream ciphers output (chapter 6), further divided into subdirectories according to the testing method (`eacirc`, `dieharder` and `sts-nist`)
- `data-sha3`
  experimental results related to distinguishing SHA-3 hash functions output (chapter 8), further divided into subdirectories according to the testing method (`eacirc`, `dieharder` and `sts-nist`)
- `data-other`
  experimental results from EACirc for other performed tests along with full audio samples used for experiment described in section 5.3
- `thesis-src`
  thesis text source files including bibliography and used images