# BoolTest: The fast randomness testing strategy based on boolean functions with application to DES, 3-DES, MD5, MD6 and SHA-256

Marek Sýs, Dušan Klinec, Karel Kubíček, and Petr Švenda

Masaryk University, Brno, Czech Republic
{syso,ph4r05,karel.kubicek}@mail.muni.cz, svenda@fi.muni.cz

**Abstract.** The output of modern cryptographic primitives like pseudorandom generators and block or stream ciphers is frequently required to be indistinguishable from a truly random data. The existence of any distinguisher provides a hint about the insufficient confusion and diffusion property of an analyzed function. In addition to targeted cryptoanalysis, statistical tests included in batteries such as NIST STS, Dieharder or TestU01 are frequently used to assess the indistinguishability property. However, 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. We propose a simple, yet surprisingly powerful method called *BoolTest* for the construction of distinguishers based on an exhaustive search for boolean function(s). The *BoolTest* typically constructs distinguisher with fewer input data required and directly identifies the function's biased output bits. We analyze the performance on four input generation strategies: counter-based, low hamming weight, plaintext-ciphertext block combination and bit-flips to test strict avalanche criterion. The *BoolTest* detects bias and thus constructs distinguisher in a significantly higher number of rounds in the round-reduced versions of DES, 3-DES, MD5, MD6 and SHA-256 functions than the state-of-the-art batteries. Finally, we provide a precise interpretation of *BoolTest* verdict (provided in the form of *Z-score*) about the confidence of a distinguisher found. The *BoolTest* clear interpretation is a significant advantage over standard batteries consisting of multiple tests, where not only a statistical significance of a single test but also aggregated decision over multiple, potentially correlated tests, needs to be correctly performed.

**Keywords:** statistical randomness testing; hypothesis testing, boolean function

## 1 Introduction

Both newly designed as well as widely used cryptographic primitives (block cipher, stream cipher, hash function, pseudo-random generators, etc.)[1] are subjected to various analysis techniques like linear, differential and algebraic cryptanalysis which looks for flaws or information leakage in the primitive design. The standard techniques try to

[1] This is extended version of the paper 'The Efficient Randomness Testing using Boolean Functions' (Sýs et al., 2017, SeCrypt)

find any significant correlations between the tested primitive input (plaintext), output (ciphertext) and key bits (if used). The existence of correlated bits indicates a weakness of the function, which might be exploitable to predict bits of a secret key or next output bits of the pseudorandom generator. 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 as a quick and cheap tool before the deeper cryptanalysis is performed (Simion, 2015). Commonly, well-designed crypto-primitives should produce output with the same characteristics as truly 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 property from its specific point of view. 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. Although there is an unlimited number of tests in principle, batteries opt for implementation of only several selected ones for the practical reasons. The randomness in such a context is a probabilistic property, and we can commit two types of errors – Type I (truly random data rejected) and Type II (non-random data not rejected).

The batteries implement many tests of various complexity – from the very simple Monobit computing statistic of bits (frequency of ones and zeros) to the very complex statistics computed from large blocks (e.g., computation of linear profile). The complexity of tests usually determines the amount of data necessary to compute the histograms for comparison. In order to decrease the Type I and II errors, sufficiently many data sequences (up to several GBs of data) are required in practice.

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

1. **An insufficient strength to detect bias in unweakened functions** – 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 standard security parameters.
2. **An insufficient detection sensitivity if only a small amount of data is available** – The tests might be too insensitive to detect biases when an only limited amount of data is available for the testing. The tests usually require from 10 MB up to several GBs of data which may not be available in particular test scenario.
3. **The difficulty of test results interpretation** – The interpretation of test results is often only generic in the form of "something is wrong with the provided data". Only a few tests are able to identify concrete dependent bits and provide this crucial information to a 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

The *BoolTest* implementation and paper supplementary material can be found at https://crocs.fi.muni.cz/papers/booltest2018.

amount of data yet directly identifying the biased output bits. In fact, we are looking for the strongest distinguisher possible (of cryptographic function from the truly random data) within the given amount of tested 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 also be 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 with an easy interpretation whether a robust distinguisher for tested data and function was found (or not). The standard batteries are notoriously difficult to interpret as both the results of a single test as well as multiple, potentially cross-correlated tests needs to be properly reasoned about.
- **Interpretable test for small data:** We have shown that *BoolTest* not only requires significantly less 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.
- **Large number of function analyzed:** The *BoolTest* sensitivity was tested on common and widely used cryptographic functions like AES or SHA-3 (with over 20 functions tested total), all with a gradually reduced complexity via the decreased number of internal rounds. The sensitivity of *BoolTest* is mostly equal to the state-of-the-art batteries like TestU01 when tested on 100 MB data streams with some notable differences. The counter-based input generation strategy results in more internal rounds still distinguishable by *BoolTest* for DES, 3-DES, Keccak, MD5, MD6, SHA-1, SHA-256, and TEA functions when no bias is detected anymore by the standard batteries. Conversely, the TestU01 battery is able to distinguish a higher number of rounds in DES, MD5, MD6, and SHA-1 functions when the strict-avalanche testing input generation strategy is used.
- **A different strategies for input data generation examined:** A tested cryptographic function is first repeatedly executed to produce a sufficiently long output data stream, which is then supplied for the testing. However, the properties of the inputs supplied to the tested function during the output generation are crucial and significantly influence the randomness properties of generated output. The four different input data generation strategies were tested: 1) counter-based, where input block is in the form of incremental counter, 2) very low hamming weight block, 3) random input block with a corresponding pair formed by single bit flip (focused on the analysis of strict avalanche criterion) and 4) random input block, but also inserted into the output data stream to test for input/output correlation.
- **Practical distinguisher for *C/Java rand*:** Among others, we found previously unknown biases in the output of *C rand()* and *Java Random* pseudo-random genera-

