The Efficient Randomness Testing using Boolean Functions

Marek Sýs, Dušan Klinec, Petr Švenda
Masaryk University, Botanicka 68a, Brno, Czech Republic
{syso, ph4r05}@mail.muni.cz, svenda@fi.muni.cz

Keywords: statistical randomness testing; hypothesis testing, boolean function

Abstract: The wide range of security applications requires data either truly random or indistinguishable from random. The statistical tests included in batteries such as NIST STS or Dieharder are frequently used to assess the randomness property. We designed a principally simple, yet powerful, statistical randomness test working on bit level. It is based on a search for boolean function(s) indicating a bias when applied to the tested stream not expected for truly random data. The deviances are detected in seconds rather than tens of minutes required by the common batteries. Importantly, the boolean function indicating the bias directly describes the pattern responsible for this bias. This allows to construct the bit predictor or to fix the cause of bias in the function design. The present bias is typically detected in at least an order of magnitude less data than required by NIST STS or Dieharder. The tests included in these batteries are either too simple to spot the common biases (like the Monobit test) or overly complex (like the Fourier Transform test) requiring an extensive amount of data. The proposed approach called *BoolTest* fills this gap. The performance was verified on more than 20 real world cryptographic functions – block and stream ciphers, hash functions and pseudorandom generators. Among others, the previously unknown bias in the output of *C rand()* and *Java Random* generators was found.

1 Introduction

A newly designed cryptographic primitive (block cipher, stream cipher, hash function, pseudo-random generators etc.) is subjected to a cryptanalysis looking for flaws or information leakage in the primitive design. Standard cryptanalysis techniques like linear, differential and algebraic cryptanalysis are used to find correlations between input, output and key bits (if used). The existence of correlated bits indicates a weakness of the function. Although these techniques can be partially automated, the aid of the skilled cryptanalyst is still needed.

Fully automated but weaker statistical test suites (e.g., NIST STS, Dieharder, TestU01) are often used before cryptanalysis. Commonly, well-designed crypto-primitives should produce output with the same characteristics as random data. Test suites examine the correlation of function output bits through randomness analysis of data it produces. Each test suite (often called battery) usually consists of tens of empirical tests of randomness. Each test looks for a predefined pattern of bits (or block of bits) in data and thus it examines randomness from its specific point of view.

Although there is an unlimited number of tests in principle, batteries opt for implementation of only several selected ones for the practical reasons. Each test computes a histogram of a specific feature of bits (or block of bits). The histogram is statistically compared with the expected histogram (for random data). The result (*p-value*) of the test is probabilistic measure how well both histograms match. Data are considered to be non-random if histograms differ significantly. The randomness in such a context is a probabilistic property and we can commit two types of errors – Type I (truly random data are rejected) and Type II (non-random data are not rejected).

Batteries implement many tests of various complexity – from a very simple Monobit computing statistic of bits (frequency of ones and zeros) to the very complex statistic of large blocks (computation of linear profile). The complexity of tests usually determines the amount of data needed to compute histograms for comparison. In order to decrease the Type I and II errors, sufficiently many data sequences are required in practice. The complex and usually slow tests require up to GB's of data which can be hard to generate in some cases, while several MB's are suffi-

Implementation and paper supplementary material can be found at https://crocs.fi.muni.cz/papers/secrypt2017.

cient for very simple tests.

We can identify following generic limitations of batteries with respect to the analysis of cryptographic functions:

1. **An insufficient strength to detect bias in complete function** – The tests included in a battery are usually too weak to detect biases in an output of a modern cryptographic function with a full number of rounds and other security parameters.
2. **An insufficient sensitivity to detect bias if small amount of data is provided** – The tests might be too weak to detect biases when an only limited amount of data available is for the testing. The tests usually require from 10MB up to several GBs of data which may not be available or efficient to collect in particular test scenario.
3. **The difficulty of results interpretation** – The interpretation of test results is often only general in the form "something is wrong with data/function". Only few tests are able to identify concrete dependent bits and provide this crucial information to cryptanalyst.

Our goal is to resolve the last two aforementioned problems and to construct the set of statistical tests that will be stronger in detecting the bias when given a limited amount of data yet directly identifying the biased output bits. In fact, we are looking for the strongest distinguisher (of cryptographic function) from randomness possible within the given amount of data and complexity of a distinguishing function. A distinguisher is iteratively constructed in the form of simple function starting from the simplest possible and proceeding towards the more and more complex boolean functions. Surprisingly, such a test is missing from all three commonly used test suites. Our approach is a generalization of the simple Monobit test and was practically tested on the wide range of cryptographic functions of various types – block and stream ciphers, hash functions and pseudo-random number generators (PRNGs). We have found practical strong distinguishers which can be also used as the bit predictors (although usually weak) for the corresponding functions.

In short, we make the following contributions:

- **Simple, yet strong test:** We designed the principally simple, yet surprisingly strong test called *BoolTest* based on the boolean functions. The output of cryptographic functions like AES or SHA-3 (with over 20 tested) with reduced number of rounds still distinguishable are comparable to commonly used statistical batteries.
- **Interpretable test for small data:** We have shown that *BoolTest* not only requires significantly fewer data and runs faster (seconds) but also allows for the direct interpretation of a distinguisher found – which particular bits in tested function output are biased together and how.
- **Practical distinguisher for *C/Java rand*:** Among others, we found previously unknown biases in the output of *C rand()* and *Java Random* pseudo-random generators forming surprising strong practical distinguishers regardless of the initial seed used. A deeper analysis of these distinguishers is provided.
- **Open-source implementation:** We release the code of *BoolTest* (Sýs and Klinec, 2017) as an open-source to facilitate further research in this area and complement the standard test batteries.

The paper is organized as follows: in Section 2, we describe a commonly used tests of randomness together with related work and motivation for more efficient tests. The Section 3 provides background and detailed description of our strategy for distinguisher construction based on boolean functions with relevant implementation details which significantly speed up the computations. The comparison of results with common statistical batteries on more than 20 functions are provided in Section 4 together with the detailed discussion of practical distinguishers found for the Java and C pseudo-random generators. The Section 5 is devoted to the statistical interpretation of results, followed by conclusions given in Section 6.

2 Related work

NIST STS (Rukhin, 2010), Dieharder (Brown et al., 2013) (a extended version of the Diehard) and TestU01 (L'Ecuyer and Simard, 2007) are the most commonly used batteries for statistical randomness testing. The NIST STS is the basic battery required by NIST to test RNGs of cryptographic devices by the FIPS 140-2 certification process (NIST, 2001) with four of NIST STS tests required as power-up tests executed on-device. The Dieharder battery is an extension of original Diehard battery (Marsaglia, 1995) with some (but not all) NIST STS tests also included. The Dieharder is generally more powerful than NIST STS with the ability to detect smaller biases but also requires more input data.

