Determining Cryptographic Distinguishers for eStream and SHA-3 Candidate Functions with Evolutionary Circuits

Petr Švenda, Martin Ukrop, and Vashek Matyáš

Masaryk University, Brno, Czech Republic,
{svenda,xukrop,matyas}@fi.muni.cz

Abstract. Cryptanalysis of a cryptographic function usually requires advanced cryptanalytical skills and extensive amount of human labor with an option of using randomness testing suites like STS NIST [15] or Dieharder [3]. These can be applied to test statistical properties of cryptographic function outputs. We propose a more open approach based on software circuit that acts as a testing function automatically evolved by a stochastic optimization algorithm[1]. Information leaked during cryptographic function evaluation is used to find a distinguisher [9] of outputs produced by 25 candidate algorithms for eStream and SHA-3 competition from truly random sequences. We obtained similar results (with some exceptions) as those produced by STS NIST and Dieharder tests w.r.t. the number of rounds of the inspected algorithm.

1 Introduction

Typical cryptanalytical approach against a new cryptographic function is usually based on application of various statistical testing tools (e.g., STS NIST [15], Dieharder [3]) as the first step. Then follows application of established cryptanalytical procedures (algorithmic attacks, differential cryptanalysis, etc.) combined with an in-depth knowledge of the inspected function. This, however, usually requires extensive human cryptanalytical labor.

General statistical testing can be at least partly automated and easy to apply, but will detect only the most visible defects in the function design. Additionally, statistical testing tools are limited to a predefined set of statistical tests. That on one hand makes the follow-up analytical work easier if the function fails a certain test, yet on the other hand severely limits the potential to detect other defects.

We propose a novel approach that can be used in a similar manner as general statistical testing suites, but additionally provides the possibility to automatically construct new tests. Every test is represented by an emulated hardware-like circuit. Evolutionary algorithms are used to design the circuit layout. Although such an automated tool will not (at least for the moment) outperform a skilled cryptographer, it brings two major advantages:

[1] This paper is significantly extended version of results presented in [18].

– It can be applied automatically against multiple cryptographic functions with no additional human labor – working implementation of the inspected function is sufficient. Cryptographic function competitions (e.g., SHA-3 [1], eStream [8]) are especially suitable due to standardized interface.
– Novel and/or unusual information leakage "side channels" may be used. The proposed approach requires no pre-selection of function parts, input/output bits or used statistics – these decisions are left for the evolutionary algorithm.

We tested our idea by evolving random distinguishers for several eStream and SHA-3 candidate functions. To assess the success of this method, we focused on functions with inner structure containing repeated rounds. By gradually increasing the number of rounds, one can identify the point where this approach still provides results (i.e., the function output can be distinguished with probability significantly better than random guessing). Results are very similar to those obtained from STS NIST and Dieharder statistical test suites w.r.t. the number of rounds of the inspected function. The implementation of the whole framework is available as an open-source project EACirc [14].

2 Previous Work

Numerous works tackled the problem of distinguisher construction between data produced by cryptographic functions and truly random data, both with reduced and full number of rounds. Usually, statistical testing with standard battery of tests or additional custom tailored statistical tests are performed.

In [19], detailed examination of eStream Phase 2 candidates (full and reduced round tests) with STS NIST battery and structural randomness tests was performed, finding six ciphers deviating from expected values. More recently, the same battery, but only a subset of tests, was applied to SHA-3 candidates with a reduced number of rounds as well as only to their compression functions [7].

A method to test statistical properties of short sequences typically obtained by block ciphers or hash algorithms for which some STS NIST tests can not be applied due to insufficient length was proposed in [17]. 256-bit versions of SHA-3 finalists were subjected to statistical tests using a GPU-accelerated evaluation [12]. Because of massive parallelization, superpoly tests introduced by [6] were possible to be performed, detecting some deviations in all but the Grøstl algorithm.