tors 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: Section 2 describes principles of the commonly used batteries and provides the motivation for more efficient tests. 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 and other functions. Section 5 is devoted to the statistical interpretation of results reported by *BoolTest*. Section 6 surveys the previous work and is followed by the conclusions given in Section 7.

## 2 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., a 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 100 MB data produced by truly random number generator (which should pass all tests), divide it into 128-bit 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 TestU01 battery detects the bias. Moreover, for 0.1 % none of the standard tests (batteries NIST STS, Dieharder and TestU01) detect this – still significant – bias. The problem lays in a structure of patterns the tests are searching for in tested data.

Dieharder and NIST STS batteries analyze 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 analyzes 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. Such representation 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 test 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 analyzed sequence or a new sequence:

- Maximal observed test statistic can be directly used to assess the randomness of the 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 also be 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 also allows to speed up the entire process since many intermediate computational results (for simpler functions) can be reused.

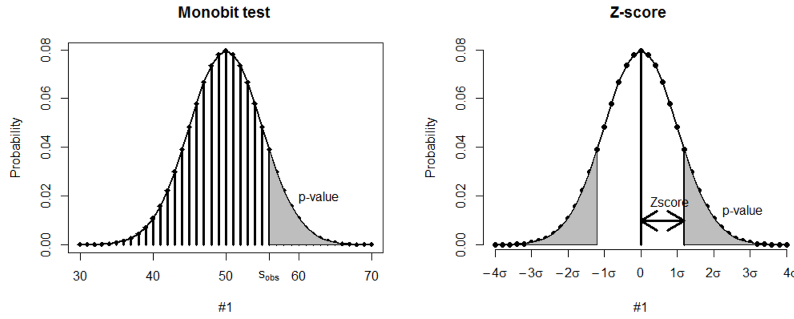
### 3.1 Test of randomness

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 analyzed 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 $\chi^2$ 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 analyzed feature) than the tested sequence (Rukhin, 2010). The *p-value* is compared with the significance level $\alpha$ typically set to smaller values $0.01, 0.005$ or $0.001$ for the randomness testing. If the *p-value* is smaller/bigger than $\alpha$ hypothesis is rejected/accepted and data are considered to be non-random/random. The following example illustrates how *p-value* is computed for the 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 the Monobit test as:

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

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 1a 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 1a illustrates the value of *p-value* for $n = 100$ and $s_{obs} = 56$.




(a) 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 (Sýs et al., 2017).

(b) The relation of *Z-score* and *p-value*. *Z-score* is expressed in the units of the standard deviation (Sýs et al., 2017).

Fig. 1: Monomial bit test and Z-score

### 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 more extreme 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:

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

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 1b illustrates the relation of a *Z-score* (standardly expressed in the units of standard deviation $x \cdot \sigma$) and the corresponding *p-value* (area of two tails).

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

The symbol $p$ denotes the probability that the result of boolean function $f$ is equal to 1 for random input. The symbol $n$ denotes the number of non-overlapping blocks (of $m$ bits) in the analyzed 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 2 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 into 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).

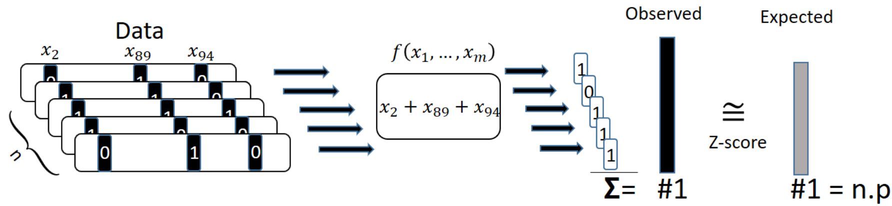



Fig. 2: Our approach and the computation of *Z-score* using boolean function $f(x_1, \cdots, x_m) = x_2 + x_{89} + x_{94}$. *Z-score* is computed as the statistical distance of observed #1 for tested data and $\#1 = p.n$ expected for truly random data (Sýs et al., 2017).

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 typical 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 exponentially 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 improve the entire performance significantly.

