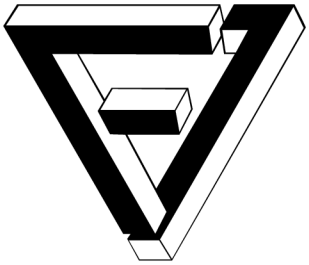
Masaryk University
Faculty of Informatics



BoolTest and Grey-box Randomness Testing

Master's Thesis

Jiří Gavenda

Brno, Autumn 2023

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jiří Gavenda

Advisor: Mgr. Marek Sýs, Ph.D.

Acknowledgements

My gratitude goes to my thesis supervisor, Marek Sýs, for his guidance and never-ending suggestions on how to improve the text of this thesis. I appreciate my family and friends for their emotional support during the painful process of writing. Last but not least, I need to thank my girlfriend for her unwavering support and understanding.

Abstract

BoolTest is a fast, configurable randomness testing tool. This thesis aims to improve upon BoolTest by taking into account the properties of sources of presumably random data. We design a new randomness distinguisher construction based on an approximation of the examined source of data. This construction identified a strong distinguisher for the pseudorandom number generator Xorshift64. We then explore what properties good distinguishers usually have. Based on our observations, we design and implement a new randomness testing tool, CoolTest. We compare CoolTest to BoolTest and standard randomness test suits on the outputs of 12 cryptographic functions. We show that on the outputs of the 12 functions, CoolTest produces significantly better results than BoolTest and is at least comparable to the standard test suits. Moreover, we present other versions of CoolTest, which can find non-random patterns that the main version of the tool can not find. Besides that, we propose a data-preprocessing method that helps both BoolTest and CoolTest find stronger distinguishers.

Keywords

distinguisher, BoolTest, CoolTest, randomness testing, hypothesis testing

Contents

Introduction 1

1 Statistical Randomness Testing 3
1.1 Statistical Hypothesis Testing . . . 3
1.2 Distinguishers . . . 4
1.3 BoolTest . . . 6
1.3.1 Interpretation of Results . . . 7

2 Functions and Inputs 8
2.1 Properties of Cryptographic Functions . . . 9
2.1.1 Balancedness . . . 9
2.1.2 Correlation Immunity . . . 10
2.1.3 Strict Avalanche Criterion . . . 10
2.1.4 Nonlinearity . . . 10
2.2 Choice of Inputs . . . 10
2.3 Generating of Simple Cryptographic Functions . . . 13

3 Approximated Distinguishers 15
3.1 Function Approximation . . . 16
3.1.1 Black-box Approximation . . . 16
3.1.2 Round-by-round Approximation . . . 17
3.2 Elimination of Low-degree Monomials . . . 19
3.3 Evaluation . . . 19
3.3.1 Xorshift . . . 20

4 Distinguisher Analysis 23
4.1 Random Sampling . . . 23
4.2 Evaluation of All Distinguishers . . . 24

5 CoolTest 28
5.1 Distinguisher Construction . . . 28
5.1.1 Phase 1 – Brute-force Search . . . 29
5.1.2 Phase 2 – Greedy Search . . . 30
5.1.3 Phase 3 – Distinguisher Strengthening . . . 31

5.2 Early Stopping 32
5.2.1 Validation 33
5.2.2 Simple Conditions 33
5.3 Implementation 34
5.4 Evaluation 35
5.4.1 Xorshift 38
5.5 Other Versions 39
5.5.1 `OR` Version 40
5.5.2 Polynomial Version 41
5.5.3 Histogram Version 41

6 Future Work **44**

Conclusion **46**

A Electronic Attachments **51**

B Xorshift Polynomials **52**

C Evaluation of All Distinguishers – 4 Bits **53**

D Generation of Data **55**

Introduction

Digital security heavily relies on hash functions, block ciphers and other cryptographic primitives. While browsing the internet, using payment cards, or even unlocking cars, these functions help protect our privacy and secure our assets. It is often necessary that the outputs of these cryptographic functions are indistinguishable from random data. If an attacker could find some patterns in the outputs that distinguish them from random data, they could use it to attack the cryptographic primitive and the systems using it. Randomness is essential in other aspects of cryptographic functions and protocols, too. Random data are used as nonces, keys, challenges, initialisation vectors in block cipher modes, and many other parts of cryptographic protocols.