Stochastic algorithms were also applied in cryptography to some extent. A nice review of usage of genetic algorithms in cryptography up to year 2004 can be found in [5], a more recent review is provided by [13]. In [10] a comparison of genetic techniques is presented, with several suggestions which genetic techniques and parameters should be used to obtain better results. TEA algorithm [23] with a reduced number of rounds is a frequent target for cryptanalysis with genetic algorithms. In [4], a successful randomness distinguisher for XTEA limited to 4 rounds is evolved. The distinguisher generates a bit mask with high Hamming weight, which, when applied to function input, results in deviated χ^2 Goodness

of Fit test of the output. However, no distinguisher for full number of rounds was found. Subsequent work [11] improves an earlier attack with quantum-inspired genetic algorithms, finding more efficient distinguishers for a reduced round TEA algorithm succeeding for 5 rounds.

We adopted the genetic programming [2] technique with steady-state replacement. An important difference of our approach is the production of a program (in the form of a software circuit) that provides different results depending on given inputs. Previous work produced a fixed result – e.g., a bit mask in [4, 11] that is directly applied to all inputs.

3 Software Circuits Designed by Evolution

Software circuit is a software representation of a hardware-like circuit with nodes ("gates") responsible for computation of simple functions (e.g., AND, OR). Nodes are positioned in several layers with connectors ("wires") in between. A node may be connected to all nodes from the previous layer, to only some of them, or to none at all. A simple circuit overview can be seen in Fig. 1. Contrary to real single-layer hardware circuits, connectors may also cross each other.

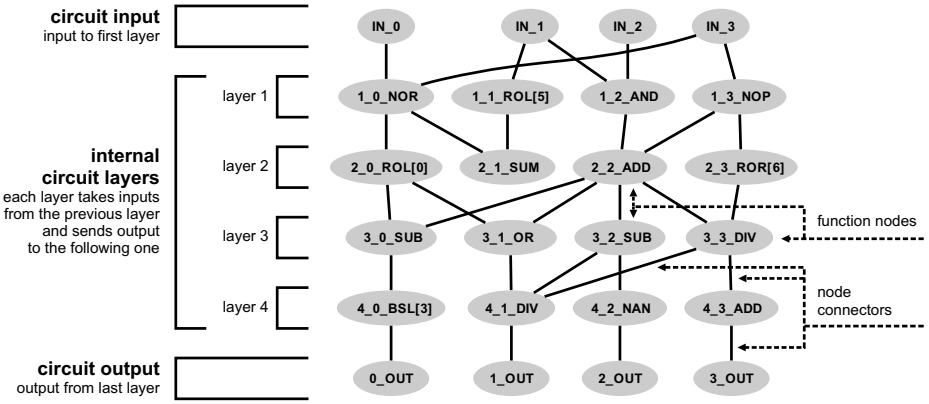

Fig. 1. Simple example of software-emulated circuit.

Usage is versatile – from Boolean circuits where functions computed in nodes are limited to logical operators to artificial neural networks where nodes compute the weighted sum of the inputs. Besides studying complexity problems, these circuits were used in various applications like construction of a fully homomorphic scheme or in design of efficient image filters. Circuit evaluation can be performed by a software emulator or possibly directly in hardware when FPGAs are used.

3.1 The Process of Evolution

The main goal is to find a circuit that will reveal an unwanted defect in the inspected cryptographic function. For example, if a circuit is able to correctly predict the n^{th} bit generated by a stream cipher just by observing previous $(n-1)$ bits, then this circuit serves as a next-bit predictor [24], breaking the security of the given stream cipher. When a circuit is able to distinguish output of the tested function from a truly random sequence, it serves as a random distinguisher [9] providing a warning sign of function weakness. Note that a circuit does not provide correct answers for all inputs – it is sufficient if a correct answer is provided with a probability significantly better than random guessing.

The greatest challenge is the precise circuit design. It can be laid out by an experienced human analyst or created and further optimized automatically. We use the latter approach and combine a software circuit evaluated on a CPU/GPU with evolutionary algorithms. The whole process of circuit design, as also depicted in Fig. 2, is as follows:

1. Several circuits ("candidate solutions") are randomly initialized – both functions in nodes and connectors are chosen at random. Note that such a random circuit will, most probably, not provide any meaningful output for given inputs and can even have disconnected layers.
2. If necessary, new test vectors used for success evaluation are generated.
3. Every individual (circuit) in the population is emulated on all test inputs. The fitness function assigns each circuit a rating based on the obtained outputs (e.g., what fraction of inputs were correctly recognized as being output of a stream cipher rather than a completely random sequence, see Sect. 3.2 for details).
4. Based on the evaluation provided by the fitness function, a potentially improved population is generated from the existing individuals by mutation and sexual crossover. Every individual (circuit) may be changed by altering operations computed in nodes and/or adding/removing connectors between nodes in subsequent layers.
5. The process is repeated from step 2. Usually hundreds of thousands or more repeats are necessary until the desired success rate of distinguisher is achieved.