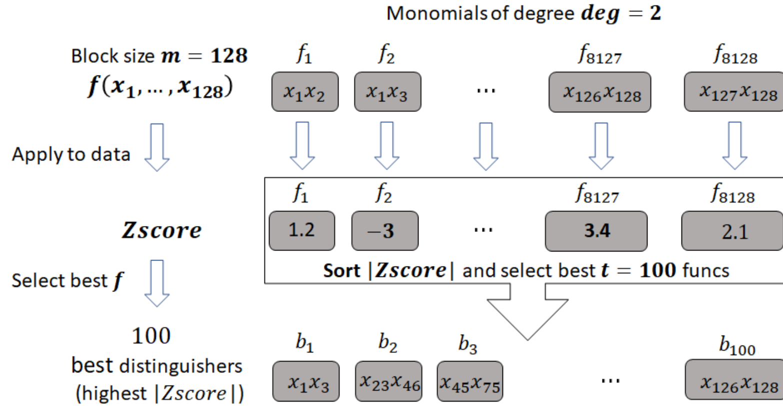



Fig. 3: Illustration of the first phase of distinguisher construction. The phase is parametrized by parameters $m, deg, t$. Input to this phase is data (fixed) to be analyzed. All Boolean functions in the first (and also in the second) phase are defined over $m$ variables ($m = 128$ here). Each monomial (out of $\binom{m}{deg}$) monomials) of degree $deg$ ($deg = 2$ here) is applied to data and corresponding $Z-score$ is computed. Then absolute values of $Z-score$s are computed and sorted. The result of the first phase is set of $t$ ($t = 100$) strongest distinguishers – Boolean functions $b_i$ (monomials) with highest absolute values of $Z-score$.

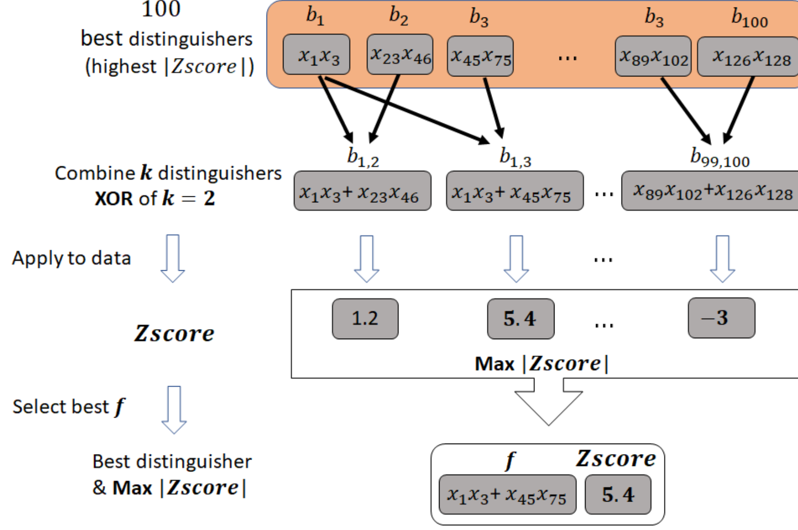



Fig. 4: Distinguisher construction – second phase. Second phase is parametrized by parameters $m$, $k$. The inputs to this phase are: the same data (as in previous phase) to be analyzed, and the set $t$ of best distinguishers $b_i$ found in the first phase. In this phase new Boolean functions are constructed as sum ($XOR$) of $k$ Boolean functions $b_i$. All constructed B.f. are applied to data corresponding $Z-score$ is computed. The result of this phase and of entire approach is the function $f$ – distinguisher, and the value – $Z-score$. The $Z-score$ can be used to assess the randomness of analyzed data.

### 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

$$X_f = (X_1 \text{ AND } X_2) \text{ XOR } X_3$$

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 $x_i$, $x_i \cdot x_j$ are fully pre-computed 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. $P(x_i) = 0.5$
2. $P(x_i x_{i+1} \cdots x_{i+k-1}) = p_i p_{i+1} \cdots p_{i+k-1} = 2^{-k}$
3. $P(x_i + x_j) = p_i(1 - p_j) + (1 - p_i)p_j$
4. $P(x_i + x_j + x_k) = P((x_i + x_j) + x_k)$ using associativity and the rule 3.

By 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.

Table 1: Experimental execution time of *BoolTest* running on Python 3.6.4, Intel Xeon CPU E5-2630 v3 2.40. *BoolTest* configuration is: block bit size - degree - term combination degree

| data \ conf | 256-2-3 | 256-3-3 | 512-2-3 | 512-3-3 |
|---|---|---|---|---|
| 10 MB | 23 s | 1 m 12 s | 21.1 s | 3 m 29 s |
| 100 MB | 2 m 40 s | 9 m 53 s | 1 m 42 s | 31 m 30 s |

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 2 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 3 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 3, 4 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 2, 3, 4.

In practice, we use polynomials and terms of a relatively small degree, so we do not 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 $O(n\,\alpha(n))$. Practical running times of the overall BoolTest are stated in Table 1.

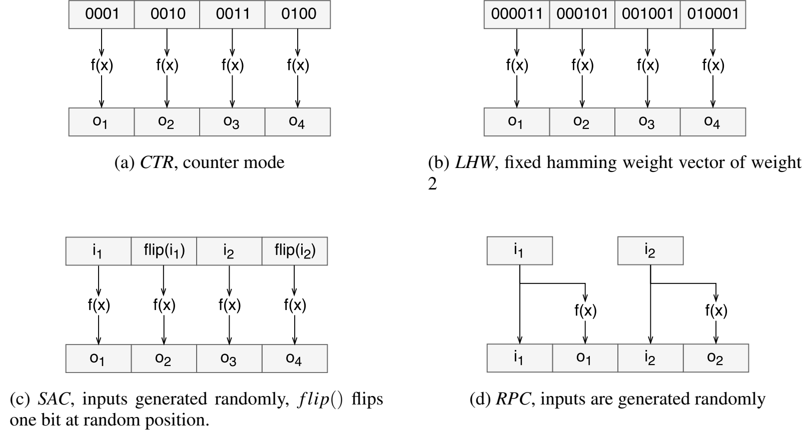
## 4 Testing methodology and results

To demonstrate the practical usability of the 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 are generated using four different strategies.

