On the origin of yet another channel

Petr Švenda and Václav Matyáš
{svenda, matyas}@fi.muni.cz

Masaryk University, Brno, Czech Republic

Abstract. Cryptanalysis of a cryptographic function like stream, block or hash function usually requires human cryptanalytical skills and labour. However, some automation is possible – e.g., by randomness testing suites like NIST/Diehard that can be applied to test statistical properties of cryptographic function outputs. Yet such testing suites are limited only to predefined statistical functions. We propose more open approach based on combination of software circuits and evolutionary algorithms to search for unwanted statistical properties like next bit predictability or random data non-distinguishability. Design of a software circuit acting as a testing function is automatically evolved by a stochastic optimization algorithm and uses the potentially unknown "other channel" leaking information during cryptographic function evaluation.
We tested this approach on candidate algorithms for SHA-3 and eStream competitions with comparable (but slightly worse) results as STS NIST and Diehard tests w.r.t. the number of rounds of the inspected algorithm, where tests are still able to detect unwanted statistical properties in output. Additionally, the proposed approach is not limited only to assess randomness-like properties in function output, but can be also used for other tests like whether a function is invertible or how does its avalanche effect degrade.

1 Unguided hunt for weaknesses in cryptographic functions

The main motivation for this work is to provide a tool with the crucial ability to automatically probe for unwanted properties of cryptographic functions that signalize flaws in the function design. Such properties might be (note that we intentionally target a broad range of cryptographic functions):

- predictability of next output bit (stream ciphers),
- corrupted avalanche effect (hash functions, stream ciphers),
- distinguishability of function outputs from truly random data (block ciphers), etc.

Typical cryptanalytical approach against new cryptographic function is usually based on application of various statistical testing tools (STS NIST, Diehard) as the first step and then application of established cryptanalytical procedures (algorithmic attacks, differential cryptanalysis) combined with an in-depth knowledge of the inspected function. The first step can be at least partly automated

and (relatively) easy to apply, but will detect only most visible defects in function construction or apply only to limited number of algorithm rounds. The second approach usually yields much stronger insight and detected more defects, but usually requires extensive human cryptanalytical labour. Additionally, general 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 does not pass a certain test, yet on the other hand severely limits the potential to detect other defects.

We designed and tested an automated process that can be used in a similar manner as general statistical testing suites, but additionally provides the possibility to construct (again automatically) new tests. We represent "tests" as a hardware-like circuit with a software emulator to execute the circuit over given inputs and to compute outputs and evolutionary algorithms (EAs) to design the circuit layout ("wires" and "gates"). Although such an automated tool will not (at least for the moment) outperform a skilled cryptographer on particular cryptographic function, it still has two main advantages:

- It can be applied automatically against multiple different cryptographic functions with no additional human labour working implementation of the inspected function is sufficient.
- It may discover and use unanticipated information leakage "channels" from the function than those usually assumed by cryptographers.

We implemented the tool (more details given in Section 3) and tested our idea on SHA-3/e-Stream candidate functions (details are given in Section 4). Results are very similar to those obtained from NIST and Diehard test suites w.r.t. the number of rounds of the inspected function where tests were able to find some defects. Based on experience with behaviour and significance of results, we add detailed discussion about potential extensions, expressive power of an circuit and interesting behaviour detected (Section 5). Conclusions are given in Section 6.

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 battery of tests (e.g., STS NIST [Ruk10] or Dieharder [Bro04]) or additional custom tailored statistical tests are performed. The STS NIST battery was used to evaluate fifteen AES (round 2) candidates, demonstrating some deviation from randomness in six candidates [Sot99]. In [TDcc06], 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 the tests, was applied to the SHA-3 candidates (in the second round of competition, 14 in total) for a reduced number of rounds as well as only to compression function of algorithm

[DEKS10]. Additionally, custom-built statistical tests based on strict avalanche criterion and others were used, resulting in estimation of relative security margins of candidates w.r.t. the number of rounds. [SDEK10] proposed a method to test statistical properties of short sequences typically obtained by block ciphers or hash algorithms for which some from STS NIST can not be applied due to insufficient length. Probabilities expressed by p-values are calculated for each short subinterval and improved method based on recalculation of expected probabilities is provided. Example results applied to selected block and hash functions are presented. 256-bit versions of SHA-3 finalists were subjected to statistical tests using a GPU-accelerated evaluation [Kam12]. Both algorithms and selected tests from STS NIST battery were implemented for the nVidia CUDA platform. Because of massive parallelization, superpoly tests introduced by [DS09] were possible to be performed, detecting some deviations in all but the Grøstl algorithm.