It is extremely important to ensure that functions intended to generate random data do so reliably. To test these functions, a series of outputs is generated and then statistically analysed to determine if the sequence looks random. Over time, many different randomness tests were developed, from the simplest one that counts the number of ones and zeroes to more complicated tests, for example, one that simulates 200 000 games of Craps. These statistical tests are collected into statistical test suits. However, these test suits have some drawbacks. For some of the tests, tens of MBs up to several GBs of data are needed in order to detect a bias in the data. Moreover, most of the tests are too complicated and even though the results indicate that the data are not random, we can not easily conclude which bits are biased and what relation exists between them. Sýs et al. presented a different approach that attempts to solve these problems. The statistical tests used by their tool, BoolTest, are based on Boolean functions that are easy to interpret, i.e. the Boolean function gives us exact bits and a relation between them that was discovered in the data. Moreover, it can detect a bias even on small amounts of data for which some of the tests from the test suites can not be used.

BoolTest works fully in a black-box manner without taking into consideration the specific properties of the outputs of tested functions. In this work, we aim to improve upon BoolTest by analysing the relation between the source of data and tests that perform well on the data. Our goal is to, based on the analysis of the relation, propose and implement an alterna-

tive construction of the Boolean functions that can be used for statistical randomness testing.

In the first chapter, we start by explaining how statistical tests can be used to determine whether some data are random. Then, we explain how BoolTest works and how to interpret its results.

The second chapter is dedicated to functions that are supposed to produce random-looking data and their properties – balancedness, correlation immunity, strict avalanche criterion, and nonlinearity. We explain how the properties of these functions affect whether we can find non-random patterns in the data they produce. Moreover, we discuss the importance of correctly choosing strategies for generating inputs for such functions, e.g. generating random inputs, consecutive values, inputs with low hamming weight, etc. The choice of inputs influences whether we can find correlations between output bits and what kind of correlations we discover. Besides that, we design a simple construction of permutation functions. We use them while developing methods for function approximation and to generate data for the evaluation of distinguishers (randomness tests) that we construct.

The third chapter presents a new approach to distinguisher construction based on an approximation of the function whose outputs we test. We evaluate this method on the permutations from the second chapter and the pseudorandom number generator Xorshift64. To better understand the relationship between functions and the suitability of a given statistical test to capture the correlations in the function's outputs, we conduct two experiments. These experiments are presented in the fourth chapter. We use observations from these experiments to design a new BoolTest-like tool.

This tool, called CoolTest, is presented in the fifth chapter. We describe our approach to the construction of statistical tests. Then, we evaluate the tool on 12 cryptographic functions and compare it to BoolTest and standard statistical batteries. At the end of the chapter, we describe how CoolTest can be modified to find non-random patterns that can not be discovered by the original version of the tool.

Finally, we devote the sixth chapter to future work. We present how some of CoolTest's weaknesses that we discovered during its evaluation could be mitigated and how the tool could be further improved.

Chapter 1

Statistical Randomness Testing

This chapter starts with a short introduction to statistical hypothesis testing. We explain that the statistical tests result in a *p-value*. If the *p-value* is sufficiently low, it indicates that the examined data are not random. Alternatively, if the computation of *p-value* is too demanding and our data are normally distributed, we can use *Z-score* as a faster alternative.

The statistical tests that we use are called distinguishers, as they distinguish between random and artificially made data. We define distinguishers more precisely later in this chapter, together with an explanation of different approaches to randomness testing. Finally, we describe how BoolTest constructs its distinguishers and discuss how the results of such a tool can be interpreted.

1.1 Statistical Hypothesis Testing

During statistical hypothesis testing [1], we examine a hypothesis called a *null hypothesis*. In the case of randomness testing, the hypothesis is that the source of data that we are testing produces random data. This *null hypothesis* is rejected if we have enough evidence for the opposite statement, called an *alternative hypothesis*. Otherwise, we do not reject the *null hypothesis*, and the hypothesis either holds or a better test is needed in order to reject the hypothesis.

Since statistical hypothesis testing is based on just a sample of data that the source can produce, there is always a possibility of an error. *Type I error* occurs when we reject a *null hypothesis* even though it is true. When the *null hypothesis* is false, but we do not reject it, it is called a *type II error*.

We do not have any simple means to directly limit the *type II error*, i.e. calling non-random data random. The best we can do is to improve the statistical tests so that they capture the differences between random and artificially made data as well as possible.

On the other hand, the probability of making a *type I error* is entirely under our control, and we set it for each experiment in advance. We denote this probability α, and it is often referred to as a *significance level*. Assuming that the *null hypothesis* holds and we observe some result, the probability of obtaining such or a more extreme result is referred to as a *p-value*. If we observe a low *p-value*, either an unlikely event occurred and a source of random data produced such an extreme sample of data, or likely the source does not produce random data. Provided that a *p-value* is lower than the *significance level*, we reject the *null hypothesis*.