### 4.1 Preparation of data for testing

Several different data generation strategies were used to analyze target function output confusion and diffusion properties, namely *CTR*, *LHW*, *SAC* and *RPC* as shown in Figure 5 and explained below. With the given strategy, the 100 MB of output data are generated and used as an input for the randomness testing battery.

The *CTR* strategy generates blocks of particular size each containing the current block index. Intuitively the high bits are set to zero while the low bits are iterating until the 100 MBs of data is generated.




(a) *CTR*, counter mode

(b) *LHW*, fixed hamming weight vector of weight 2

(c) *SAC*, inputs generated randomly, $flip()$ flips one bit at random position.

(d) *RPC*, inputs are generated randomly

Fig. 5: Data generation modes and formation of output stream for testing.

The *LHW* stands for low Hamming weight as it generates function's input blocks with the fixed and low Hamming weight. The weight is derived from the block size as it is required to avoid cycling of the generator, i.e., depleting of all options on the block size. If tested function $f$ has input block size of 128 bits, the Hamming weight is set to 4, because $16\binom{128}{4} \approx 170\,\text{MB}$. For 64-bit input block, the minimal required Hamming weight is 6, as $8\binom{64}{6} \approx 600\,\text{MB}$. The idea behind the *LHW* strategy is to cover the whole input block with small changes only, keeping the total Hamming weight low thus feeding the minimal possible entropy to a function. Both *CTR* and *LHW* serve as low-entropy input generators.

The *SAC* strategy aims to test the Strict Avalanche Criterion. It generates pairs of blocks where the first block in the pair is randomly generated and the second one is almost the same except for single bit flip at a randomly selected position. Both blocks are then used as an input to tested function $f$.

The *RPC* strategy stands for random-plaintext-ciphertext and generates random input block $p_i$, which is an input to a tested function $f$. The resulting data block used for statistical analysis is then $p_i||f(p_i)$, the concatenation of plaintext and ciphertext. This particular testing method adds additional entropy to the tested function making the detection more difficult, e.g., it is expected that number of function's internal rounds with still detectable bias would be lower when compared to low-entropy inputs such as *CTR* and *LHW*. On the other hand, we can directly analyze function's input-output correlation.

### 4.2 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 $deg \in \{1,2,3\}$, $m \in \{128,256,384,512\}$, $t = 128$, and $k \in \{1,2,3\}$.

### 4.3 Common cryptographic functions with CTR strategy

In order to compare our results with the standard batteries, we tested the data generated with *CTR* strategy both with *BoolTest* and NIST STS, Dieharder and TestU01 test suites (Alphabit, BlockAlphabit, Rabbit, Small Crush). Table 2 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 100 MB of data. We consider tested data to be rejected by a battery if at least one test from the battery fails with the conservative significance level set to $\alpha = 1\,\%$.

Table 2: The number of rounds (of selected primitives and PRNGs) in which non-randomness was detected for 100 MB 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 |
| Grøstl | 2 | 2 | 2 | 2 | 2 |
| Hamsi | 0 | 0 | 0 | 0 | 0 |

| function | NI | Di | U01 | Bool1 | Bool2 |
|---|---|---|---|---|---|
| JH | 6 | 6 | 6 | 6 | 6 |
| Keccak | 2 | 2 | 2 | 3 | 3 |
| Lex | 3 | 3 | 3 | 3 | 3 |
| Lesamta | 2 | 3 | 3 | 2 | 2 |
| Luffa | 7 | 7 | 7 | 7 | 7 |
| MD6 | 8 | 8 | 8 | 9 | 8 |
| Simd | 0 | 0 | 0 | 0 | 0 |
| Salsa20 | 6 | 4 | 6 | 4+ | 4+ |
| TEA | 4 | 4 | 4 | 4+ | 4+ |
| TSC-4 | 13 | 12 | 13 | 13+ | 13+ |
| Twister | 6 | 6 | 7 | 6 | 6 |

Table 2 shows the best results of our tool obtained for two particular *BoolTest* settings: $Bool1(\, deg = 2, k = 1, m = 384)$ and $Bool2(deg = 2, k = 2, m = 512)$. In 15 out of 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, and DCH, Decim, Grain, Salsa20 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 Salsa20 with

4 rounds, TEA reduced to 5 rounds, and TSC-4 reduced to 14 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 10 MB, 100 MB, and 1 GB of input data. The results are summarized in Table 3. For test suites, the number of passed tests are shown.