Stochastic algorithms were also applied in cryptography to some extent, focusing initially mostly on simple transposition and substitution ciphers or problems like efficient knapsack algorithm. A nice review of usage of genetic algorithms in cryptography up to year 2004 can be found in [Del04], a more recent review of evolutionary methods used in cryptography is provided by [PG11]. TEA algorithm [WN95] with a reduced number of rounds is a frequent target for cryptoanalysis with genetic algorithms [CVn05,Hu10,GHD07]. In [GHD07] a comparison of genetic techniques is presented, with several suggestions which genetic techniques and parameters should be used to obtain better results. We adopted the genetic programming [BNKF97] technique with steady-state replacement [LLL08]. An important difference of our approach from previous work 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., bit mask in [CVn05,Hu10] that is directly applied to all inputs.

Structure of a software circuit resembles artificial neural networks (NN) to some extent. Notable differences are in the learning mechanism and in a high number of layers used in our software circuit (NN usually use only three). Recently, deep belief neural networks (DBNN) were proposed [HOT06] with the learning algorithm based on restricted Boltzmann machines that also use 5 or even more layers. Still, a software circuit uses mutation and crossover to converge towards an optimum instead of back propagation in case of classical NN or lay-by-layer learning algorithm for DBNN. Also, different functions may be computed inside every node in case of software circuit instead of weighted sum of DBNN.

3 Software circuit designed by evolution

Software circuit is a software representation of a hardware-like circuit with nodes ("gates") responsible for computation of simple functions like AND or OR taking inputs and providing outputs. Nodes are positioned in layers where outputs from the previous layer are provided as inputs to the nodes in the following layer by

connectors ("wires"). Input to the whole circuit is simulated as an output of the first layer of nodes and output of last layer is taken as the output of whole circuit. Connectors might connect node to all nodes from a previous layer or only to some of them.

Examples of such a circuit might be a Boolean circuit where functions computed in nodes are limited to logical functions or 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 [Gen10] or in design of efficient image filters [SSV12].

3.1 How to design circuit layout

Circuit evaluation can be performed by a software emulator that propagates input values, computes functions in nodes and collects outputs or possibly directly in hardware when FPGAs are used [SSV12]. Circuit design can be laid out by an experienced human designer, automatically synthetized from the source code or even automatically designed and then improved by an optimization algorithm. We use the last approach and combine a software circuit evaluated on a CPU (or also on GPUs) with evolutionary algorithms (EAs). The main goal is (somehow) to find a circuit that will reveal an unwanted defect in the inspected cryptographic functions. For example, if a circuit is able to correctly predict the n^{th} bit from a key stream generated by a stream cipher just by observing previous (n-1) bits, then this circuit serves as a next-bit predictor [Yao82], breaking the security of the given stream cipher. Note that a circuit need not to provide correct answers for all inputs – it is sufficient if a correct answer is provided with a statistically significant probability better than random guessing.

When combined with evolutionary algorithms (broader term than genetic algorithms, covers also stochastic algorithms inspired by nature evolution), the whole process of circuit design consists from the following steps:

1. Several software circuits are randomly initialized (randomly selected functions in nodes, randomly assigned existence of connectors between nodes) forming population of candidate individuals. Every individual is represented by one circuit. Note that such a random circuit will most probably not provide any meaningful output for given inputs and can even have disconnected layers (no output at all).
2. If necessary, generate new test vectors used later by a so-called fitness function for evaluation.
3. Every individual (circuit) in the population is emulated and obtained outputs are evaluated by the fitness function that will assign a rating based on how well does this individual perform in solving a given task (e.g., what fraction of inputs were correctly recognized as being output of stream cipher rather than completely random sequence).
4. Based on the evaluation provided by the fitness function, a potentially improved population is generated by mutation and crossover operators (genetic algorithms) from individuals taken from the previous generation. Design of

every individual (circuit) may be changed by changing operations computed in nodes or add/remove connectors between nodes in subsequent layers.