3.2 The Evaluation of Circuit Success

Evolutionary algorithms need to be supplied with a metric of success – a so-called fitness function. This is used to measure quality of candidate circuits. Proper definition of fitness function is crucial for obtaining a working solution to the defined problem. In this work, we limit ourselves to randomness distinguishability as a target goal. Other goals like next-bit predictor [24] or defector of strict avalanche criterion [22] can be used.

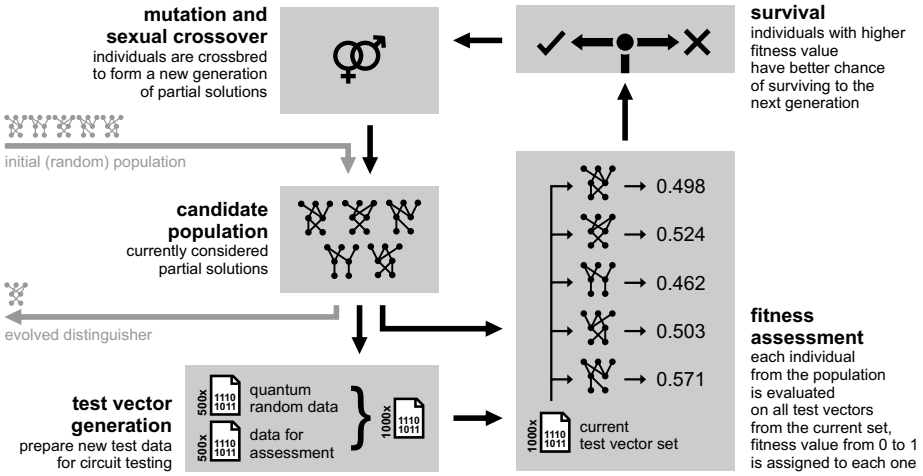

Fig. 2. Simplified work-flow of the evolution process in EACirc.

A circuit input is a sequence of bytes produced either by the inspected function (first type) or generated completely randomly (second type). A circuit output is an encoding of the guessed source. Different encodings are possible: single bit (e.g., 0 meaning "random data" and 1 meaning "function output") or multiple bits (e.g., low versus high Hamming weight of whole output byte). Results in this work use only the byte's highest bit for easier interpretation, but Hamming weight seems to be a better choice for later experiments. Additionally, a circuit can be allowed to make multiple guesses by producing multiple output bytes. A circuit thus has the possibility to express its own certainty in the predicted result (e.g., by setting 2 out of 3 outputs to predict random data and remaining one to predict the function output) as well as to evolve more than one predictor inside a single circuit.

For evaluation of a circuit performance, we use supervised learning with test sets containing pairs of inputs and expected outputs generated prior to the evaluation. Corresponding circuit outputs are compared with expected values to see if the circuit predicted the input source correctly. The success rate (fitness) is then computed as a ratio of correctly predicted test vectors to the total number.

3.3 Evolution and Circuit Parameters

For our experiments, we used the following settings to maintain a good trade-off between the evaluation time (influenced mainly by the number of test vectors) and the ability to prevent over-learning (influenced by the test set change frequency).

– Every test set contains 1 000 test vectors with exactly half taken from inspected function's output and second half taken from random data. Order of

test vectors in the set is not important as test vectors are handled by circuit completely independently.
- Every test vector has the length of 16 bytes.
- Test set is periodically changed every 100^{th} generation to prevent over-learning on a given test set.

Circuit and evolution parameters were fine-tuned empirically based on previous experiments. The settings for the presented experiments are as follows:

- 5 layers, 8 nodes in every internal layer, 16 input nodes (corresponding to 16 input bytes in every test vector) and 2 output nodes.
- Population consists of 20 individuals refreshed by the steady-state replacement strategy with two thirds of individuals replaced every generation.
- 30 000 generations were executed in a single evolution run with 30 separate evolution runs running in parallel.
- Mutation is applied with probability of 0.05 and changes function in a given node or connector mask by addition or removal of connector to a given node.
- Crossover is applied with a probability of 0.5 and performs single point crossover with the first i layers taken from the first parent and remaining layers from the second parent.