TestU01 can be viewed as a current state of the art of randomness testing. TestU01 is a library that implements more than 100 different tests of randomness. These tests are grouped into 6 sub-batteries called small Crush, Crush, Big Crush, Rabbit, Alphabit, BlockAlphabit. The first three sub-batteries are proposed to test floating point random numbers from the interval [0,1]. Small Crush (10 tests), slower Crush (96 tests) and very slow but powerful Big Crush (all 106 tests). The amount of data used for analysis increases with the number of tests and their complexity. The small Crush/Crush/Big Crush need at least 206MB/2.6 GB/51.3GB data to run all tests of the battery. Other three batteries are proposed for testing the binary sequences specifically. The Rabbit (26 tests), Alphabit and BlockAlphabit (9 tests) batteries are not limited in fact (Rabbit is restricted to 500 bits) in the size of data they need for the analysis.

Batteries analyze data with an assumption that data were generated by a black box function. It is clear that more information we have about the generator better randomness analysis we can perform. There are three basic approaches (linear, differential and algebraic cryptanalysis) for randomness analysis of data produced by a primitive which are based on its internal structure. Nice tutorial on linear and differential cryptanalysis can be found in (Heys, 2002). Various methods of algebraic cryptanalysis are described in the book (Bard, 2009). There are several automated tools that implement aforementioned approaches. These tools look for dependency between inputs and outputs of the primitive (and key, IV bits). List of current such cryptanalytical methods and tools implemented in recent years can be found at (Mouha, 2010).

In (Filiol, 2002) a new and strong method of statistical testing of hash functions and symmetric ciphers was proposed. In this approach each output bit is described as a boolean function in the algebraic normal form (ANF). The test statistic is based on a number of monomials in ANF. Since number of monomials is exponential in number of variables, the randomness is evaluated based on a number of monomials of degree exactly d which has χ^2 distribution for random boolean functions. Another automated cryptanalytic tool (Englund et al., 2007) is based on the strong d-monomial test. In (Englund et al., 2007) monomial test was generalized to perform chosen IV statistical attacks on stream ciphers. In (Stankovski, 2010), a greedy method was proposed to find distinguishers from randomness for stream and block ciphers. The method is based on maximum degree monomial test similar to d-monomial test. Previous methods are based on ANF of analyzed function which is statistically compared with ANF of random boolean function expected for random data. This is completely different to our approach where boolean function itself defines the statistic of a test of randomness.

The automated testing tool for cryptographic primitives named Cryptostat (Kaminsky and Sorrell, 2014) is focused on testing block ciphers and message authentication codes. Cryptostat consists of several tests each computing the probability that block of bits of the ciphertext equals to bits taken from plaintext and key. Bits are selected either randomly or block of consecutive bits are taken. The tests of CryptoStat are reducible to Bernoulli trials and they are evaluated using Bayesian conditional probability.

Hernandez and Isasi proposed an automated construction of distinguisher for TEA block cipher (Hernández and Isasi, 2004). The distinguisher in the form of input bitmask of the 192-bit block is searched for. As the search space of all possible bitmasks is too large, a heuristic based on genetic algorithm was used to construct a distinguisher for TEA limited up to 4 rounds. In the (Garrett et al., 2007), authors optimized the Hernandez's approach with quantum-inspired genetic algorithm and found distinguisher for TEA limited to 5 rounds.

The similar but more general approach is used in EACirc framework (EACirc, 2017) which constructs distinguisher (test of randomness) for crypto primitive without knowledge about primitive design (black-box). In the EACirc test of randomness is constructed for the predefined representation as circuit-like software over the set AND, XOR, NOR, NOT of boolean operations. The ciphers with a limited number of rounds were tested with results comparable to NIST STS battery. Although the Dieharder battery still provides overall better randomness analysis EACirc was able to detect some non-randomness for Hermes and Fubuki (Sýs et al., 2014) where both batteries failed to detect any deviances.

2.1 Motivation for better tests

Tests in batteries can be roughly divided into three main categories w.r.t. their complexity. 1) The very simple tests compute statistic of bits (e.g., histogram of ones and zeros) within an entire tested sequence or within smaller parts of the whole sequence. 2) The slightly more complex and usually slower tests compute statistic of a small block of bits (e.g., an entropy of 8-bit blocks) within a sequence. 3) The complicated and slow tests compute a complex statistic (e.g., the histogram of rank for matrices, linear complexity) within the large parts of the sequence.

How well the common batteries perform in the analysis of crypto primitives? Let's take the 100MB data produced by truly random number generator (which should pass all tests), divide it into 128 bits blocks and introduce minor modification to original random stream – the last bit (b_{127}) of every block is changed so that *xor* with the very first bit (b_0) of that block gives always 0 as the result ($b_0 \oplus b_{127} = 0$) instead of in only half of the cases as expected. Even such a strong bias is detected only by a handful of tests, most significantly by the block frequency test. If the resulting 0 is produced 1% more frequently than 1 (instead of always as previously), only one test of the Crush battery detects the bias. And for 0.1% none of the standard tests (batteries NIST STS, Dieharder, Crush) detect this – still significant – bias. The problem lays in a structure of patterns tests of batteries look in data. The Figure 1 shows the patterns of bits as analysed by standard batteries. Dieharder and NIST

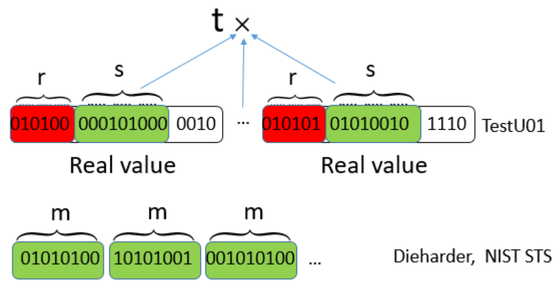


Figure 1: The patterns analyzed by the TestU01, NIST STS and Dieharder battery.

STS batteries analyse randomness according to consecutive m bits for small m (typically $m < 20$). The tests included in TestU01 take a different approach as data are transformed into series of real values with first r bits (of every real value) discarded and only next s bits used for the analysis. TestU01 analyses data usually as point in k dimensions and thus t consecutive blocks of s bits represent point in t dimensions. Values of r, s are typically in range $[0, 32]$ and t is usually small value < 10.

Very simple and bit oriented tests like Monobit test are usually also the fastest. Besides the speed, the additional advantage of simple tests is usually the small amount of data necessary to compute correct results (statistic distribution is approximated well). The more complex tests need significantly more data for the sufficient approximation and thus also for detection of bias (if present). Another drawback of standard tests is a lack of possibility to retrospective identify the exact biased or correlated bits even when a test is able to detect some bias. The observed statistic computed by a test is given by frequencies (histogram) of some feature. For more than two bins we are usually unable to identify which bin has unexpectedly high or low value (w.r.t. reference values). Hence we cannot identify the concrete input bits responsible for the production of the extreme value of the observed statistic. On the other hand, if histogram contains only two bins, the value in one bin automatically determines the value of the second bin.