5. Repeat from step 2. Usually hundreds of thousands or more repeats are performed, therefore the evaluation of a single circuit in step 3 must be fast enough (currently, we are in the milliseconds range).

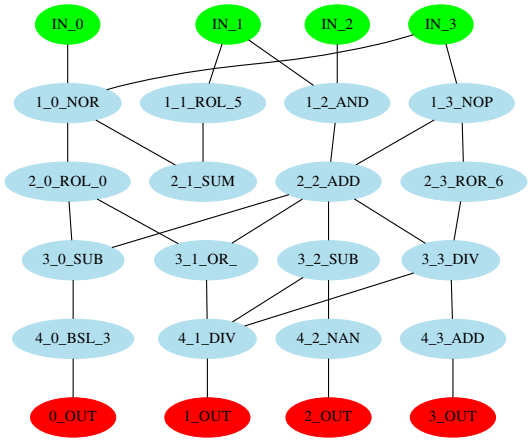

Fig. 1: Software circuit with input nodes (IN_x), inner nodes, output nodes (x_OUT) and connectors. Note that not all input or output nodes need to be used as well not all inner nodes need to output any value.

3.2 How to evaluate circuit performance?

Evaluation of a circuit performance is a crucial yet tricky part of the proposed process. Evaluation of a circuit is so called "supervised teaching" – we have pairs of inputs and expected outputs (given by a "teacher"). Outputs from a circuit for given inputs are compared with expected outputs and circuit performance is then graded accordingly. When done incorrectly, the process will not provide a circuit solving the expected problem. The progress may fail at least on two fronts:

1. Improperly defined problem to be solved by circuit. For example, if we define a problem to be solved so that there is more than one correct answer (e.g., to find a preimage for a given hash function output) and yet we insist only on one particular preimage being correct (although other values also provide same hash), the circuit will not be able to converge towards a working solution, even though the hash function is invertible. Alternatively, a problem may be too hard to be solved, yet a working solution for a limited number of rounds still would be of interest from the cryptanalysis perspective. Finally, a circuit may seem to be solving the problem, yet we do not learn anything

about the function itself – so called "overlearning"). E.g., if we ask the circuit to distinguish between a function output and a completely random value, but we do not change the test vectors, so we just learn which particular test inputs belongs to which category and achieve a very good performance on the testing set, but not on new verification data.
2. Unsuitable settings for EAs to progress towards a better solution (usually caused by an improper fitness function or insufficient amount of computational time). EAs work well where a gradual improvement with small steps towards a better solution is possible. Problems for which you either have a solution that solves it at once or you have nothing are not suitable. For example, defining fitness function as binary YES (all output bits of circuit match expected bits) or NO (otherwise) will hamper EA chances to find a working solution. A better approach is to calculate fraction of bits that match over many different tests vectors (input, expected output). Last but not least, changing the test vectors either too often (EA fails to adapt) or too infrequently (EA will overlearn) can lead to dead ends.

So far, we adapted the following problems to be tested by a circuit, with results presented for the first one (random distinguisher) in Section 4:

1. Random distinguisher – the circuit input is a sequence of bytes produced either by the inspected function or generated completely randomly and the output is the guessed source (e.g., if the Hamming weight of circuit output is higher than $\frac{1}{2}$ than it is the function output, otherwise it is a random sequence). A circuit is successful if able to distinguish function outputs from random sequences significantly better than by random guessing. Truly random sequences were taken from the Quantum random bit generator service [STS+08].
2. Next bit predictor – the circuit input is a vector of n bits taken from an output of the function (e.g., stream cipher) and the expected circuit output is the value of the $(n+1)th$ bit. The problem can be relaxed to the prediction of multiple bits, Hamming weight or other property of following byte(s). A circuit is successful if able to predict correct value(s) significantly better than by random guessing. A typical target would be some keyed function with an unknown key, yet unkeyed functions can be targeted as well – caution must be taken to prevent a circuit simply learning the unkeyed function itself and use it to compute the expected output.
3. Strict Avalanche Criterion defector – the avalanche effect property of function F (e.g., hash) expects half bits to flip in the output on average, even for a single bit change in the input. The input for the circuit is a sequence of bytes X. Expected output from the circuit is such a sequence Y that will produce significantly more (or less) bit flips than expected when processed by a function (Hamming_distance$(F(X), F(Y)) >> |X|/2$). A special care must be taken not to let circuit just learn the function itself.

Note that an interpretation of a circuit output might not be just an exact match to the expected value. Even when only Yes or No is expected, one can let

the circuit to encode its answer into a bit value/Hamming weight/majority value of the circuit output, as such a less strict matching allows for multiple ways how the circuit can signalize Yes answer and gives more flexibility to the EA.

3.3 Practical implementation

We implemented our evaluation software circuit both on CPU and GPU combined with the GAlib optimization library [GL207]. Significantly larger test vectors are possible (10^5 instead of 10^3) with a GPU implementation (nVidia CUDA), where many different evaluations can be executed in parallel with negligible performance impact. On average, a 70x speedup w.r.t. CPU implementation was achieved. Additionally, we use BOINC infrastructure to perform distributed computation with more thousand CPU cores and 16 nVidia GF 465 cards. One of well described problem of neural networks is difficulty to understand the resulting solution. To ease understanding of our software circuits, we implemented an automatic removal of nodes and connectors that do not contribute to the resulting fitness value, transformation into the Graphviz dot format for visualization and also transformation into a C program source code that executes the functionality of a particular circuit without the need to run circuit emulator. For start, we used following elementary operations for nodes: no operation (NOP), logical functions (AND, OR, XOR, NOR, NAND), bit manipulating functions (ROTR, ROTL, BITSELECTOR), arithmetic functions (ADD, SUBS, MULT, DIV, SUM), read specified input even from internal layer (READ) and produce constant value (CONST). Typical initial settings for software circuit parameters were: 5-8 layers, 16-32 input nodes, 16-32 nodes in internal layer, 1-16 output nodes. Typical settings for EA parameters were like: 20 individual in population, 1000 test vectors (CPU version), 0.05 probability of mutation, and 0.5 probability of crossover.

4 Results for eStream/SHA3 candidates

The approach described above was tested on candidate function for eStream [ECR04] and SHA-3 [SHA07] competitions. The big advantage came with availability of implementations with the same programming interface (API) for all candidates – one can automatically test (e.g., random distinguisher) on large number of functions without need to change (significantly) corresponding code.

Presented results serve as a baseline over multiple functions rather than the best result we can achieve against particular function. Because of the significant number of inspected functions parametrized additionally with different number of rounds and multiple parallel runs for every such a combination, we were not able to optimize for best results for every separate function. Indeed, we were able to obtain improvements (better distinguishing rate) for selected functions where we selectively applied more optimizations. For example, when circuit with memory (see Section 5) was used against Decim limited to three rounds, distinguishing success rate raised from 0.53 to 0.62.

4.1 eStream candidates

From 34 candidates in the eStream competition, 23 were potentially usable for testing (renamed or updated versions, problems with compilation). For start, we limited ourselves to only 7 of these (Decim, Grain, FUBUKI, Hermes, LEX, Salsa20, TSC) having structure allowing reduction of complexity by decreased number of rounds in a straightforward way.

In this work, we aim to obtain a software circuit capable of correctly distinguishing between a stream of bytes generated by a eStream candidate function with an unknown key and stream of truly random bytes. We worked with three scenarios with respect to the frequency of key change:

1. 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.