Reference experiments were performed using statistical batteries (Dieharder, STS NIST) as a more traditional approach of randomness distinguishing. Note that we will not discuss all results for Dieharder and STS NIST in details as such discussion was already done several times before [7, 19]. We will focus only on the identification of the highest round where some defects are still detected and the significance of such detection – whether almost all tests fail or only a minority of them.

STS NIST was run on 100 sub-streams, each consisting of 1 000 000 bits. *Random Excursions* and *Random Excursions Variant* tests were omitted due to execution problems. All 15 available tests were run in all supported configurations. From the Dieharder suite, only the tests corresponding to the original Diehard collection were used (*Diehard sums test* was omitted due to implementation problems). Each of the chosen tests was run just once, but was allowed to process as much data as it required.

The total volume of data processed by EACirc varies greatly from the volume used by statistical batteries. As can be seen from (1), the results output by EACirc are based on a sample of about 2.3 MB of assessed data.

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

When using STS NIST and Dieharder, we worked with an external file with 250 MB of the assessed stream. The usage of STS NIST amounts to about 11.92 MB while running the whole test set of Dieharder processed about 582 MB altogether with the smallest test consuming about 3 MB and the largest one about 127 MB. Furthermore, we would like to stress out that EACirc makes decisions on much smaller samples of 16 bytes at a time only.

3.4 Implementation Details

We used the following elementary operations for nodes: no operation (NOP), logical functions (AND, OR, XOR, NOR, NAND, NOT), bit manipulating functions (ROTR, ROTL, BITSELECTOR), arithmetic functions (ADD, SUBS, MULT, DIV, SUM), reading a specified input byte even from an internal layer (READX) and producing a constant value (CONST).

As the optimization process requires many evaluations of candidate circuits, we use our computation infrastructure to perform distributed computation with more than a thousand CPU cores. EACirc is implemented with the ability to recover computation based on logs and periodically saved internal state. This provides a possibility to perform evolution with unlimited number of generations even when a computation node itself lasts only a limited time before reboot.

Truly random data used for test vectors were produced by the Quantum Random Bit Generator Service [16] and High Bit Rate Quantum Random Number Generator Service [21].

To ease the analysis of the evolved circuits, we implemented an automatic removal of nodes and connectors not contributing to the resulting fitness value. Furthermore, circuits can be visualized using the Graphviz library.

To independently replicate results provided by a circuit emulator and to double check for possible implementation bugs, EACirc supports exporting circuits into the code of a plain C program. The resulting C program can be compiled separately and computes only the circuit from which it was generated, but completely circumvents the circuit emulator.

4 Application to eStream and SHA-3 Candidates

The testing methodology described in Sect. 3 was applied against several cryptographic functions in order to probe for unwanted properties of their output. We decided to analyze randomness of stream cipher outputs from the recent eStream competition [8] and candidate hash functions from SHA-3 competition [1]. Testing these implementations enabled us to utilize the unified function interface prescribed in the competitions. After this wide-testing, one can cherry-pick only such functions where a well-working circuit is found for further cryptanalysis.

Previous works evaluated statistical properties of candidate functions with the full number of rounds as well as with a reduced number of rounds [19]. Testing full number of rounds usually provides only limited information – either the function is very weak and exhibits weaknesses even in the full number of rounds or no defect at all is detected, even when an serious exploitable attack might exist for a limited number of rounds. In this work, we therefore inspected the functions in reduced-round versions trying to obtain at least the same results as with STS NIST/Dieharder batteries.

4.1 Reference Case

Before performing the experiments themselves, we needed to establish reference values corresponding to random guessing. We therefore let circuits distinguish

between two groups of test vectors, which were both taken from truly random data. Intuitively, our approach should fail to find a working distinguisher and should behave as random guessing.

The predicted behavior was confirmed by an experiment with same settings as those used for testing functions. All statistical tests from Dieharder (20/20) and STS NIST (162/162) successfully passed on this random data, and no working distinguisher was found.