According to the previous reasoning, the histogram (of frequencies) should preferably consist of two bins. To identify the biased or correlated bits, the searched relation should be bit-oriented as well. One statistical test of randomness can be used to examine only one relation of specific bits (within a block). In order to find correlated bits, we need to repeat the process many times with many different relations and bits selected. The time required to evaluate the tests should be reasonably small and therefore the inspected relation represented as a simple boolean function is a natural choice. A relation analysed expressed as a simple boolean function is fast to compute as only bitwise operations are used to compute the required histogram. Moreover, the exact (and not only approximated) reference distribution expected for the truly random data can be computed analytically. Finally, one can easily order two candidates (boolean functions) based on their complexity (degree and number of components) and find the simplest function which exhibits unexpected bias thus providing a more sensible guide for cryptanalyst. The following Section provides more details for the constructions of such distinguishers.

3 The randomness distinguisher based on the boolean functions

Our approach is inspired by the Monobit test which examines the proportion of ones and zeros within the provided sequence. The frequencies of ones and zeros computed in Monobit represent results of a boolean function $f(x_1) = x_1$ when applied to all bits of the sequence. This can be generalized to an arbitrary boolean function $f(x_1, x_2, \cdots, x_m)$ of m variables applied to non-overlapping blocks of m bits.

In our approach, we construct set of boolean functions (potentially distinguishers) defining different tests of randomness. All tests (functions) are applied the same way (see Section 3.2) to given sequence resulting in a set of test statistics. The results of our approach are the maximal observed test statistic and the corresponding boolean function.

The maximal observed test statistic and the boolean function can be used to evaluate the randomness of analysed sequence or a new sequence:

- Maximal observed test statistic can be directly used to assess the randomness of analyzed sequence. The interpretation of maximal test statistic is based on the distribution of maximal test statistic obtained for reference random data (see Section 5).

- Found boolean function can be also used to assess the randomness of a new sequence from the same source as described in Section 3.2.

The distinguisher (boolean function) is constructed iteratively from simpler and weaker distinguishers (simpler boolean functions). Besides the fact that simpler distinguishers are found first, this allows also to speed up the entire process since many intermediate computational results (for simpler functions) can be reused.

3.1 Test of randomness

The majority of empirical randomness tests are based on the statistical hypothesis testing. Tests are formulated to evaluate the null hypothesis – "data being tested are random". Each test computes a specific statistic of bits or block of bits which is a function of tested data. Firstly, a histogram of patterns for the given dataset is computed by the test. Then the histogram is transformed into a single value – observed test statistic which represents randomness quality of a sequence according to an analysed feature. The distribution (null distribution) of the test statistic under the null hypothesis (data are random) is used to evaluate the test. Exact null distribution of a test statistic is usually complex function hence its close approximation is used instead. The most of the tests have χ^2 or *normal distribution* as their null distribution. A test checks where the observed test statistic appears within the null distribution. The hypothesis is rejected if value happens to be in extreme parts of the null distribution (tail). In such a case, the tested data are considered to be non-random. An observed test statistic is usually transformed to a *p-value* (using the null distribution). The *p-value* represents the probability that a perfect random number generator would have produced a sequence "less random" (more extreme according to analysed feature) than the tested sequence (Rukhin, 2010). *P-value* is compared with the significance level α typically set to smaller values $0.01, 0.005$ or 0.001 for the randomness testing. If the *p-value* is smaller/bigger than α hypothesis is rejected/accepted and data are considered to be non-random/random. The following example illustrates how *p-value* is computed for Monobit test.

Example 1. *The Monobit test examines whether number of ones (#1) and zeros (#0) in a sequence of n bits are close to each other as would be expected for random data. The test statistic is computed as* $s_{obs} = \frac{|\#0-\#1|}{\sqrt{n}}$. *The reference distribution of the test statistic is half normal as stated in (Rukhin, 2010) but this is just approximation. The* p-value *is computed in Monobit test as:*

$$p\text{-}value = erfc\left(\frac{s_{obs}}{\sqrt{2}}\right) = erfc\left(\frac{|\#0-\#1|}{\sqrt{2n}}\right)$$

using the well-known complementary error function ($erfc$)(Press et al., 2007). Same p-value *can be computed for statistic $s_{obs} = \#1$. The exact distribution of* $\#1$ *is binomial distribution $B(n, 0.5)$ for a sequence of n bits. Figure 2 illustrates the exact reference binomial distribution for $s_{obs} = \#1$ and sequences of $n = 100$ random bits (bins). The figure also shows that the discrete binomial distribution can be approximated well by the continuous normal distribution for sufficiently large n (documentation of NIST STS recommends $n \geq 100$). The* p-value *represents the probability that RNG would generate data with more extreme test statistic than s_{obs}. A* p-value *can be computed as an area below the normal distribution in the tail bounded by the observed test statistic s_{obs}. Figure 2 illustrates the value of* p-value *for $n = 100$ and $s_{obs} = 56$.*

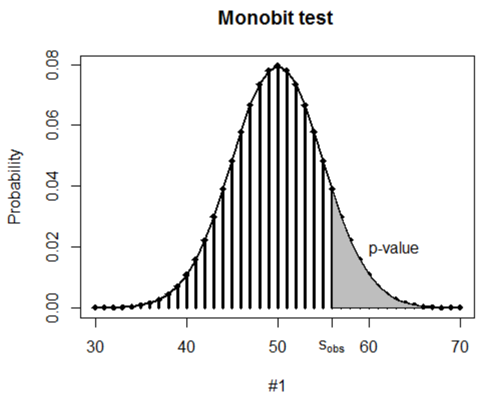


Figure 2: Discrete binomial distribution $B(100, 0.5)$ and its approximation by continuous normal distribution $\mathcal{N}(50, 25)$. Area in the right tail represents *p-value* for the test statistic defined by $s_{obs} = \#1$ for a sequence with $n = 100$ bits.

3.2 Distinguisher evaluation

In order to evaluate the strength of the distinguisher (test), we use common principles from randomness testing. We adapt and generalize the Monobit test. The distinguisher (boolean function) defines the test of randomness and the computed test statistic is used directly as the measure of the strength of distinguishers. A bigger value of observed statistic means stronger distinguisher and conversely. To generalize the Monobit test, let us characterize steps of a test of randomness.

An empirical test of randomness consists (in general) of the following steps:

1. Compute the histogram H of some features (within data).
2. Compute (transform the histogram to) the observed test statistic s_{obs}.
3. Compute the null distribution (exact or its close approximation) $D(x)$ of the test statistic under the null hypothesis (random data).
4. Compute the *p-value* from s_{obs} using the distribution $D(x)$.