The type of statistical test used in this thesis is the binomial test [2], i.e. the data collected during our randomness tests are expected to have a binomial distribution $B(n, p)$. The calculation of a *p-value* can be computationally demanding, especially if we need to compute it for many statistical tests at once. To avoid this computation, we usually approximate the binomial distribution as a continuous normal distribution $\mathcal{N}(\mu, \sigma^2)$. Normal distribution approximates the binomial distribution well if $n > 9\frac{max(p,1-p)}{min(p,1-p)}$ [3]. In the normal distribution, we can use *Z-score* to evaluate the tests. If needed, the *Z-score* can be transformed into a *p-value* using a lookup table. However, this *p-value* is only an approximation of the *p-value* in the original binomial test. The *Z-score* is computed as follows:

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

where k denotes the number of positive outcomes obtained during an experiment consisting of n sample inputs, and p is the probability of obtaining a positive outcome for a random input.

For example, we could test how many non-overlapping pairs of bits in a sequence are equal to 00. The expected probability of a pair of bits to be equal to 00 is $p = 0.25$. Then, k is the actual number of 00s in the sequence, and n is the number of pairs of bits in the sequence.

1.2 Distinguishers

In this section, we describe randomness testing as a distinguishing attack. We explain how we use the term distinguisher and what kind of distinguishers we use in this work. Then, we present different approaches to distinguisher construction to provide context for the subsequent description of the randomness-testing tool BoolTest.

Randomness testing can be described as a distinguishing attack [4]. In a randomness-distinguishing attack, an attacker obtains data either from a true random number generator or from a function mimicking such a generator. The attacker then tries to correctly guess the source of the data. This attacker is often referred to as a *distinguisher*. The distinguishing attack

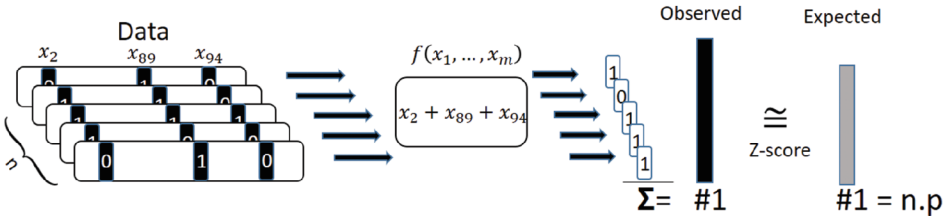


Figure 1.1: Example of the process of evaluation of the whole statistical test with the distinguisher f by BoolTest taken from [5].

is successful if a distinguisher has a non-negligible advantage over random guessing.

In this work, we use the same type of distinguishers as BoolTest does. These distinguishers are statistical tests performed on a sequence of data, as shown in Figure 1.1. They work as follows:

1. Data are divided into blocks of a given length, e.g. 128 bits.
2. Then, a Boolean function (distinguisher) is applied on each block of data and the total number of 1s returned is counted.
3. Based on the expected number of 1s ($n \cdot p$) that the Boolean function would return for random data, a *p-value* or *Z-score* is computed as a result of our test.
4. If the *p-value* (*Z-score*) is significantly low (high in the absolute value), i.e. lower than the *significance level*, it indicates that the data are not random.

As the only difference among such distinguishers is the Boolean function whose expected distribution we compare to the observed one, in this thesis, we use the term distinguisher for the Boolean function and not for the whole statistical test. Throughout this work, we call a distinguisher (a Boolean function) *good* for a given cryptographic function if it can be used to detect a significant bias (e.g. *p-value* lower than $\alpha = 10^{-4}$) on the outputs of the function.

There are many approaches to statistical randomness testing. Standard statistical batteries contain many statistical tests that can be used on any (large enough) data from any source. These statistical tests are constructed so that they work for as many sources of non-random data as possible or that they complement the other tests from the battery well, so at least some of the tests fail for non-random data. Among the most commonly used test suits are NIST STS [6], Dieharder [7], and TestU01 [8]. A different approach was used in [9], where cryptanalytic methods led to a discovery that the second output byte of the RC4 stream cipher has the value 0 twice as often as

expected for random data. They use this knowledge to distinguish between random data and data produced by the stream cipher.

A randomness test can also be constructed specifically for data from a given cryptographic function. In 2022, Brunetta and Picazo-Sanchez described how to use machine learning techniques to distinguish between random data and data produced by a given cryptographic function [10]. Another example of statistical test construction based on the knowledge of the data is BoolTest, presented by Sýs et al. [5]. We describe this approach in the following subsection.