To express the success of evolution, we inspect the fitness of the best individual in the population just after the change of the test set (so as to suppress the influence of over-learning). We compute the average of these maximum fitness values over the whole run. Due to the probabilistic nature of evolutionary algorithms, we repeated every run 30 times and display the average of all runs. All in all, the success of EACirc is expressed by the average fitness value of the best individual in the population right after the change of test set, further averaged across 30 independent runs.

The evolution success for distinguishing two sets of random data, equivalent to random guessing, was 0.52 with independent runs differing in 3^{rd} or 4^{th} decimal place. We anticipated that the difference from the naive value of 0.50 was influenced by population size and the size of test set. As experimentally verified, decreasing the number of individuals in the population or increasing the number of vectors in a test set shifts the evolution success towards the naive value. We can thus conclude that, in our settings, the fitness value of 0.52 corresponds to indistinguishable streams.

4.2 Results for eStream Candidates

From 34 candidates in the eStream competition, 23 were potentially usable for testing (due to renamed or updated versions and 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. We considered three scenarios with respect to the frequency of encryption key change:

1. The key is fixed for all generated test sets and vectors. Even when test sets change, new test vectors are generated using the same key.
2. Every test set was generated using a different key. All test vectors in a particular test set are generated with the same key.
3. Every test vector (16 bytes) was generated using a different key.

Table 1 summarizes results for the selected eStream candidates depending on a number of algorithm rounds and key change frequency. Interpretation of values in table is the following: Dieharder provides three levels of evaluation for a

Table 1. Results for selected eStream candidates with both full and reduced number of internal rounds with respect to the key change frequency.
*During the first 8 rounds, TSC produces no output. This caused 4 Dieharder tests to get stuck, effectively reducing the number of tests to 16.

stream cipher	# 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	Dieharder (x/20)	STS NIST (x/162)	EACirc	Dieharder (x/20)	STS NIST (x/162)	EACirc
Decim	1	0.0	0	0.99	0.0	0	0.85	0.0	5	0.99
	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
FUBUKI	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
Grain	1	0.0	0	1.00	0.0	0	0.67	18.5	162	0.52
	2	0.0	0	1.00	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
Hermes	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
LEX	1	0.0	0	1.00	0.0	0	0.96	3.0	1	1.00
	2	4.0	1	1.00	4.0	1	1.00	3.5	1	1.00
	3	0.5	1	1.00	3.5	1	1.00	4.0	1	1.00
	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
Salsa20	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
TSC	1-8	0.0*	0	1.00	0.0*	0	1.00	0.0*	0	1.00
	9	1.0	1	1.00	1.5	1	1.00	2.0	1	1.00
	10	2.0	13	1.00	3.0	13	1.00	3.0	12	1.00
	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

particular test: pass, weak and fail. Values 1, 0.5 and 0 were assigned to these levels respectively and sum over all tests is computed and displayed. For STS NIST, the number of all passed tests is displayed. This is deduced from the distribution of p-values across all 100 runs with respect to the significance level of $\alpha = 0.01$. The values for EACirc express the average maximum success rate, further averaged through multiple runs (for precise meaning see Sect. 4.1).

Cells representing a stream successfully distinguished from random are denoted by gray background for easier comprehension. Border cases (only a very small deviation found) are shaded in light gray.

The results indicate that, in this case, EACirc performs more or less the same as standard statistical batteries (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 streams than EACirc (for detailed numbers see Sect. 3.3).

4.3 Results for SHA-3 Candidates

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 were again taken from the last successful phase of the competition. 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.
- While the unlimited version produced a random-looking output, their most limited version did not.

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 was cut in half to produce 2 independent test vector inputs of 16 bytes each. In case of generating a continuous stream (for statistical testing), we concatenated the digests.

The results, summarized in Tabs. 2-3, indicate that in this case EACirc performs slightly worse than standard statistical batteries. Although in most of the cases it found 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 Sect. 3.3).