In our approach, the histogram of results of the boolean function $f(x_1,\cdots,x_m)$ of m variables applied to non-overlapping m-bit blocks of the sequence is computed. Our test statistic is *Z-score* (Sheskin, 2003) defined as:

$$Z\text{-}score = \frac{\#1 - pn}{\sqrt{p(1-p)n}}, \qquad (1)$$

which normalize a binomial distribution $B(n,p)$. Binomially distributed variable #1 is normalized to *Z-score* which is distributed normally. *P-value* can be directly computed from the *Z-score*. Figure 3 illustrates the relation of a *Z-score* (standardly expressed in the units of standard deviation x.σ) and the corresponding *p-value* (area of two tails).

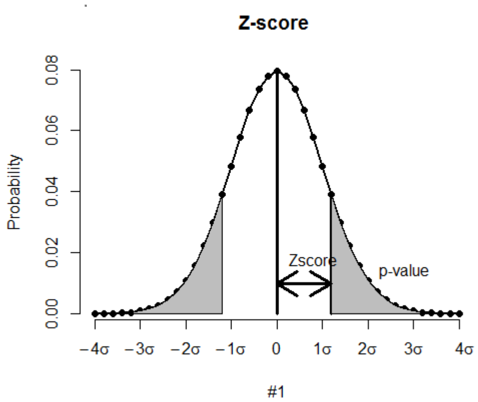


Figure 3: The relation of *Z-score* and *p-value*. *Z-score* is expressed in the units of the standard deviation.

The symbol p denotes the probability that the result of boolean function f is equal to 1 for random input. Symbol n denotes the number of non-overlapping blocks (of m bits) in the analysed sequence (not the number of bits). Similarly, as in the Monobit test, our histogram consists of two frequencies #1 and #0 but only #1 is computed ($\#0 = n - \#1$) and is used for the evaluation. The only difference is that the expected probability p is not $p = 0.5$. In general, p is arbitrary value from the interval [0,1] which depends on the given boolean function f. The *Z-score* and relevant statistical theory is discussed in Section 5 in more details.

Figure 4 illustrates our approach with the boolean function $f(x_1,\cdots,x_m) = x_2 + x_{89} + x_{94}$. Firstly, data to be analyzed are divided to multiple non-overlapping blocks. Then the number of results equal to one (#1) is computed (blocks serve as the inputs for the function f). The final result – *Z-score* is computed as the statistical distance between observed and expected number of ones (#1).

To perform the test, we have to compute only #1 and the expected probability p (as the p changes with the function f). The algorithm for the computation of p is described in Section 3.5. We may omit the computation of the *p-value* since the strength of distinguishers can be compared directly using their *Z-scores*. The bigger *Z-score* is the stronger distinguisher is obtained and vice versa.

3.3 Distinguisher construction

Our approach assumes that stronger and more complex distinguishers can be obtained as a combination of the weaker and simpler ones. This assumption is natural in a sense that if this would not be true we have to find more complex distinguishers by brute force anyway. As we start with a test of the simpler candidate distinguishers first, we naturally obtain the simplest possible yet strong enough distinguisher. The potentially stronger, but more complex distinguishers are evaluated later. We work with the boolean functions of m variables for some fixed m. The construction is iterative. We first start with the simplest boolean functions $f(x_1,\cdots,x_m) = x_i$ for $i \in \{1,2,\cdots,m\}$ and construct more and more complex (more monomials, higher degree) functions. Since we want to find the weakness (biased bits) in the output of a tested cryptographic function, the number of variables m of a boolean function should correspond with the size of function's output. Therefore, the value of m is set to $m = 128$ or to its small multiple $256, 384, 512$ to match frequent block sizes used in common cryptographic functions. For such small values of m, we can check all such simple boolean functions by brute force. The construction is divided into two phases:

1. Firstly, the set S of k strongest and simple distinguishers is found: We search through the set of monomials (x_i, $x_i.x_j$, $x_i.x_j.x_k$) of small degree $\leq deg$ since a total number of functions raise expo-

In most cases distribution $D(x)$ is given.

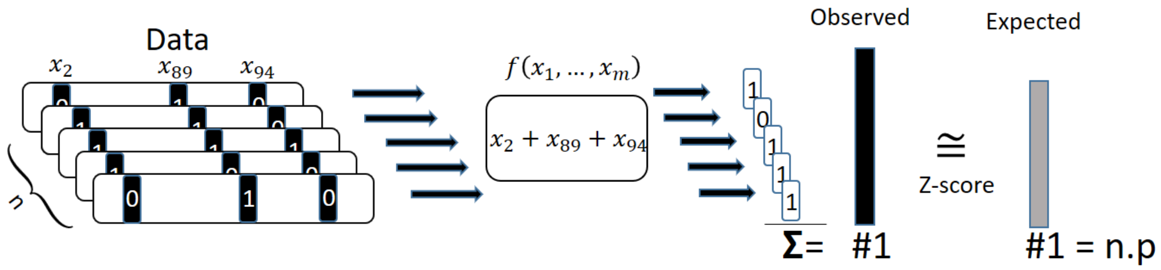


Figure 4: Our approach and the computation of *Z-score* using boolean function $f(x_1,\cdots,x_m) = x_2 + x_{89} + x_{94}$. *Z-score* is computed as the statistical distance of observed #1 for tested data and #1 $= p.n$ expected for truly random data.

nentially with the degree. We assess the strength of all monomials and set S of strongest (biggest *Z-score*) t distinguishers ($|S| = t$) is sent to the next phase.

2. In the second phase, we construct more complex distinguishers: The simple distinguishers (elements of S from the first step) are combined using the addition (XOR) operator. We construct all possible functions in the form of $f(x_1,\cdots,x_m) = b_1 + b_2 + \cdots + b_k$ such that $b_i \in S$ and k is fixed.

The advantage of the described process is that the simple boolean functions are tested first and if the sufficiently strong distinguisher (large *Z-score*) is found the process can be terminated at any point. Moreover, construction of complex boolean function from simpler allows reusing the intermediate results (distribution of ones and zeroes) computed in the earlier stages to significantly improve the entire performance.

3.4 Implementation details

A result of the boolean function $f(x_1,\cdots,x_m)$ can be computed efficiently using fast bitwise operators AND and XOR. Moreover, these operators allow us to compute 32, 64 or 128 results at once (based on the CPU architecture and the instruction set). The principle follows the way how the distinguisher is constructed. We firstly compute "basis" of results for the simple boolean functions when applied to all input blocks (of m-bits) of a given sequence. Then the basis vectors are used to compute results for the arbitrary complex boolean function applied to the same inputs.

- Firstly, a "basis" of results is constructed. For each variable $x_i, i \in \{1,\cdots,m\}$ we fill the basis vector X_i (bit vector) by results of boolean function $f_i(x_1,\cdots,x_m) = x_i$ when applied to all input m-bit blocks of the tested data.
- The vector of all results X_f of the function f can be computed using our vector basis X_i in the same way as result of f is computed using x_i. In fact, to compute the vector of all results it suffices to perform same operations with vectors X_i instead of x_i where AND and XOR are operators of boolean vectors now. The basis vectors are packed into words for more efficient computation.