1.3 BoolTest

BoolTest [5] is a configurable tool for distinguishing between non-random and random data that allows a straightforward interpretation of the discovered patterns. It constructs distinguishers in the form of Boolean functions using the operations AND and XOR. Therefore, throughout this work, we write Boolean functions in algebraic normal form (ANF) as this representation uses the two logical operations. We call such functions Boolean polynomials or just polynomials and write them as such. The logical AND is represented as multiplication of variables and XOR as addition, e.g. we write $(x_1 \,\mathsf{AND}\, x_2) \,\mathsf{XOR}\, x_3$ as $x_1x_2 + x_3$.

BoolTest first constructs monomials using the AND operation and then sums them together using XOR. The two steps can be described as follows:

1. First, all single-monomial polynomials of a given degree are constructed. It means if we fix the degree to be 3, the tested polynomials look as follows: $x_0x_1x_2$, $x_0x_1x_3$, ... Each polynomial is evaluated on every block of the provided data and the number of blocks for which the polynomial returns 1 is then used to compute their *Z-score*. A provided number of these polynomials with the highest absolute value of their *Z-score* is then chosen and used in the second step.

2. In the second step, BoolTest takes all subsets of size k of the polynomials obtained from the first step. These subsets are then summed to obtain new polynomials. If we set $k = 2$, such a polynomial could, for example, look like this: $x_1x_4x_7 + x_2x_{11}x_{15}$. The *Z-score* is then computed for these new polynomials based on the expected and observed number of 1s.

From the polynomials evaluated in the second step, we choose the one with the highest absolute value of *Z-score* as the final distinguisher. If the *Z-score* of this distinguisher is sufficiently large, BoolTest found a significant pattern in the data and thus indicates that the data are most likely not random. The polynomial also clearly shows which output bits are correlated and their relation.

1.3.1 Interpretation of Results

An important question when using tools like BoolTest is how to interpret the results of the tool properly. As the randomness test computes the *Z-score* for multiple distinguishers and chooses the one with the highest score, the *Z-score* of the final distinguisher can not be used directly. The issue here is that even on random data, if we try a large enough number of independent distinguishers, we will eventually find one with a high *Z-score*, as the *Z-scores* are normally distributed. The distinguishers tested by BoolTest are not independent as some of them share the same variables, but the maximal *Z-score* still grows with the number of tested distinguishers. Moreover, due to the dependencies between distinguishers, the maximal *Z-score* can not be easily transformed into a *p-value* of the whole experiment. BoolTest offers two solutions to this problem.

The first solution is based on empirically obtaining the reference distribution of statistics of the maximal *Z-scores* for a given setting of BoolTest. It is done by running the tool repeatedly with the setting on a large enough number of randomly generated sequences of data. Then, based on the results and chosen *significance level*, we define the interval of *Z-score* outside which the randomness hypothesis should be rejected. The disadvantage of this approach is that it can be used only with the tool settings for which we have precomputed the reference distribution. Still, if we have that, it is straightforward without any other limitations.

The other approach is to split the data into a training and a testing sequence. In the context of BoolTest, this technique is called halving, as the two sequences are exactly halves of the original data. When the data is divided, the distinguisher is found using the first half of the data. Once we have the distinguisher, we simply evaluate it on the second half of the data and directly compute the *p-value*. Since we expect the data to be random, there should be no connection between the first and the second half. Therefore, as we evaluate only one distinguisher, the *p-value* can be directly used without any transformation of the value. The disadvantage of this approach is that we need more data to achieve the same results as the first approach. On the other hand, we can use BoolTest with any setting without the need to previously run it on large quantities of random data.

Chapter 2

Functions and Inputs

In this chapter, we discuss how the properties of cryptographic functions and inputs provided to the functions influence the possibility of finding a good distinguisher. First, we introduce some essential properties of cryptographic functions, focusing mainly on hash functions and block ciphers. Then, we explain the importance of choosing the right inputs for a function so that we can spot the relations among the output bits of the function. It is crucial because, with some inputs, it is virtually impossible to find any significant pattern in the outputs, even for strongly biased functions. Finally, we present a way of generating simple functions that work similarly to usual cryptographic functions. We later use them to analyse the relationship between cryptographic functions and good distinguishers.

It is essential to understand that cryptographic or any other functions can be viewed from many different perspectives. Usually, we consider them to be some kind of black box or a sequence of operations. In this thesis, it is sometimes helpful to understand them as a vector of Boolean functions. This way, we can think separately about any output bit. But at the same time we can see more complicated relations among these bits, e.g. we can easily spot that some set of bits is not linearly independent. To stress that the polynomial representation is more suitable in some parts of the text, we use the term *vectorial Boolean function*.