Table 2. Results for selected SHA-3 candidates with both full and reduced number of internal rounds.
*Only 16 Dieharder tests were performed, due to execution problems in some cases.

hash function	# of rounds	Dieharder (x/20)	STS NIST (x/162)	EACirc
ARIRANG	0	0.0	0	1.00
	1	0.0	0	1.00
	2	0.0	0	1.00
	3	0.0	0	1.00
	4	20.0	161	0.52
Aurora	0	0.0	1	0.99
	1	0.0	1	0.75
	2	0.5	132	0.78
	3	0.5	132	0.52
	4	20.0	160	0.52
	17	19.5	161	0.52
Blake	0	0.0	0	1.00
	1	0.0	0	0.52
	2	20.0	162	0.52
	14	20.0	159	0.52
Cheetah	0	0.0	1	1.00
	1	0.0	1	1.00
	2	0.0	0	1.00
	3	0.0	0	0.90
	4	0.0	1	0.86
	5	0.0	1	0.52
	6	20.0	161	0.52
	16	20.0	162	0.52
CubeHash	0	0.0	0	1.00
	1	0.0	0	0.52
	2	20.0	161	0.52
	8	20.0	162	0.52
DCH	0	0.0*	0	1.00
	1	0.0*	0	0.73
	2	19.5	162	0.52
	4	20.0	162	0.52
Dynamic SHA	0	0.0	0	1.00
	1	0.0	0	1.00
	2	0.0	1	0.99
	3	0.0	1	0.95
	4	0.0	18	0.74
Dynamic SHA (continued)	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
Dynamic SHA2	1	1.0	1	0.94
	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
ECHO	1	9.0	24	0.73
	2	9.0	24	0.52
	3	20.0	161	0.52
	8	20.0	161	0.52
Grøstl	0	0.0	0	0.98
	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
Hamsi	0	2.5	1	0.98
	1	2.5	1	0.52
	2	19.5	161	0.52
	3	20.0	162	0.52

Table 3. Results for selected SHA-3 candidates with both full and reduced number of internal rounds.
*Only 16 Dieharder tests were performed, due to execution problems in some cases.

hash function	# of rounds	Dieharder (x/20)	STS NIST (x/162)	EACirc
JH	0	0.0	0	1.00
	1	0.0	0	0.99
	2	0.0	1	0.99
	3	0.0	1	0.99
	4	0.0	1	1.00
	5	0.0	3	1.00
	6	0.0	3	0.98
	7	20.0	161	0.52
	42	20.0	162	0.52
Lesamnta	0	0.0	0	1.00
	1	0.0	0	1.00
	2	0.0	0	1.00
	3	0.0	0	0.52
	4	20.0	162	0.52
	32	20.0	162	0.52
Luffa	0	0.0	0	1.00
	1	0.0	0	1.00
	2	0.0	1	0.99
	3	0.0	1	0.99
	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
MD6	0	0.0*	0	1.00
	1	0.0*	0	1.00
	2	0.0	0	1.00
	3	0.0	0	1.00
	4	0.0	0	1.00
	5	0.0	0	0.98
	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

hash function	# of rounds	Dieharder (x/20)	STS NIST (x/162)	EACirc
SIMD	0	0.0	1	0.99
	1	0.0	1	0.52
	2	19.5	162	0.52
	4	19.5	161	0.52
Tangle	0	0.0	0	1.00
	1	0.0	0	0.99
	2	0.0	1	0.99
	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
Twister	0	0.0	0	1.00
	1	0.0	0	1.00
	2	0.0	0	1.00
	3	0.0	0	1.00
	4	0.0	0	1.00
	5	0.0	0	1.00
	6	0.0	0	1.00
	7	0.0	0	0.52
	8	20.0	161	0.52
	9	20.0	162	0.52

5 Analysis of Evolved Distinguisher

After performing a wide range of experiments, we analyzed one selected case in a more detailed manner. We studied the dependence of distinguisher success rate on the number of generations already computed. Further attention was paid to the evolved circuit and the statistical properties it uses to draw the final verdict (random vs. non-random).

5.1 Achieved Success Rate

The general relationship between fitness value and the number of evolved generations in evolutionary algorithms is very specific – 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. 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). However, even with over-learning, the success rate of distinguishing two sets of random data only rarely exceeded the value of 0.55.

The phenomenon of over-learning can be easily suppressed by changing the test vectors more frequently or increasing the number of vectors in a set. On the other hand, higher test set change frequency or more vectors would increase computational complexity. Therefore a reasonable trade-off is used.