The principle can be illustrated on the following example. Let assume that we want to compute 64 results of the boolean function $f(x_1,x_2,x_3,x_4) = x_1x_2 + x_3$ for 64 blocks B_i each having 4-bits. We firstly compute basis bit-vector X_i that represents results (64 bits) of boolean function $f_i(x_1,x_2,x_3,x_4) = x_i$ applied to all blocks B_i. Vector of results X_f for the function f (applied to B_i) can be computed as

$$X_f = (X_1\ AND\ X_2)\ XOR\ X_3$$

for operators AND,XOR working with bit-vectors. The vector of results X_f can be computed using just two bitwise operations working with 64-bits words. The longer sequences should be divided into words of 64 bits.

In our approach, boolean functions of a small degree and with the small number of monomials (t) are constructed. Therefore vectors $X_i, X_{i,j} = X_i\ AND\ X_j$ corresponding to functions $x_i, x_i.x_j$ are fully precomputed and used as the basis for result computation.

3.5 On the computation of expected p

Determining p, i.e., the probability of evaluating polynomial f to 1 under the null hypothesis that tested data are random, is equivalent to finding all variable settings under which f evaluates to 1. This problem is exponentially complex with the size of the f.

Let p_i be the probability of x_i evaluating to one, $P(f)$ the probability of f evaluating to 1 under all possible settings. The basic cases are:

1. $P(x_ix_{i+1}\cdots x_{i+k-1}) = p_ip_{i+1}\cdots p_{i+k-1} = 2^{-k}$
2. $P(x_i + x_j) = p_i(1-p_j) + (1-p_i)p_j$
3. $P(x_i + x_j + x_k) = P((x_i + x_j) + x_k)$ using associativity and the rule 2.

Using these rules it is easy to determine $P(f)$ for a general polynomial in algebraic normal form (ANF) in linear time w.r.t. a number of variables (under the assumption of disjoint terms). However, the evaluation is more time-consuming if the terms are dependent as the relations above do not hold. The solution for the problem with dependent terms requires to evaluate a polynomial for all possible variable settings, then count the number of cases where $f(x) = 1$ and finally compute resulting $P(f)$. This time complexity of the algorithm is exponential with respect to the number of variables.

We use few tricks to reduce the computation time. Let denote $f = b_1 + b_2 + \cdots b_k$, where $b_i = \prod_{j=1}^{deg} x_j$ is a term of degree deg. If $deg(f) = 1$ the rule 1 is used. In case of dependent terms we fall-back to naïve algorithm – evaluate f in all settings.

As example, lets examine the polynomial $f_1 = x_1x_2x_3 + x_1x_5x_6 + x_7x_8x_9$. Using naïve approach the f_1 is evaluated 2^8 times. With the closer look it can be evaluated as: $P((b_1 + b_2) + b_3)$, as b_3 is independent of other terms so whole evaluation is done only in 2^5 steps and one rule 2 application. To generalize this trick we just need to compute dependency components between terms b_i.

The terms b_i, b_j are dependent if $b_i \cap b_j \neq \emptyset$, i.e., they share at least one variable. The trick is to apply the naïve algorithm to all dependent components of the polynomial, then merge the components using rules $2,3$ as they are pairwise independent. Component finding is implemented with *Union-Find* algorithm with complexity $O(\alpha(n))$ which yields the resulting complexity $O(n\,\alpha(n))$

To further optimize the evaluation, we can convert the input polynomial to a canonical form by renumbering the variables and sorting the terms. E.g. $x_{60}x_{120}x_{48} \rightarrow x_1x_2x_3$. Then by caching previous computations (e.g., LRU), we can avoid some expensive computations in a dependent component evaluation.

Another optimization is to use pruning and recursive application of the rules above when evaluating dependent components. Consider $b = x_1x_2x_3 + x_1x_5x_6$. In branch $x_1 = 0$ we directly have $b = 0$ thus all evaluation sub-branches are pruned. In branch $x_1 = 1$ we have $b' = x_2x_3 + x_5x_6$. By applying the algorithm recursively, we see x_2x_3, x_5x_6 are independent and no naïve algorithm is used, only rules $1,2,3$.

In practice, we use polynomials and terms of a relatively small degree so we don't use optimization with pruning and LRU caching as evaluating terms by the naïve algorithm is faster with this sizes. The overall benefit is the fast dependent component detection and in practice, the vast majority of polynomials have independent terms which yield very fast $P(f)$ computation, in $O(n\,\alpha(n))$.

4 The results

To demonstrate the practical usability of proposed approach, we tested the approach on a variety of cryptographic primitives – hash functions, block and stream ciphers and (pseudo-) random number generators (PRNG). The results are compared with the existing automated approaches utilized by the randomness statistical test batteries NIST STS, Dieharder and TestU01. The data used for analysis were generated as keystream for stream ciphers or encrypted/hashed incremental counter for block ciphers and hash function with random fixed key used.

4.1 Parameters of boolean functions

Our approach is parameterized by the parameters deg, m, t and k. We search for the distinguisher with m variables and of k monomials each with degree of deg. The parameter t represents the number of best monomials used to construct distinguisher in the second phase (as described in Section 3.3). For instance, parameters $deg = 2, m = 4, t = 128, k = 3$ means that we searched for 128 strongest distinguishers (boolean functions) of the form $f(x_1,x_2,x_3,x_4) = x_i.x_j$ for different $x_i, x_j \in \{x_1,x_2,x_3,x_4\}$ in the first phase. In the second phase we combine every k-tuple of them to find the strongest distinguisher of the form $f(x_1,x_2,x_3,x_4) = x_i.x_j + x_k.x_l + x_r.x_s$ among the all possible combinations. We tested data produced by various crypto functions with various settings. We used the combination deg, m, t, k where $deg \in \{1,2,3\}, m \in \{128,256,384,512\}, t = 128, k \in \{1,2,3\}$.