With the notion of vectorial Boolean functions, we can revise the model of processing one block of data by the vectorial function and the distinguisher and introduce the notation used in this and the following chapter. Figure 2.1 shows an example of the process with the use of the notation. We denote $X \in \mathcal{F}_2^n$ the input vector consisting of Boolean variables (bits) $x_0...x_{n-1}$. This input is then processed by a vectorial function $F(X) : \mathcal{F}_2^n \to \mathcal{F}_2^m$, which consists of Boolean functions $F = (f_0, ..., f_{m-1})$. Usually, we use functions F with the same input and output size, i.e. $n = m$. The output of F is then called Y and consists of m bits, where each bit y_i is the output of f_i, i.e. $y_i = f_i(x_0, ..., x_{n-1})$. Finally, we apply the distinguisher $d(Y) : \mathcal{F}_2^m \to \mathcal{F}_2$

on the output and receive 0 or 1, which is then added up with results from other blocks and used for the computation of a *Z-score*.

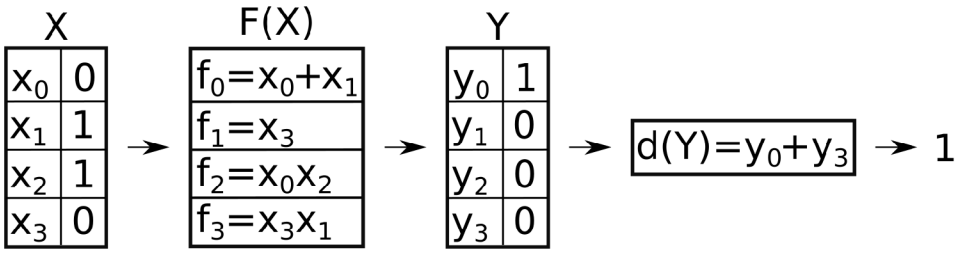


Figure 2.1: Example of processing of one block. The block is first fed into a vectorial Boolean function F, and its output is then evaluated using a distinguisher d.

2.1 Properties of Cryptographic Functions

In this thesis, we focus mainly on hash functions and block ciphers, but all the properties we cover are general properties that can be found in many different types of functions. At first glance, hash functions and block ciphers seem entirely different. However, there are some things that actually make them really similar. We can even use a block cipher to create a hash function [11].

With both these classes of functions, we would like if an attacker, when provided with an output, had no clue what the function's input looked like. For this, it is also necessary that similar inputs do not result in similar outputs. Moreover, it should be difficult to approximate such a function using low-degree polynomials as this could lead to known cryptanalytic attacks. Any feasible attempt of an approximation should yield only a negligible advantage over random guessing to the attacker.

We use these properties to argue why some choices of inputs are better than others. Moreover, the permutation functions that we construct at the end of this chapter ideally should have these properties. Now, let us look at these properties in more detail.

2.1.1 Balancedness

A Boolean function is balanced [12] if it returns 1 for exactly half of all possible input values. A vectorial Boolean function $F = (f_1, f_2, ..., f_m) : \mathbb{F}_2^n \to \mathbb{F}_2^m$ is balanced, if every linear combination of functions from $f_1, f_2, ..., f_m$ is balanced. It means that $\forall x, y : |\{i \in \mathbb{F}_2^n | F(i) = x\}| = |\{i \in \mathbb{F}_2^n | F(i) = y\}|$.

Cryptographic functions need this property because if some outputs are returned more often than others, an attacker could have a non-negligible advantage when guessing output for a new input. Block ciphers need to

have this property by definition, as for a fixed key, they need to map every input vector to exactly one output vector of the same size.

2.1.2 Correlation Immunity

A function is k-th degree correlation immune [13] if any of its output bits is statistically independent of any subset of input variables of size k.

This property is important since, if there is a noticeable correlation between the input and output values, an attacker can find this correlation and predict output bits for a new input with a higher than random success rate.

2.1.3 Strict Avalanche Criterion

If a function meets the *Strict Avalanche Criterion* [14], then if we change any bit of an input, every output bit has a 50% chance of being changed.

This property is especially interesting in our setting. If a function does not satisfy this property, then we can create many similar inputs for which the outputs have something in common. A distinguisher can then find this common pattern among the outputs.