Function name (total rounds)	Rounds detectable by NIST (run/set/vector)	Rounds detectable by circuit (run/set/vector)
Decim (8)	5/5/2	3/3/1
FUBUKI (4)	0/0/0	0/0/0
Grain (13)	2/2/0	2/2/0
Hermes (10)	0/0/0	0/0/0
LEX (10)	3/3/3	3/3/3
Salsa20 (20)	2/2/0	2/2/0
TSC (32)	10/10/10	10/10/10

Table 1: Results for eStream candidates.

4.2 SHA-3 candidates

Similarly, we tested also SHA-3 competition candidates. From 51 candidates for the first round, only 42 were potentially usable for testing due to compilation problems, source code size, speed etc. We limited ourselves to 18 candidates that can be easily 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. Following candidates were considered: ARIRANG, Aurora, Blake, Cheetah, CubeHash, DCH, Dynamic SHA, Dynamic SHA2, ECHO, Grøstl, Hamsi, JH, Lesamnta, Luffa, MD6, SIMD, Tangle and Twister.

Function name (total rounds)	ARIRANG (4)	Aurora (17)	Blake (14)	Cheetah (16)	CubeHash (8)	DCH (4)	Dynamic SHA (16)	Dynamic SHA2 (17)	ECHO (8)	Grøstl (10)	Hamsi (3)	JH (42)	Lesamnta (32)	Luffa (8)	MD6 (104)	SIMD (4)	Tangle (80)	Twister (9)
Rounds detectable by STS NIST	3	3	1	5	1	1	7	12	2	3	1	6	3	7	9	1	22	7
Rounds detectable by software circuit	3	2	0	4	0	1	7	10	1	2	0	6	2	7	8	0	22	6

Table 2: Results for SHA-3 candidates.

4.3 Discussion

Detailed results including the circuits found, analysis of some of them and details for the STS NIST settings can be found in this paper supplementary data[1] and [Ukr13]. The following main points were observed:

1. The circuit providing good distinguishing results for the particular combination of function, number of rounds and used key usually significantly decrease in success rate when function or even key is changed. Therefore, whole process of the evolution to particular combination is to be perceived as a test in some cases.
2. Not all operations and connectors are relevant for the circuit performance. Irrelevant components can be automatically pruned out, easing understanding of the circuit.
3. STS NIST is better in detection of statistical deviances than proposed approach for multiple SHA-3 candidates, but usually only one more round.

5 Increasing the expressivness of the circuit

Previous section provided us with the baseline results for the wide range of functions. We layed out the following metrics to measure the success of the proposed approach:

1. Proposed approach should have the power to express statistical tests used in STS NIST battery (i.e., test from NIST can be encoded in the form of software circuit, but not necessarily automatically by genetic programming).
2. Proposed approach should provide at least same results (w.r.t. number of rounds of limited function) as the STS NIST battery. Preferably, there should

[1] Detailed results can be found at `http://www.fi.muni.cz/~xsvenda/papers/spw2013/`.

be at least one (cryptographic) function with the number of rounds limited to N, where STS NIST fails to detect significant defects in sequence, but proposed approach does.

3. Proposed approach may fail to achive same results as the STS NIST, but then should provide other significant advantage like smaller computation or memory requirements or requiring significantly less input data (thus allowing for use as function's online tester).
4. Provide better distinguishing success rate when combined with STS NIST then STS NIST provides alone (approach is providing additional test coverage not provided by STS NIST, even when approach alone cannot distinguish with better probability than STS NIST).

Generally, we achieved points 1 (when extension with memory is considered, see Section 5) and 3, in some cases achieved point 2 and failed yet to achieve point 4. Basic version of the proposed approach uses inputs only 16-32 bytes long. Note that constructing distinguisher between function output and truly random data based on as short sequence as 16 bytes is significantly more difficult than same task for long sequence. At first, longer sequence decrease the impact of small fluctuation of both random data and function's output (e.g., if there is inbalance of number of zeroes and ones by only one bit in 16 bytes, it is 1/128 of whole input, making it relatively significant difference, even when one can completelly expect such a situation in truly random data. Contrary, if 1 000x1 000 000 bits are used (STS NIST), such a difference is insignificant). Second, function's output can produce periodic behavior easily detectable in longer data stream, but completely invisible in 16 bytes only. Indeed, when data for STS NIST battery is generated in such a way that change key every 16 bytes, proposed approach provides exactly same results on the tested functions.