4.2 Common cryptographic functions

In order to compare our results with the standard batteries, we tested the data also with NIST STS, Dieharder and TestU01 test suites (Alphabit, Block-Alphabit, Rabbit, small Crush). The Table 1 summarizes the results and strength of tools according to a number of rounds for which deviation from distribution expected for random data (null hypothesis) is detected by the respective tool for 100MB of data. We consider data fails a battery if it fails one of its tests with the conservative significance level set to $\alpha = 1\%$. The Table 1 shows the best results of our tool obtained for two particular settings $Bool1(deg = 2, k = 1, m = 384)$ and $Bool2(deg = 2, k = 2, m = 512)$. In 15 out of

Table 1: The number of rounds (of selected primitives and PRNGs) in which non-randomness was detected for 100MB data for NIST STS (NI), Dieharder (Di) and TestU01 (U01). Our approach is presented for two well performing settings Bool1($deg = 2, k = 1, m = 384$) and Bool2($deg = 2, k = 2, m = 512$). Character '$<$' means that more rounds were distinguished by boolean function found with other parameters than two presented.

function	NI	Di	U01	Bool1	Bool2
AES	3	3	3	3	3
ARIRANG	3	3	4	3	3
AURORA	2	2	4	2	2
BLAKE	1	1	1	1	1
Cheetah	4	4	6	4	4
CubeHash	0	0	1	0	0
DCH	2	2	2	1	1
Decim	6	6	6	5	5
Echo	1	1	1	1	1
Grain	3	2	2	2	2
Grostl	2	2	2	2	2
Hamsi	0	0	0	0	0
JH	7	6	9	6	6
Keccak	2	3	4	3	3
Lex	3	3	3	3	3
Lesamta	2	3	3	2	2
Luffa	7	7	7	7	7
MD6	8	8	9	9	8
SHA256	4	4	3	3	4
Simd	0	0	0	0	0
Salsa20	6	4	6	<4	<4
TEA	3	4	4	4	4
TSC-4	13	12	13	<13	<13
Twister	6	6	7	6	6

24 functions tested, *BoolTest* was able to detect non-randomness in stream produced by the same number of rounds in round-reduced cryptographic functions when compared to NIST STS. The more and fewer rounds were distinguished for Keccak, MD6, TEA and DCH, Decim, Grain, JH, Salsa20, TSC-4 functions respectively.

It should be noted that *BoolTest* was able to find boolean functions with other parameters than *Bool1* and *Bool2* capable of detecting non-randomness of TSC-4 reduced to 14 rounds and Salsa20 with 4 rounds, but performing same or worse on the remaining configurations.

The second practically important property of any test is the least amount of data necessary to spot the bias if present. We tested and compared the performance of *BoolTest* with statistical batteries using 10MB, 100MB, and 1GB of input data. The results are summarized in the Table 2. For test suites, the number of passed tests are shown. The computed *Z-scores* are shown for the *BoolTest* and two best settings according to given a set of analysed functions. The results of *BoolTest* and test suites which can be interpreted as detected non-randomness (null

Table 2: Results of NIST STS (NI), Dieharder (Di), TestU01 (U01) and our approach with two settings Bool3($deg = 1, k = 2, m = 384, t = 128$) and Bool4($deg = 1, k = 2, m = 512, t = 128$) obtained for 10MB, 100MB and 1GB of data produced with primitives with limited number of rounds.

size	func	NI	Di	U01	Bool3	Bool4
	AES (3)	∀	18	15	8.6	6.7
	TEA (4)	∀	20	∀	20.6	11.5
	Keccak (3)	∀	∀	15	3.7	5.3
10MB	MD6 (9)	∀	∀	∀	3.9	13.3
	SHA256 (3)	0	0	6	88.7	242
	AES (3)	∀	16	15	8.9	15.0
	TEA (4)	14	21	∀	73.6	5.2
	Keccak (3)	14	22	15	3.8	9.2
100MB	MD6 (9)	∀	∀	∀	3.7	26.4
	SHA256 (3)	0	0	4	50.7	828
	AES (3)	9	18	14	12.8	41.2
	TEA (4)	13	24	∀	127	4.3
	Keccak (3)	∀	26	15	3.5	32.0
1GB	MD6 (9)	13	25	15	4.1	26.4
	SHA256 (3)	0	1	3	78.0	3043

hypothesis rejected) are highlighted in gray. Based on the results, we can conclude that test based on boolean functions usually requires an order of magnitude fewer data to detect bias than common batteries.

4.3 Pseudo-random number generators

The proposed approach was tested on several commonly used non-cryptographic pseudo-random number generators (PRNGs): Mersenne Twister 19937, Multiply-with-Carry C++ generator, Ranlux24, T800, TT800 from TestU01 and *C stdlib rand()* and *Java java.util.Random*. The practical distinguishers were found for the last two generators (as discussed below) and no distinguisher was found for any tested parameters and data sizes up to 1 GB for the remaining ones.

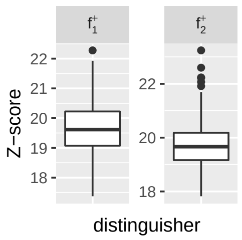
Using *BoolTest*, we were able to find universal distinguishers i.e., which work for large groups of PRNG seeds, for *C stdlib rand()* and *Java java.util.Random* (*C rand*, *Java rand* in short). We tested *BoolTest* on 1000 different bit streams generated by the C *rand* respectively, each bit stream generated by using a different random seed from the interval $[0, 2^{32} - 1]$.

Let define an input bit stream as τ_i and the best distinguisher and its corresponding *Z-score* value for τ_i returned by *BoolTest* as (ξ_i, δ_i). Figures 5 and 6 depict the set of the best distinguishers $\xi_i \in \{f_1, f_2, f_3, f_4, f_5, f_6\}$ and their *Z-scores* found by *BoolTest* on input bit streams $\tau_1, \ldots, \tau_{1000}$. In order to emphasize the *Z-score* deviation polarity each distinguisher has the *Z-score* results are split into two box plots, for positive and negative *Z-scores* values. The number of occurrences of the distinguisher f_1^+ is $|f_1^+| = |\{i; \xi_i = f_1 \wedge \delta_i \geq 0\}|$. E.g., the f_1^- column

$f_1, f_2, f_3, f_4, f_5, f_6$ are particular boolean functions.

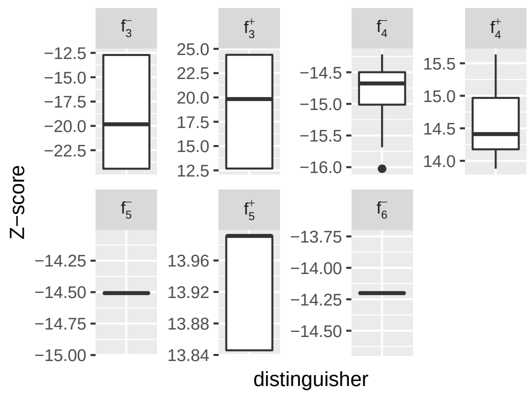
represents all the *Z-score* values $\delta_i < 0$ where $\xi_i = f_1$ and the f_1^+ column represents $\delta_i \geq 0$ where $\xi_i = f_1$.