2.1.4 Nonlinearity

The nonlinearity [15] of a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is defined as a value and denoted as $\mathcal{N}_f$. It is computed as $\mathcal{N}_f = \min_{\phi \in \mathcal{A}^n} \Delta(f, \phi)$, where $\mathcal{A}^n$ is the set of all affine functions on $\mathbb{F}_2^n$, and Δ denotes a distance defined as the number of distinct inputs for which the output of the Boolean function and the affine function are different.

If a cryptographic function has a low nonlinearity, it can be approximated well by a linear function, which can be used for linear cryptanalysis [16] or fast correlation attack [17]. Generally, when constructing a cryptographic function, the goal is that the Boolean functions have as high nonlinearity as possible. Ideally, when represented as a vector of Boolean functions, cryptographic functions should consist of maximally nonlinear functions, so-called bent functions [18]. Unfortunately, bent functions lack other vital criteria, e.g. they are never balanced and their algebraic degree is limited [19].

2.2 Choice of Inputs

The non-random patterns that can be discovered in the outputs of a cryptographic function do not depend only on the cryptographic function itself. Another key factor is the input data provided to the cryptographic function to obtain the tested outputs.

Let us illustrate this with a simple example. Consider the simplest possible vectorial Boolean function on 4 variables – the identity function,

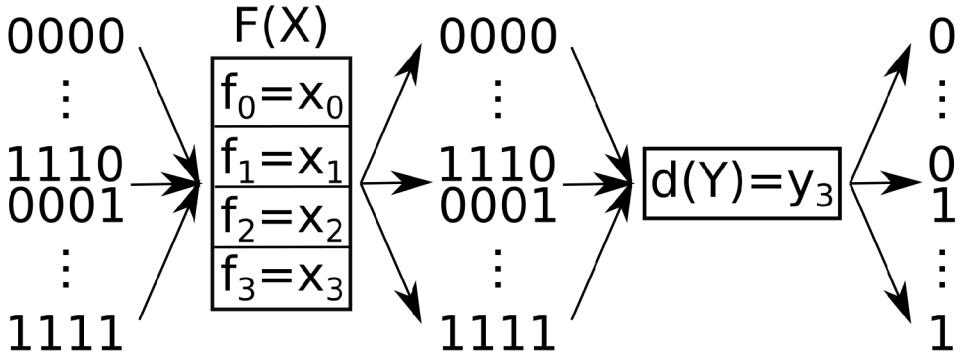


Figure 2.2: Example of evaluation of all inputs of size 4 by an identity function and then by a distinguisher y_3.

i.e. $F(x_0, x_1, x_2, x_3) = (x_0, x_1, x_2, x_3) = (y_0, y_1, y_2, y_3)$. Intuitively, it would seem that this function is so simple that we should be able to find a good distinguisher without any issues with any input data. Let us consider what happens if we use all 16 possible input vectors to evaluate the distinguisher. As shown in Figure 2.2, we receive a set of all 16 outputs from the vectorial function. There is no surprise since the identity function just returned what it obtained. However, if we take any distinguisher (e.g. $d = y_3$), we will find out that the *Z-score* for this distinguisher is 0 since when provided with all possible values exactly once the distinguisher necessarily returns the expected number of 1s. Almost the same would happen if the input data were generated randomly. Any significant result would be just a coincidence. Completely different result can be obtained if we use just the lower half of all inputs (0000...1110). The inputs have the last bit always set to 0 and the same holds for the outputs. With a distinguisher $d = y_3$, we can obtain a *Z-score* of 2.8, which indicates a significant bias.

In this example, balancedness is the reason why no distinguisher could find an abnormality when all possible inputs are used. On the other hand, the success with inputs 0000...1110 can be attributed both to strong input-output correlation and the absence of an avalanche effect.

In [5], Sýs et al. tested four different data generation strategies - CTR, LHW, SAC and RPC. The CTR strategy generates blocks that are binary representations of consecutive numbers from 0 to some predefined value based on the total number of blocks we want to generate. The LHW strategy is used to generate values with fixed low Hamming weight. As its name suggests, the SAC strategy primarily tests the Strict Avalanche Criterion. This strategy generates pairs of blocks, where the first block is the function's output for a random input, and the second one is the output for the same input with just one bit flipped. The fourth strategy - RPC, aims to test the correlation immunity of a cryptographic function. It generates pairs