To further improve performance of the evolutionary circuit, we propose additional extensions of the basic version presented before. Note that as modification of a circuit is based evolutionary algorithms, performance may be increased by application of generic techniques for faster convergence towards the optimal solution like modification of EA parameters. We will not cover these techniques here. Note that increasing the circuit expressiveness may actually decrease the convergence speed for easier problems as search space is usually increased by such a technique.

We can divide such techniques into several groups:

- Techniques to increase amount of data processed by the circuit: READX, memory (block-based, bit-based), circuit providing formula with expected occurrence counter
- Techniques to increase expressivity of circuit: loops, circuit of circuits
- Techniques to increase complexity of functions used by the circuit: linear genetic programming inside node, code fragments inside node, complex instruction from known tests

5.1 Techniques to increase an amount of data processed by the circuit

Previous results were provided for the situation where circuit perform distinguisher was based on 16 input bytes only – significant disadvantage w.r.t statistical batteries processing from tens up to hundreds of megabytes of data. With such a setting, it is significantly harder situation for the evolutionary circuit to find a working distinguisher. Therefore, we propose several techniques how circuit expressivity and amount of data used can be expanded.

Possibly easiest way how to provide more data to circuit is to introduce additional instruction (called READX) which provides one of the input byte from the circuit's inputs directly to the function node with READX instruction. One may perceive such an instruction as a direct wire between node and requested input. Such an input can be already encoded by the circuit, but for the price of several NOP instructions. With a special instruction, input is directly accessible and more importantly, length of circuit's input can be more than the size of input layer as READX can obtain any value from it. Human interpretation of circuit with READX instruction is straightforward. However, if the input data should be enlarged by e.g., one hundred input bytes, one hundred READX instructions must be present in the circuit to process all inputs by the circuit (but not all inputs bytes are relevant for a distinguisher).

More promising, but also more complicated (also from the perspective of the interpretation of well performing circuit) is introduction of a *memory*. Instead of circuit processing whole input in a single run, input is divided into B blocks with same length N and processed one by one. Circuit is extended by additional M inputs and M outputs (same number). Outputs from a processing block B_i are provided as inputs for the block B_{i+1}, together with N inputs from test vector. Such an extension provides possibility to extract some statistics from input block B_i, store it into the memory M and later combine with statistics from following blocks. Most importantly, test vector length can be significantly increased and all input bytes are directly processed by the circuit. Final output (distinguisher) is then based on memory and last input block. We already obtained preliminary results for such a modification which shows better results (better distinguishing ratio) than single-run circuit. Note that longer input test vector also increases circuit's execution time accordingly.

Finally, one can incorporate circuit into bigger framework which will perform part of the computation and evaluation separately. If evolutionary circuit is providing working distinguisher, this distinguisher must be based on some redundancy in input stream of a tested function which is not present in truly random data. Such a redundance is expressed (if found) by some formula encoded inside the circuit. Circuit's output is then interpreted not directly as distinguishing verdict (truly random data or function data), but only as classificator of input data into one of several categories $C_1, ..., C_n$ (e.g., if circuit provide one output byte, then classifies into 256 categories). If multiple inputs are given to a circuit and classified, particular distribution over categories C_i is obtained. The goal is to find such a circuit, which will provide significantly different distribution D_f

of categories C_i for inputs coming from the tested function f then a distribution D_r produced by the inputs taken from truly random data. If such a circuit can be found, input stream of data (multiple inputs) are signalized as function's output if significant deviation from pre-computed distribution D_r is detected in D_f and signalized as truly random data otherwise (with given confidence level). Fitness function is again defined as a ratio of correctly classified test vectors from presented test set. Note that test vectors now need to be extended from single input block to multiple input blocks take from same source (function or random data) – single input block would provide only very crude distribution D. Also, much longer input data are naturally provided to circuit to facilitate decision.