Note for the *C rand* the deviation is only positive while for *Java Random* it is usually symmetric.



$$f_1 = x_{71}x_{319} + x_{295}x_{379}$$
$$f_2 = x_{88}x_{375} + x_{127}x_{351}$$

Figure 5: The best distinguishers, *C rand()*, 1000 x 1 MB data samples, 384 bit block, random 32 bit seed, Ubuntu 16.04. The best distinguisher occurrences in 1000 tests: $|f_1^+| = 520$, $|f_2^+| = 480$



$$|f_3^-| = 352,\ |f_3^+| = 374,\ |f_4^-| = 48,\ |f_4^+| = 86$$
$$|f_5^-| = 45,\ \ |f_5^+| = 63,\ \ |f_6^-| = 32,\ |f_6^+| = 0$$

$$f_3 = x_{38}x_{326} + x_{39}x_{327} + x_{326}x_{486}$$
$$f_4 = x_{38}x_{326} + x_{205}x_{327} + x_{326}x_{486}$$
$$f_5 = x_{38}x_{326} + x_{326}x_{486} + x_{327}x_{359}$$
$$f_6 = x_{38}x_{326} + x_{167}x_{327} + x_{326}x_{486}$$

Figure 6: The best distinguishers, *Java Random*, 1000 x 1 MB data samples, 512 bit block, random 32 bit seed. Java OpenJDK 1.8.0_121, Oracle Java 1.7.0_6, 1.8.0_65

The distinguishers from the Figure 6 were discovered with the parameters $(deg = 3, k = 3, m = 512)$. In this setting the *BoolTest* examined input bit stream of increasing sizes: $\{19200, \ldots, x_j, 2x_j, \ldots, 300 \cdot 1024^2\}$ bytes and found $\{f_3, f_4, f_5, f_6\}$ distinguishers after examining 37.5 MB bit stream. In the previous iterations with smaller input bit stream only weak distinguishers were found. When using different settings $(deg, m, k) \in \{\{1,2,3\} \times \{128, 258, 384, 512\} \times \{1,2,3\}\}$ we were able to find only weaker distinguishers which required significantly more data to achieve the same *Z-score*. Interestingly, $\{f_3, f_4, f_5, f_6\}$ we discovered after examining 37.5 MB bit stream work very well also for smaller data sizes, as depicted on figure 3.

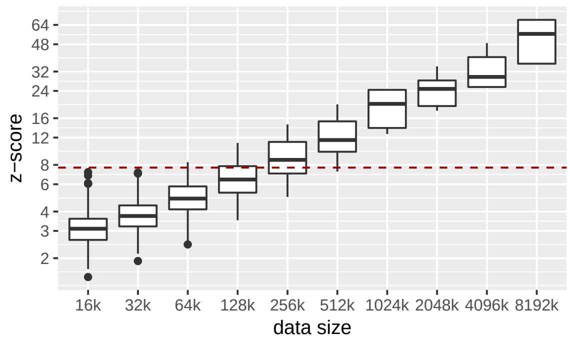


Figure 7: The size of the input bit stream vs. $|$*Z-score*$|$ using distinguishers $\{f_3, f_4, f_5, f_6\}$ for *Java Random*, 1000 random seed samples per data size category. Dashed line represents ref. *Z-score* value for the test.

It is evident there exists a good distinguisher of a low degree exists for *Java Rand* but due to *top-heuristics* the *BoolTest* was not able to find it with other combinations rather than $(deg = 3, k = 3, m = 512)$ and the particular size of the data. The utilization of suitable optimization methods like genetic algorithms could lead to stronger distinguishers also for other tested functions.

Note that once the universal distinguisher for a tested function is found, the application on data produced by this function is straightforward and requires only small amount of data produced. The Table 3 compares the *BoolTest* performance for tested PRNGs with standard test suites.

Table 3: Results of NIST STS (NI), Dieharder (Di), Test U01 (U01) and *BoolTest* obtained for 1MB, 10MB and 100MB of data. ∀ means all tests passed, fraction means number of tests passed from the total number. *BoolTest* column represents an average of $|$*Z-score*$|$ values produced by the best distinguishers $\{f_1, f_2, f_3, f_4, f_5, f_6\}$ on 1000 randomly seeded input bit streams.

size	func	NI	Di	U01	*BoolTest*
1MB	c	-	∀	∀	19.67
	java	-	∀	∀	17.78
10MB	c	∀	∀	∀	60.92
	java	∀	∀	∀	55.98
100MB	c	∀	22/23	∀	191.37
	java	∀	∀	15/16	176.62

5 The statistical interpretation

The result of *BoolTest* is the maximal *Z-score* computed within a set of boolean functions. The interpretation of *Z-score* for a single boolean function is simple and straightforward. *Z-score* is normally distributed random variable and *p-value* can be computed directly from it. However, the computation of *p-value* from the maximal *Z-score* (*Z-SCORE*) computed by our tool is more complicated. In this Section, we describe the *Z-score*, the *p-value* and the statistical theory related to our approach. Afterward, we discuss interpretations of the result of *BoolTest* based on reference results computed for random data.

5.1 *P-value* and *Z-score*

The *p-value* represents the probability that more extreme results are obtained (for the true hypothesis) than we observed (s_{obs}). In our case, *p-value* represents the probability that a perfect random number generator would produce less random sequences than the sequence being tested. *P-value* is computed from the observed test statistic s_{obs} and the reference distribution D or its close approximation. The null distribution of many tests is binomial distribution $B(n,p)$. It is approximated well (for $n > 10p$ and $n.(1-p) > 10$) by normal distribution $\mathcal{N}(\mu,\sigma^2)$ (Wackerly et al., 2002). Normal distribution is symmetric around mean μ and therefore *p-value* is computed as an area under bell curve in both tails (see Figure 3). Sometimes *Z-score* is computed instead of *p-value* since they are related

$$p\text{-}value = erfc\left(\frac{Z\text{-}score}{\sqrt{2}}\right)$$

(Chevillard, 2012) and computation of *Z-score* is simpler and faster. The *Z-score* represents the distance from the mean μ in units of σ. The binomial distribution $B(n,p)$ is approximated by $\mathcal{N}(\mu,\sigma^2)$, with the parameters $\mu = np$ and $\sigma^2 = np(1-p)$ (Sheskin, 2003) i.e. *Z-score* of binomially ($B(n,p)$) distributed #1 is computed as

$$Z\text{-}score = \frac{\#1 - pn}{\sqrt{p(1-p)n}} = \frac{\#1-\mu}{\sigma}.$$

Table 4: The mean (μ) of maximal *Z-SCORE* computed by *BoolTest* for various settings $k, deg \in \{1,2,3\}, m \in \{128,256,384,512\}$ and $t = 128$.

deg	1			2			3		
m \ k	1	2	3	1	2	3	1	2	3
128	2.84	3.89	4.73	3.97	5.26	6.57	4.74	5.76	7.61
256	3.01	4.23	5.16	3.92	5.66	7.11	4.78	6.12	8.29
384	3.14	4.42	5.32	3.92	5.82	7.44	4.79	6.22	8.58
512	3.24	4.56	5.52	3.98	6.02	7.68	4.83	6.30	8.81