In Fig. 3 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 universal distinguisher, this would be a sufficient evidence for non-randomness of Salsa20 output stream. Also, the circuit is (over-)learning very quickly to a particular data set. Such a behavior led us to inspect the over-learning speed as another potential metric of success instead of how well the circuit is working after a test vector change.

We can further notice that, after initial fluctuations, the circuit success rate shows another periodic behavior about every 4 000 generations. The circuit stabilizes 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 behavior is specific to Salsa20 and its source probably comes from the cipher design. A detailed analysis will be the part of our future work.

5.2 Detailed Distinguisher Inspection

Other type of detailed study of Salsa20 limited to 2 rounds included the evolved distinguishers. 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.

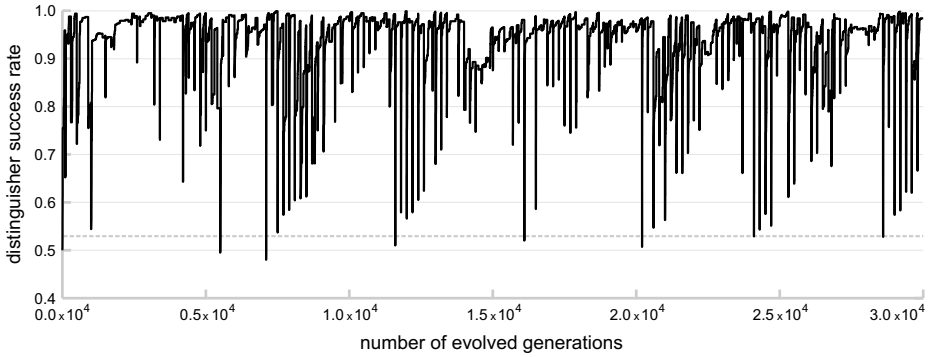

Fig. 3. Circuit success rate for distinguishing Salsa20 limited to 2 rounds from quantum random data (note the shifted scale on y-axis). The dotted line represents the value of 0.52 (stream indistinguishable from random).

Circuits evolved in parallel runs exhibited very similar behavior – in many 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 our 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. More details can be found in [20].

6 Discussion

Based on results obtained with the proposed software circuits designed by genetic programming, a comparison to statistical batteries like STS NIST and Dieharder can be undertaken.

On one hand, the proposed method is based on a completely different approach than statistical tests used in batteries, opening space for detecting dependencies between solutions not covered by tests from batteries. It offers a possibility to construct a distinguisher based on a dynamically constructed algorithm, rather than a predefined one from batteries. Once a working distinguisher is found, it requires extremely short sequences (tested on 16 bytes only) to detect function output. Statistical batteries require at least several megabytes of data. Lower amount of data extracted from a given function is necessary to provide a working distinguisher (at maximum, we used 2.2 MB). Data required by STS NIST and Dieharder were much larger. Note that some tests may provide indication of failure even when less data is available.

On the other hand, subtle statistical defects may not be detected because of very short sequences the circuit is working on. However, several modifications to the proposed approach might enable the processing of larger sequences (e.g., circuit with iterative memory processing data in chunks). Furthermore, the resulting distinguisher may be hard to analyze – what is the weakness detected and what should be fixed in the function design?

The proposed approach requires significantly higher computational requirements during the evolution phase when compared to statistical batteries. However, evaluation of the evolved circuit on additional data is then very fast.

One has to keep in mind that the found distinguisher may be fitted to a particular candidate function (and possibly even a particular key, if the key is not changed periodically in the training set), instead of discovering generic defects in the tested function.

7 Conclusions

We proposed a general design of a cryptanalytical tool based on genetic programming and applied it to the problem of finding a random distinguisher for 25 cryptographic functions taken from eStream and SHA-3 competitions. In general, the proposed approach proved to be capable of closely matching the performance of STS NIST and Dieharder battery. A robust evaluation of various scenarios was performed w.r.t. the key change frequency as well as the number of internal rounds.

The proposed approach provides a novel way of inspecting statistical defects in cryptographic functions and may provide a significant advantage when working with very short sequences once the learning phase of evolution is completed. Our future work will cover techniques that will enable processing significantly more data to provide more fair comparison to STS NIST and Dieharder batteries, as these are making statistical analysis on tens (STS NIST) up to hundreds (Dieharder) of megabytes of data.