5.2 Techniques to increase inner functions complexity

Set of operations available to software circuit design can be extended from elementary operations like Boolean functions or simple arithmetic operations to code fragments automatically extracted by parser from implementation of inspected function with hope of achieving better results than NIST/Diehard batteries. Previously described circuit constructions used very simple operations like AND, MULT or SUM in the circuit processing nodes. When a circuit needs some more complex operation, it needs to be constructed from these simple blocks, occupying multiple layers and connectors. Potentially, a better performance in weakness hunting can be obtained if the set of allowed operations are extended by the more complex ones. As a particular cryptographic function is inspected, sub-operations of this function might be viable candidates for such a complex operations. However, selection of such an operation requires some knowledge of the function itself, hampering advantages of fully automated approach.

We propose to keep with the fully automated approach and let sub-operations be extracted from an existing implementation of the inspected function automatically. Once extracted, evolution algorithm is allowed to select these fragments as the function for processing nodes and emulate these fragments on inputs if selected. Note that by partial execution, we do not aim to replicate exact behaviour of target code (same output for same input), but rather to provide similar code that can be applied over any input data provided by rest of circuit.

For practical verification of this idea, we choose implementation of target function (e.g., stream cipher) in the Java language with advantage of human-readable bytecode generated directly by Java compiler and disassembler (javac g, javap -c). Resulting file with text representation of bytecode is automatically parsed and any subpart of code described by triple *[method_name, start_instruction_offset and end_instruction_offset]* can be emulated by simple stack-based execution machine.

Several challenges need to be tackled with such a partial execution:

– Handling of method inputs – in regular program execution, method arguments are pushed to stack before method call. If method is to be executed from the middle, stack with arguments has to be filled by other means. In

our implementation, part of values provided by previous layer via connectors is stored into stack before partial execution is performed.
- Bottom of the stack is reached before end of execution – because not all instructions in method are executed (e.g., push), bottom of the stack might be reached prematurely as the current instruction to be executed expects value(s) on stack whereas there is (are) none. In our implementation, we simply ignore the instruction in case of not enough arguments are present on stack.
- Handling of global arrays – in case when global arrays are used to load and store values during execution, such memory structures must be prepared and set before first access. In our implementation, part of values provided by previous layer via connectors is used. Also, instruction is ignored when data necessary for emulation are not present.
- Form of the output from a partial execution emulated by given node. More specifically, which byte (or multiple bytes) should be provided as nodes output? In our implementation, we simply take value at the top of the stack as output.

Note that challenges described above are solved in ad-hoc manner and may fail to execute many instructions from original code. Still, if target function itself uses e.g., finite field multiplication (FFMul in AES) partial execution described above will enable software circuit to directly execute such an operation.

Once execution stack described above is available, fragments can be taken not only from existing code, but can be also generated randomly. Every fragment in the node will then consist from several bytecode instruction emulated over node's input – technique known as linear genetic programming.

Another options are to use parsers like the ANTLR parser generator [PQ94] or ASTParser [KT06], alternatively BytecodeParser [God12] working directly on the bytecode level, thus easing emulation later. Optionally, one may use language supporting reflection and allowing for runtime code modification. However, this will decrease the evaluation speed of a single circuit (w.r.t. C/C++ performance) and prevent a GPU-based acceleration. Note that a large number of candidate circuits needs to be evaluated; otherwise EAs are unlikely to find a viable solution.

6 Conclusions

We proposed a general design of a cryptoanalytical tool based on genetic programming and applied it to the problem of finding a random distinguisher for several stream ciphers (with a reduced number of rounds) taken from the SHA-3 (18 functions) and eStream (7 functions) competition. Baseline with results was established for these functions. In general, the proposed approach proved to be capable of matching the performance of the NIST statistical testing suite in scenario, where the tested function key is changed for every test vector and close matching results when key is changed less often. When a key is changed less often, longer sequence with the same key is produced and available for inspection

by statistical testing suite where a basic version of the proposed approach is not able to deal with inputs longer than tens of bytes. Therefore, we proposed several extensions like circuit with memory or more complex function in the node capable to process very large inputs.

Future work will be devoted to evaluation of extension proposals and their comparison with basic version of proposed approach.