5.2 Maximal Z-score

In order to interpret results (*Z-SCORE*) of our tool we have to find their distribution for the null hypothesis i.e. random data. The value of *Z-SCORE* is determined by the boolean functions constructed in two phases. Final *Z-SCORE* is computed as a maximal *Z-score* within a set of boolean functions constructed in the second phase according to used setting deg, m, k, t. The theoretical assessment of the null distribution of *Z-SCORE* (for our tool) is difficult task in general, since it should follow the process of the construction of best distinguisher. The problem can be illustrated on the simplest combination of parameters $deg = 1$ and $k \in \{1,2,3,\cdots\}$. Distribution D of *Z-SCORE* for $deg = k = 1$ can be obtained as distribution of maximum of t independent normal variables. Function $erfc(x)$ represents the cumulative distribution function (cdf) of normal variable $\mathcal{N}(0,0.5)$. For t independent variables with the same cdf $erfc(x)$, the cdf of their maximum (i.e. *Z-SCORE*) is the function $erfc(x)^t$ which is determined only by t. Distribution D can be computed as the derivation $t.erfc(x)^{t-1}.erfc'(x)$ with $erfc'(x)$ to be a probability density function of nomal variable $\sim \mathcal{N}(0,0.5)$. Due to exponent t it is clear that D has smaller variance than $\mathcal{N}(0,0.5)$ i.e. values are closer to mean which is still hard to express. For $deg = 1$ and $k = 2$ the boolean functions are dependent since some of them share monomials. This problem grows with the increasing of deg, k, t. The simplest solution is to find D using the simulation on random numbers. We performed experiment to compute the mean and the standard deviation to identify the reference distribution of *Z-SCORE* for random data. We tested data produced by AES with 10 rounds for selected settings $k, deg \in \{1,2,3\}, m \in \{128,256,384,512\}$ and $t = 128$. We firstly confirmed that value of *Z-SCORE* does not depend on size of data analysed. Than we tested all combination of parameters and computed for each setting 100 maximal *Z-scores*. The value of the mean for each setting is shown in the Table 4. We also computed the standard deviation for all settings. In all cases the standard deviation was smaller than 0.4. To interpret the result one can use mean μ (corresponding to used parameters) from the table and "normalize" the *Z-SCORE*. Our normalized *Z-SCORE* has smaller standard deviation than 0.4 and therefore *p-value* can be bounded by:

$$p\text{-}value \leq erfc\left(\frac{Z\text{-}SCORE-\mu}{\sqrt{2}}\right).$$

6 Conclusion

The paper introduces a new class of tests for statistical detection of non-randomness in an output of cryptographic functions and random generators. The test called *BoolTest* is based on boolean functions. Our approach was evaluated on more than 20 real world cryptographic functions with results comparable to commonly used statistical batteries like NIST STS or Dieharder. The test runs significantly faster than mentioned batteries and usually requires order of magnitude less data.

Additionally, the bias spotted is directly interpretable as a relation between several fixed output bits of the analyzed function. The *BoolTest* can be used as a fast alternative to existing batteries and/or to complement its results. The direct interpretability of a boolean function based distinguisher adds benefit for human cryptologist interested in the more detailed analysis of weakness present in a inspected cryptographic function.

The future work will address boolean functions of higher degree and with more components. The brute-force examination of boolean functions used in this work can be practically performed only for the degree up 3 and with no possibility to evaluate all combination of two or more components. The adaptive learning methods may be used for selection of viable candidates for combination from the initial pool of all possible simple functions.

Acknowledgements: We acknowledge the support of the Czech Science Foundation, project GA16-08565S. The access to the computing and storage resources of National Grid Infrastructure MetaCentrum (LM2010005) is greatly appreciated.

REFERENCES

Bard, G. V. (2009). *Algebraic Cryptanalysis*. Springer Publishing Company, ISBN 978-0-387-88756-2.

Brown, R. G., Eddelbuettel, D., and Bauer, D. (2013). Dieharder: A random number test suite 3.31.1. `http://www.phy.duke.edu/~rgb/General/dieharder.php`.

Chevillard, S. (2012). The functions Erf and Erfc computed with arbitrary precision and explicit error bounds. In *Academic Press, Inc., Information and Computation*, volume 216, pages 72–95. Academic Press, Inc.

EACirc (2017). EACirc project. `https://github.com/CRoCS_MUNI/EACirc`.

Englund, H., Johansson, T., and Sönmez Turan, M. (2007). A framework for chosen IV statistical analysis of stream ciphers. In *INDOCRYPT 2007*, pages 268–281. Springer Berlin Heidelberg.

Filiol, E. (2002). A new statistical testing for symmetric ciphers and hash functions. In *ICICS 2002*, pages 342–353. Springer Berlin Heidelberg.

Garrett, A., Hamilton, J., and Dozier, G. (2007). A comparison of genetic algorithm techniques for the cryptanalysis of TEA. In *International journal of intelligent control and systems*, volume 12, pages 325–330. Springer.

Hernández, J. and Isasi, P. (2004). Finding efficient distinguishers for cryptographic mappings, with an application to the block cipher TEA. In *Computational Intelligence*, volume 20, pages 517–525. Blackwell.

Heys, H. M. (2002). A tutorial on linear and differential cryptanalysis. In *Cryptologia*, volume 26, pages 189–221, Bristol, PA, USA. Taylor & Francis, Inc.

Kaminsky, A. and Sorrell, J. (2014). Cryptostat, a bayesian statistical testing framework for block ciphers and macs. `http://www.cs.rit.edu/~ark/students/jls6190/report.pdf`.

L'Ecuyer, P. and Simard, R. (2007). TestU01: A C library for empirical testing of random number generators. In *ACM Trans. Math. Softw.*, volume 33, New York, NY, USA. ACM.

Marsaglia, G. (1995). The Marsaglia random number CDROM including the diehard battery of tests of randomness. `http://www.stat.fsu.edu/pub/diehard/`.

Mouha, N. (2010). Ecrypt II: Tools for cryptography. `http://www.ecrypt.eu.org/tools/overview`.

NIST (2001). FIPS 140-2 security requirements for cryptographic modules. NIST.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). Numerical recipes 3rd edition: The art of scientific computing. Cambridge University Press.

Rukhin, A. (2010). A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications, version STS-2.1. NIST.

Sheskin, D. J. (2003). Handbook of parametric and nonparametric statistical procedures. CRC Press, USA.

Stankovski, P. (2010). Greedy distinguishers and nonrandomness detectors. In *INDOCRYPT 2010, LNCS 6498*. Springer.

Sýs, M. and Klinec, D. (2017). Booltest – tool for fast randomness testing. `http://crocs.fi.muni.cz/papers/secrypt2017`.

Sýs, M., Švenda, P., Ukrop, M., and Matyáš, V. (2014). Constructing empirical tests of randomness. In *SECRYPT 2014*. ICETE.

Wackerly, D. D., III, W. M., and Scheaffer, R. L. (2002). Mathematical statistics with applications. Duxbury Advanced Series.