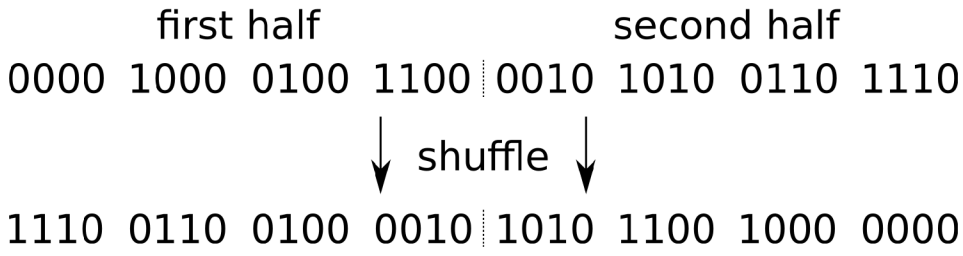


Figure 2.3: Example of shuffling of inputs generated using the CTR strategy.

of blocks, where the first block is randomly generated, and the second one is the function's output for the first block.

While the SAC and RPC strategies aim for specific types of issues in the cryptographic functions, the CRT and LHW strategies should be more general and potentially point out more types of biases at once.

When using data generated by these two strategies, the bias discovered by tools like BoolTest can be altered by changing the block size on which the tool looks for distinguishers. While using a block size the same as the output length of the cryptographic function, we can find patterns caused by the similarity of all the provided inputs. On the other hand, when using block size, which is double (or more) the output size, we can spot correlations between consecutive outputs of a cryptographic function which were generated using very similar input vectors, usually differing just in a few bit flips.

However, CTR and LHW might not work so well when we use BoolTest with the evaluation method set to halving. The issue is that the inputs from one half might have slightly different similarities than those from the other half. Similarly, consecutive blocks in the first half could have a bit different correlations than in the second half. We propose that the blocks (or pairs of blocks, etc.) are randomly shuffled. This way, we can decrease the potential differences between the two halves and help BoolTest to find more general distinguishers working better with the halving evaluation method. Figure 2.3 shows an example of a shuffling of inputs generated using the CTR strategy. Notice that in the first half of the original inputs, the value of the third bit is always 0, while in the second half, it is always 1. Moreover, the average hamming weight in the first half is 1, while in the second, it is 2. Shuffling the data in the example helps to mitigate these issues. While it is true that random shuffling does not guarantee that the two halves do not significantly differ, in practice, this technique produces good results, as shown in Chapter 5.

2.3 Generating of Simple Cryptographic Functions

To test methods in the following chapters and better understand the relation between cryptographic functions and their good distinguishers, we decided to create our own simple cryptographic functions.

As cryptographic functions usually work in rounds, we design a function working in rounds as well. This approach also has the advantage that we can gradually adjust the function to make it more and more difficult to distinguish its outputs from random data. As the simplest case for us is a function that maps inputs to outputs of the same length, we decided to use block permutations as these simple cryptographic functions. However, there are $2^n!$ permutations, on the space of binary vectors of size n. So, if we wanted to represent arbitrary permutation, it would be possible only for really small inputs as there are approximately $5.4 \cdot 10^{2639}$ permutations already for an input size 10.

We use a simple construction similar to a substitution-permutation network (SPN) [20]. The main difference between SPN and our construction is that we omit a `XOR` with a round key, as for a construction of a permutation, this is unnecessary. One round of the cryptographic function consists of permutation of the input bits and then applying an S-box to consecutive blocks, as shown in Figure 2.4. These S-boxes are permutations on small blocks, and therefore, any such permutation can be represented. Our implementation allows us to represent them either as vectors of Boolean polynomials or as a vector of all possible blocks of the given length.

Permutations of bits and S-boxes complement each other well. With only the permutation of the bits, the final permutations would be only linear.