Table 3: 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 10 MB, 100 MB and 1 GB of data produced with primitives with limited number of rounds (Sýs et al., 2017).

| size | func | NI | Di | U01 | Bool3 | Bool4 |
|---|---|---|---|---|---|---|
| 10 MB | AES (3) | ∀ | 18 | 15 | 8.6 | 6.7 |
| | TEA (4) | ∀ | 20 | ∀ | 20.6 | 11.5 |
| | Keccak (3) | ∀ | ∀ | 15 | 3.7 | 5.3 |
| | MD6 (9) | ∀ | ∀ | ∀ | 3.9 | 13.3 |
| | SHA-256 (3) | 0 | 0 | 6 | 88.7 | 242 |
| 100 MB | 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 |
| | MD6 (9) | ∀ | ∀ | ∀ | 3.7 | 26.4 |
| | SHA-256 (3) | 0 | 0 | 4 | 50.7 | 828 |
| 1 GB | AES (3) | 9 | 18 | 14 | 12.8 | 41.2 |
| | TEA (4) | 13 | 24 | ∀ | 127 | 4.3 |
| | Keccak (3) | ∀ | 26 | 15 | 3.5 | 32.0 |
| | MD6 (9) | 13 | 25 | 15 | 4.1 | 26.4 |
| | SHA-256 (3) | 0 | 1 | 3 | 78.0 | 3043 |

The computed *Z-scores* are shown for the *BoolTest* and two best settings according to given a set of analyzed functions. The results of *BoolTest* and test suites which can be interpreted as detected non-randomness (null 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.

### 4.4 Comparison of data generation strategies on selected functions

The four different data generation strategies described in Section 4.1 and their impact on the randomness properties of tested function output is shown in Table 4. The notable results are as follows:

- MD5 hash function reduced to 18 rounds and with *LWH* data generation mode as tested by TestU01. The test `smarsa_BirthdaySpacings` from TestU01 Small Crush consistently evaluates the input as non-random. The test's *p-value* increases with increasing number of rounds up to 18, but it is always significant ($\leq 10^{-9}$).

Table 4: The number of rounds (of selected primitives) in which non-randomness was detected for 100 MB data for NIST STS (NI), Dieharder (Di) and TestU01 (U01) compared to our approach *BoolTest* (BT). The input generation is described in Subsection 4.1. Gray highlight the best result for given function on every data generation strategy.

| Scenario | *CTR* | | | | *LHW* | | | | *SAC* | | | | *RPC* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fun.\Tests | NI | Di | U01 | BT | NI | Di | U01 | BT | NI | Di | U01 | BT | NI | Di | U01 | BT |
| AES | 3 | 3 | 3 | 3 | 2 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | - | 1 | 1 | 1 |
| Blowfish | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 2 | 2 | 3 | 3 | - | 1 | 1 | 1 |
| DES | 4 | 4 | 4 | 5 | 4 | 4 | 4 | 5 | 4 | 4 | 5 | 4 | 1 | 1 | 2 | 4 |
| 3-DES | 2 | 2 | 2 | 3 | 2 | 2 | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 2 |
| Grøstl | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | - | - | - | - | - | - | - | - |
| JH | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 5 | 2 | 2 | 2 | 3 |
| Keccak | 2 | 2 | 2 | 3 | 2 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 1 | - | 1 | 1 |
| MD5 | 9 | 10 | 9 | 11 | 12 | 13 | 20 | 13 | 9 | 11 | 14 | 12 | 3 | 3 | 4 | 6 |
| MD6 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 9 | 7 | 7 | 8 | 7 | 5 | 5 | 7 | 5 |
| SHA-1 | 12 | 12 | 13 | 14 | 16 | 16 | 16 | 16 | 11 | 15 | 16 | 14 | 4 | 4 | 5 | 7 |
| SHA-256 | 6 | 6 | 6 | 7 | 12 | 12 | 12 | 13 | 11 | 11 | 12 | 13 | 3 | 4 | 4 | 4 |
| TEA | 4 | 4 | 4 | 5 | 3 | 3 | 3 | 4 | 3 | 4 | 3 | 3 | - | 2 | 1 | 1 |

The test `snpair_ClosePairsBitMatch` from TestU01 Rabbit fails with even more extreme *p-values*. This test fails even for MD6 reduced to 20 out of total 64 rounds (*p-values* $\leq 10^{-19}$).

- Among the input generation strategies, *RPC* is consistently the scenario which is the most difficult to distinguish from the truly random stream. The *SAC* is more difficult than *CTR* and *LHW*, which are roughly comparable. However, for SHA-256 the *CTR* scenario is more difficult than both *LHW* and *SAC*. We hypothesize that *CTR* difficulty is caused by more chaotic bit-flips within two consecutive blocks compared to other scenarios.
- *BoolTest* is among the most successful tests for *CTR*, *LHW* and *RPC* inputs for tested data with 100 MB length. For *SAC* generation strategy, TestU01 is the most sensitive battery, while *BoolTest* performs similarly to Dieharder battery.

Note, that the interpretation of *BoolTest* result (if the tested sequence is random or non-random) is more straightforward than for the standard batteries. While *BoolTest* consists of only a single test and resulting single *Z-score*, standard batteries consist of multiple statistical tests, each with own *p-value* interpretation and also potentially correlated to other tests. This property was also confirmed while comparing the results shown in the Table 4.

### 4.5 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 $\tau_i$ and the best distinguisher and its corresponding *Z-score* value for $\tau_i$ returned by *BoolTest* as $(\xi_i, \delta_i)$. Figures 6 and 7 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 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.