Acknowledgments. This work was supported by the GAP202/11/0422 project of the Czech Science Foundation. The access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the program Projects of Large Infrastructure for Research, Development, and Innovations (LM2010005) is highly appreciated.

References

1. SHA-3 competition, announced 2.11.2007 (2007), `http://csrc.nist.gov/groups/ST/hash/sha-3/`
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic programming: An introduction: On the automatic evolution of computer programs and its applications (1997)
3. Brown, R.G.: Dieharder: A random number test suite, version 3.31.1 (2004)
4. Castro, J.C.H., Viñuela, P.I.: New results on the genetic cryptanalysis of TEA and reduced-round versions of XTEA. New Gen. Comput. 23(3), 233–243 (Sep 2005)
5. Delman, B.: Genetic algorithms in cryptography. Ph.D. thesis, Rochester Institute of Technology (2004)

6. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In: Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques. pp. 278–299. EUROCRYPT '09, Springer-Verlag (2009)
7. Doganaksoy, A., Ege, B., Koçak, O., Sulak, F.: Statistical analysis of reduced round compression functions of SHA-3 second round candidates. Tech. rep., Institute of Applied Mathematics, Middle East Technical University, Turkey (2010)
8. ECRYPT: Ecrypt estream competition, announced November 2004 (2004), `http://www.ecrypt.eu.org/stream/`
9. Englund, H., Hell, M., Johansson, T.: A note on distinguishing attacks. In: Information Theory for Wireless Networks, 2007 IEEE Information Theory Workshop on. pp. 1–4. IEEE (2007)
10. Garrett, A., Hamilton, J., Dozier, G.: A comparison of genetic algorithm techniques for the cryptanalysis of tea. International journal of intelligent control and systems 12(4), 325–330 (2007)
11. Hu, W.: Cryptanalysis of TEA using quantum-inspired genetic algorithms. Journal of Software Engineering and Applications 3(1), 50–57 (2010)
12. Kaminsky, A.: GPU parallel statistical and cube test analysis of the SHA-3 finalist candidate hash functions. In: 15th SIAM Conference on Parallel Processing for Scientific Computing (PP12). SIAM (2012)
13. Picek, S., Golub, M.: On evolutionary computation methods in cryptography. In: MIPRO, 2011 Proceedings of the 34th International Convention. pp. 1496 –1501 (2011)
14. EACirc project: https://github.com/petrs/eacirc (2013)
15. Rukhin, A.: A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications, version STS-2.1. NIST Special Publication 800-22rev1a (2010)
16. Stevanović, R., Topić, G., Skala, K., Stipčević, M., Rogina, B.M.: Quantum random bit generator service for Monte Carlo and other stochastic simulations. In: Lirkov, I., Margenov, S., Waśniewski, J. (eds.) Large-Scale Scientific Computing, pp. 508–515. Springer-Verlag (2008)
17. Sulak, F., Doğanaksoy, A., Ege, B., Koçak, O.: Evaluation of randomness test results for short sequences. In: Proceedings of the 6th international conference on Sequences and their applications. pp. 309–319. SETA'10, Springer-Verlag (2010)
18. Svenda, P., Ukrop, M., Matyas, V.: Towards cryptographic function distinguishers with evolutionary circuits. In: SECRYPT. pp. 135–146 (2013)
19. Turan, M.S., Doğanaksoy, A., Ç. Çalik: Detailed statistical analysis of synchronous stream ciphers. In: ECRYPT Workshop on the State of the Art of Stream Ciphers (SASC'06) (2006)
20. Ukrop, M.: Usage of evolvable circuit for statistical testing of randomness. Bachelor thesis, Masaryk university (2013), `https://is.muni.cz/th/374297/fi_b/thesis.pdf`
21. EQRNG Service, H.u.: http://qrng.physik.hu-berlin.de/ (2014)
22. Webster, A.F., Tavares, S.E.: On the design of S-boxes. pp. 523–534. Springer-Verlag (1986)
23. Wheeler, D., Needham, R.: TEA, a tiny encryption algorithm. In: Fast Software Encryption. pp. 363–366. Springer (1995)
24. Yao, A.C.: Theory and application of trapdoor functions. In: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science. pp. 80–91. SFCS '82, IEEE Computer Society, Washington, DC, USA (1982)