Acknowledgements: 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 programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005) is highly appreciated. Martin Ukrop provided data from the experiments with SHA-3 candidate functions evaluations.

References

[BNKF97] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. Genetic programming: An introduction: On the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers, 1997.

[Bro04] Robert G. Brown. Dieharder: A random number test suite, version 3.31.1. 2004.

[CVn05] Julio César Hernández Castro and Pedro Isasi Viñuela. New results on the genetic cryptanalysis of TEA and reduced-round versions of XTEA. *New Gen. Comput.*, 23(3):233–243, September 2005.

[DEKS10] Ali Doganaksoy, Barıs Ege, Onur Koçak, and Fatih Sulak. Statistical analysis of reduced round compression functions of SHA-3 second round candidates. Technical report, Institute of Applied Mathematics, Middle East Technical University, Turkey, 2010.

[Del04] Bethany Delman. *Genetic algorithms in cryptography.* PhD thesis, Rochester Institute of Technology, 2004.

[DS09] Itai Dinur and Adi Shamir. 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*, EUROCRYPT '09, pages 278–299. Springer-Verlag, 2009.

[ECR04] ECRYPT. Ecrypt estream competition, announced November 2004. 2004.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, March 2010.

[GHD07] Aaron Garrett, John Hamilton, and Gerry Dozier. A comparison of genetic algorithm techniques for the cryptanalysis of TEA. *International journal of intelligent control and systems*, 12(4):325–330, 2007.

[GL207] Galib 2.4.7, a c++ library of genetic algorithm components. 2007.

[God12] Stephane Godbillon. Bytecodeparser - java bytecode parser and emulator. 2012.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[Hu10] Wei Hu. Cryptanalysis of TEA using quantum-inspired genetic algorithms. *Journal of Software Engineering and Applications*, 3(1):50–57, 2010.

[Kam12] A. Kaminsky. 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)*, 2012.

[KT06] Kuhn Kuhn and Olivier Thomann. Eclipse ASTParser. 2006.

[LLL08] Liu Liu, Minqiang Li, and Dan Lin. Replacement strategies in steady-state multi-objective evolutionary algorithm: A comparative case study. In *Proceedings of the 2008 Fourth International Conference on Natural Computation*, ICNC '08, pages 645–649, Washington, DC, USA, 2008. IEEE Computer Society.

[PG11] S. Picek and M. Golub. On evolutionary computation methods in cryptography. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 1496 –1501, 2011.

[PQ94] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[Ruk10] A. Rukhin. 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.

[SDEK10] Fatih Sulak, Ali Doğanaksoy, Bariş Ege, and Onur Koçak. Evaluation of randomness test results for short sequences. In *Proceedings of the 6th international conference on Sequences and their applications*, SETA'10, pages 309–319. Springer-Verlag, 2010.

[SHA07] NIST SHA-3. SHA-3 competition, announced 2.11.2007. 2007.

[Sot99] J. Soto. Randomness testing of the AES candidate algorithms. NIST, 1999.

[SSV12] Lukáš Sekanina, Vojtech Salajka, and Zdenek Vašíček. Two-step evolution of polymorphic circuits for image multi-filtering. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[STS+08] Radomir Stevanović, Goran Topić, Karolj Skala, Mario Stipčević, and Branka Medved Rogina. Quantum random bit generator service for Monte Carlo and other stochastic simulations. In Ivan Lirkov, Svetozar Margenov, and Jerzy Waśniewski, editors, *Large-Scale Scientific Computing*, pages 508–515. Springer-Verlag, 2008.

[TDcc06] M. S. Turan, A. Doğanaksoy, and Ç. Çalik. Detailed statistical analysis of synchronous stream ciphers. In *ECRYPT Workshop on the State of the Art of Stream Ciphers (SASC'06)*, 2006.

[Ukr13] Martin Ukrop. Usage of evolvable circuit for statistical testing of randomness. uppercaseBachelor thesis, Masaryk University, Czech Republic, 2013.

[WN95] David Wheeler and Roger Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1995.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 80–91, Washington, DC, USA, 1982. IEEE Computer Society.