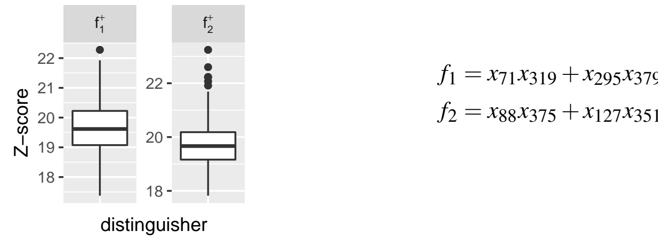
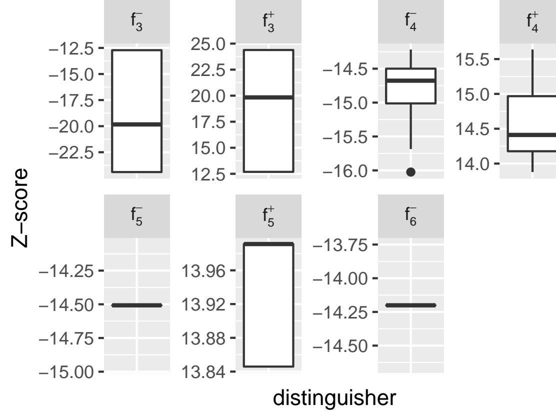



Fig. 6: The best distinguishers, *C rand()*, 1000 x 1 MB data samples, 384-bit block, random 32-bit seed, Ubuntu 16.04. The best distinguisher occurrences in 1000 tests: $|f_1^+| = 520$, $|f_2^+| = 480$

The distinguishers from Figure 7 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 in Figure 5.

It is evident there exists a good distinguisher of a low degree 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

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




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

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

Fig. 7: 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, (Sýs et al., 2017).

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. Table 5 compares the *BoolTest* performance for tested PRNGs with standard test suites.

Table 5: Results of NIST STS (NI), Dieharder (Di), Test U01 (U01) and *BoolTest* obtained for 1 MB, 10 MB and 100 MB 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* |
|---|---|---|---|---|---|
| 1 MB | c | - | ∀ | ∀ | 19.67 |
| | java | - | ∀ | ∀ | 17.78 |
| 10 MB | c | ∀ | ∀ | ∀ | 60.92 |
| | java | ∀ | ∀ | ∀ | 55.98 |
| 100 MB | c | ∀ | 22/23 | ∀ | 191.37 |
| | java | ∀ | ∀ | 15/16 | 176.62 |

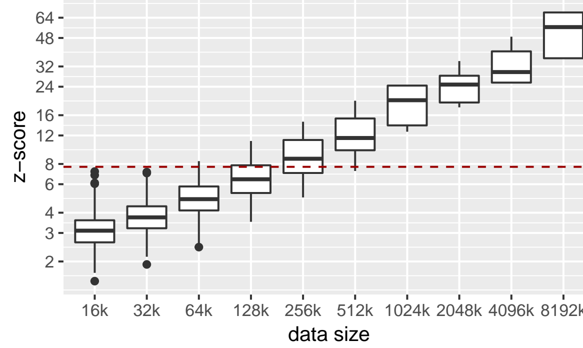



Fig. 8: The size of the input bit stream vs. $|Z\text{-}score|$ using distinguishers $\{f_3, f_4, f_5, f_6\}$ for *Java Random*, 1000 random seed samples per data size category. Dashed line represents ref. *Z-score* value for the test, (Sýs et al., 2017).

## 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* (denoted as *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 truly 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. The *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 $n \cdot (1-p) > 10$) by normal distribution $\mathcal{N}(\mu, \sigma^2)$ (Wackerly et al., 2002). Normal distribution is symmetric around mean $\mu$ and therefore *p-value* is computed as an area under bell curve in both tails (see Figure 1b). Sometimes *Z-score* is computed instead of *p-value* since they are related

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

(Chevillard, 2012) and computation of *Z-score* is simpler and faster. The *Z-score* represents the distance from the mean $\mu$ in units of $\sigma$. 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 6: The mean ($\mu$) 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 | | |
|---|---|---|---|---|---|---|---|---|---|
| $m \setminus k$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 128 | 2.80 | 3.96 | 4.81 | 3.87 | 5.35 | 5.80 | 4.71 | 6.56 | 7.61 |
| 256 | 3.05 | 4.00 | 4.81 | 4.23 | 5.66 | 6.01 | 5.13 | 7.10 | 8.24 |
| 384 | 3.14 | 4.01 | 4.77 | 4.39 | 5.86 | 6.19 | 5.32 | 7.41 | 8.55 |
| 512 | 3.24 | 3.92 | 4.79 | 4.55 | 6.02 | 6.28 | 5.54 | 7.64 | 8.86 |

Table 7: The standard deviation ($\sigma$) 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 | | |
|---|---|---|---|---|---|---|---|---|---|
| $m \setminus k$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 128 | 0.39 | 0.25 | 0.25 | 0.33 | 0.30 | 0.32 | 0.27 | 0.23 | 0.27 |
| 256 | 0.42 | 0.30 | 0.24 | 0.36 | 0.31 | 0.26 | 0.28 | 0.23 | 0.26 |
| 384 | 0.37 | 0.31 | 0.25 | 0.26 | 0.27 | 0.26 | 0.25 | 0.22 | 0.23 |
| 512 | 0.37 | 0.30 | 0.23 | 0.31 | 0.30 | 0.27 | 0.28 | 0.20 | 0.31 |

### 5.2 Maximal Z-score

In order to interpret result (*Z-SCORE*) of our tool we have to find expected distribution $f_{max}$ of *Z-SCORE* for the null hypothesis i.e. random data. The value of *Z-SCORE* is determined by the boolean functions constructed in two phases using setting $deg$, $m$, $k$, $t$. The value *Z-SCORE* is computed as a maximal *Z-score* for a set of *Z-score*s corresponding to constructed boolean functions. The theoretical assessment of the distribution $f_{max}$ should follow the process of the construction of the best distinguisher with maximal *Z-score* (in absolute value). It is hard to theoretically derive probability density function (pdf) $f_{max}$ for settings with $k > 1$. But for the settings with $k = 1$ the pdf $f_{max}$ can be derived much easier.

- For $k = 1$ the resulted *Z-SCORE* is equal to a maximal *Z-score* computed for boolean functions constructed in the first phase, since in the second phase no functions are constructed. While $Z-score$s corresponding to boolean functions (monomials $x_i$ for $i \in 1,\cdots,m$) of degree $deg = 1$ are independent normally distributed

random variables, the $Z-scores$ of functions of higher degrees are dependent since these functions share variables e.g. $x_1.x_2$ and $x_1.x_3$. The correlation is not significant and variables can be modeled as independent with small bias when compared to empirically obtained $f_{max}$.

- In the case of $k > 1$ the situation is quite different. First of all, *Z-scores* for the best $t$ distinguishers selected in the first phase are normally distributed but with different means and standard deviations. It is clear that the pdf of a maximum of $t$ normal variables is different to pdf of the second largest normal variable. The reason is obvious: the probability $P(\max(X_i) = V)$ that maximum ($l$-th order statistic) of normal variables $X_1, \cdots, X_l$ is equal to very large value is strictly smaller than probability that second maximal value (within $V_i$) is equal to $V$. Second of all dependency of boolean functions constructed in the second phase is bigger, more complex since these functions are constructed as a sum of several (two or three) partially dependent functions. Therefore to derive theoretical expected pdf $f_{max}$ for $k > 1$ is hard task in general.

In the case $k = 1$, the theoretical pdf $f_{max}$ can be derived using the fact that pdf $f_{max}$ of largest $X$ ($l$-th order statistic) for independent and identically random variables $X_1, \cdots, X_l$ with pdf $f$ is given by

$$f_{max}(x) = l \cdot f(x) \cdot F(x)^{l-1},$$

for the cumulative distribution function $F$ of variables $X_i$. We are looking for $f_{max}$ for maximum of absolute values of *Z-scores* hence $f(x) = 2 \cdot \varphi(x)$ and $F(x) = 2 \cdot (\Phi(x) - 0.5)$ for standard normal distribution $\varphi(x)$ and corresponding cdf $\Phi(x)$. The pdf $f_{max}$ of *Z-SCORE* for $k = 1$ can expressed as:

$$f_{max}(x) = l \cdot \varphi(x) \cdot (2 \cdot \Phi(x) - 1)^{l-1}$$

for $l = \binom{m}{deg}$ representing number of boolean functions constructed in the first phase. We computed functions $f_{max}(x)$ for settings $k, deg \in \{1,2,3\}$, $m \in \{128,256,384,512\}$ and realized that function $f_{max}$ represent normal distribution with mean $\mu$ very close to average $Z-score$ obtained empirically.

Table 8 shows theoretical $\mu$ computed in statistical software R.

Table 8: The mean ($\mu$) of maximal *Z-SCORE* computed by *BoolTest* and reference theoretical computation in statistical software R (R-soft) for various settings $k = 1$, $deg \in \{1,2,3\}$, $m \in \{128,256,384,512\}$.

| *deg* | 1 | | 2 | | 3 | |
|---|---|---|---|---|---|---|
| *m* | *BoolTest* | R-soft | *BoolTest* | R-soft | *BoolTest* | R-soft |
| 128 | 2.80 | 2.70 | 3.87 | 3.85 | 4.71 | 4.68 |
| 256 | 3.05 | 2.92 | 4.23 | 4.18 | 5.13 | 5.09 |
| 384 | 3.14 | 3.03 | 4.39 | 4.36 | 5.32 | 5.32 |
| 512 | 3.24 | 3.12 | 4.55 | 4.48 | 5.54 | 5.48 |

## 6 Related work

NIST STS (Rukhin, 2010), Dieharder (Brown et al., 2013) (an extended version of the Diehard (Marsaglia, 1995)) 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 out of total 15 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. Overall, Dieharder consists of 76 test variants. Dieharder is generally more powerful than NIST STS with the ability to detect smaller biases as it may analyze longer data stream.

TestU01 can be viewed as the 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 six 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]. The fastest is Small Crush (10 tests), significantly slower is Crush (96 tests) and very slow but powerful is Big Crush battery (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 206 MB/2.6 GB/51.3 GB data to run all tests of the battery. Other three batteries are proposed for testing the binary sequences specifically. The Rabbit (26 tests), Alphabit (9 tests) and BlockAlphabit (54 tests) batteries are not limited in fact (Rabbit is restricted to 500 bits) in the size of data they need for the analysis. Other statistical testing tools we are aware of are: Donald Knuth's tests (Knuth, 1969), Crypt-X suite (Caelli et al., 1998), PractRand (Doty-Humphrey, 2014), RaBiGeTe (Piras, 2004), CryptoStat (Kaminsky and Sorrell, 2013), YAARX (Biryukov and Velichkov, 2014), ENT (Walker, 2008), SPRNG (Mascagni and Srinivasan, 2000), gjrand (Jones, 2007) and BSI's test suite (Schindler and Killmann, 2002).

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 the 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 the number of monomials is exponential in the number of variables, the randomness is evaluated based on the number of monomials of degree exactly $d$ which has $\chi^2$ 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 the truly random data. This is completely different to our approach, where the 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.

Hernández and Isasi proposed an automated construction of distinguisher for TEA block cipher (Hernández and Isasi, 2004). They searched for a distinguisher in the form of input bitmask of the 192-bit block (64-bit plaintext and 128-bit key). 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 Hernández's approach with a quantum-inspired genetic algorithm and found distinguisher for TEA limited to 5 rounds. *BoolTest* is performing same as shown in Table 4, yet the scenarios are not directly comparable.

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.

## 7 Conclusion

This paper provides deeper analysis of newly proposed statistical testing tool called *BoolTest* (Sýs et al., 2017, SeCrypt). In the former work, authors focused on randomness testing of an output of cryptographic functions and random generators. In this work, we present three new strategies focusing on the correlation of input and output bits of cryptoprimitives. To compare the new strategies to the former one, we analyzed the same functions, and we extended the testbed with six new cryptoprimitives (Blowfish, DES, 3-DES, MD5, SHA-1, and SHA-256).

The new results show that *BoolTest* is filling the gap among most common statistical batteries like NIST STS, Dieharder, and TestU01. For *CTR* and *RPC* strategies, *BoolTest* is on average capable of detecting bias in one additional round. It performs comparable for *LHW* strategy and slightly worse in *SAC* strategy.

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 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 an inspected cryptographic function.

The future work will address speeding up the brute-force part of the computation by utilizing the FPGA and smarter selection of terms using a heuristic.

**Acknowledgements:** We acknowledge the support of the Czech Science Foundation, project GA16-08565S. Computational resources were provided by the CESNET LM2015042 and the CERIT Scientific Cloud LM2015085, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures."

# Bibliography


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

[Biryukov and Velichkov, 2014] Biryukov, A. and Velichkov, V. (2014). Automatic search for differential trails in arx ciphers. In *Cryptographers' Track at the RSA Conference*, pages 227–250. Springer.

[Brown et al., 2013] 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`.

[Caelli et al., 1998] Caelli, W. et al. (1998). Crypt-X suite.

[Chevillard, 2012] 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.

[Doty-Humphrey, 2014] Doty-Humphrey, C. (2014). Practically random: Specific tests in practrand.

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

[Englund et al., 2007] 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, 2002] Filiol, E. (2002). A new statistical testing for symmetric ciphers and hash functions. In *ICICS 2002*, pages 342–353. Springer Berlin Heidelberg.

[Garrett et al., 2007] 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 and Isasi, 2004] 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, 2002] Heys, H. M. (2002). A tutorial on linear and differential cryptanalysis. In *Cryptologia*, volume 26, pages 189–221, Bristol, PA, USA. Taylor & Francis, Inc.

[Jones, 2007] Jones, G. (2007). gjrand random numbers.

[Kaminsky and Sorrell, 2013] Kaminsky, A. and Sorrell, J. (2013). Cryptostat: a bayesian statistical testing framework for block ciphers and macs. *Rochester Institute of Technology, Rochester, NY*.

[Kaminsky and Sorrell, 2014] 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`.

[Knuth, 1969] Knuth, D. E. (1969). *The Art of Computer Programming*, volume 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition.

[L'Ecuyer and Simard, 2007] 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, 1995] Marsaglia, G. (1995). Diehard: a battery of tests of randomness.

[Mascagni and Srinivasan, 2000] Mascagni, M. and Srinivasan, A. (2000). Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):436–461.

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

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

[Piras, 2004] Piras, C. (2004). RaBiGeTe documentation.

[Press et al., 2007] 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, 2010] 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.

[Schindler and Killmann, 2002] Schindler, W. and Killmann, W. (2002). Evaluation criteria for true (physical) random number generators used in cryptographic applications. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 431–449. Springer.

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

[Simion, 2015] Simion, E. (2015). The relevance of statistical tests in cryptography. *IEEE Security & Privacy*, 13(1):66–70.

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

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

[Sýs et al., 2017] Sýs, M., Klinec, D., and Švenda, P. (2017). The efficient randomness testing using boolean functions. In *14th International Conference on Security and Cryptography (Secrypt 2017)*, pages 92–103. SCITEPRESS.

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

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

[Walker, 2008] Walker, J. (2008). Ent: A pseudorandom number sequence test program.