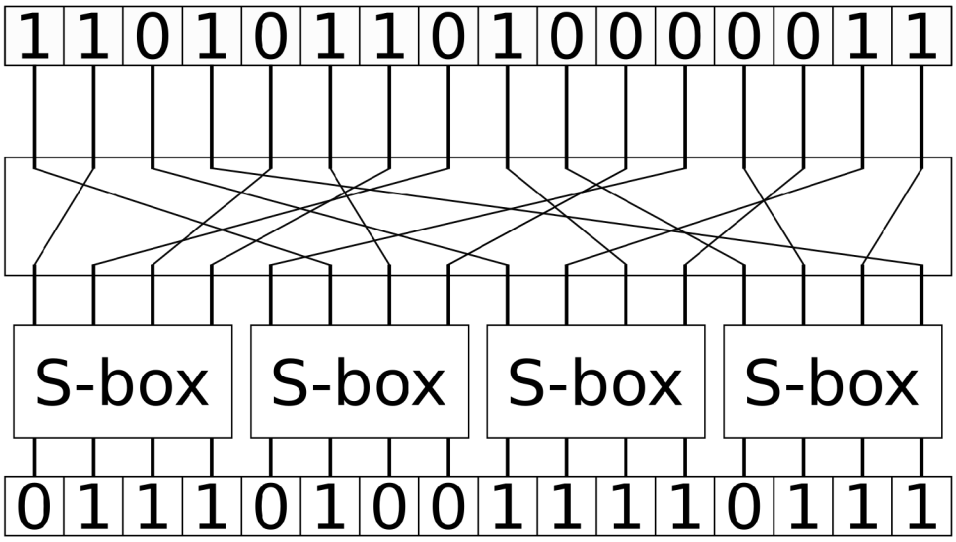


Figure 2.4: One round of our simple permutation function

Table 2.1: *p-values* of distinguishers found by BoolTest with setting 64-2-2 on data produced by 10 randomly generated permutations with various numbers of rounds. The value 0 indicates any *p-value* $< 10^{-300}$ (i.e. *Z-score* > 37).

rnds	perm0	perm1	perm2	perm3	perm4	perm5	perm6	perm7	perm8	perm9
1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	$3 \cdot 10^{-248}$	0	0	0	0	0	0	0	0
7	0	$4 \cdot 10^{-38}$	10^{-192}	0	0	0	0	0	$4 \cdot 10^{-293}$	$7 \cdot 10^{-141}$
8	$5 \cdot 10^{-89}$	$5 \cdot 10^{-4}$	$2 \cdot 10^{-18}$	0	0	0	0	0	$8 \cdot 10^{-83}$	$4 \cdot 10^{-16}$
10	$9 \cdot 10^{-1}$	10^{-1}	$8 \cdot 10^{-1}$	0	10^{-67}	$5 \cdot 10^{-76}$	0	$2 \cdot 10^{-100}$	$4 \cdot 10^{-6}$	$2 \cdot 10^{-1}$
11	$7 \cdot 10^{-1}$	$5 \cdot 10^{-2}$	1	0	10^{-32}	$3 \cdot 10^{-52}$	$4 \cdot 10^{-150}$	10^{-50}	$8 \cdot 10^{-1}$	$5 \cdot 10^{-1}$
12	$8 \cdot 10^{-1}$	$9 \cdot 10^{-1}$	$2 \cdot 10^{-1}$	0	$4 \cdot 10^{-7}$	$4 \cdot 10^{-25}$	$6 \cdot 10^{-75}$	$9 \cdot 10^{-8}$	$4 \cdot 10^{-1}$	$6 \cdot 10^{-1}$
13	$4 \cdot 10^{-1}$	$2 \cdot 10^{-1}$	$8 \cdot 10^{-1}$	10^{-194}	$6 \cdot 10^{-1}$	10^{-5}	$2 \cdot 10^{-7}$	$8 \cdot 10^{-4}$	$7 \cdot 10^{-1}$	$6 \cdot 10^{-1}$
14	$8 \cdot 10^{-1}$	$3 \cdot 10^{-1}$	$3 \cdot 10^{-3}$	$3 \cdot 10^{-102}$	$4 \cdot 10^{-1}$	10^{-6}	$3 \cdot 10^{-10}$	$7 \cdot 10^{-1}$	$2 \cdot 10^{-1}$	$9 \cdot 10^{-2}$
15	$6 \cdot 10^{-1}$	$8 \cdot 10^{-1}$	$4 \cdot 10^{-1}$	10^{-47}	$9 \cdot 10^{-1}$	$4 \cdot 10^{-1}$	$3 \cdot 10^{-2}$	$6 \cdot 10^{-1}$	$8 \cdot 10^{-1}$	$5 \cdot 10^{-1}$

Here, the S-boxes play a crucial role by introducing nonlinearity into the process. On the other hand, S-boxes alone would mix together only a few near bits and final permutations could never satisfy the avalanche criterion.

We tested 10 randomly generated permutations with an input size of 64 bits. Each of the permutations uses an S-box of size 4 bits. We generated 10 MB of data using the CTR strategy and provided it to the permutations with various numbers of rounds. Table 2.1 shows the results of BoolTest used on these output data. We used BoolTest with a block size set to 64 bits as it is the output size of the functions and both the degree and the number of summands set to 2 since it is one of the recommended settings.

With the *significance level* $\alpha = 10^{-4}$, all the permutations but one produce data that seem to be random when the number of rounds is set to 15. The results also show that the number of rounds needed to produce seemingly random data differs among the functions. Therefore, when evaluating new distinguisher constructions, it is important to find a number of rounds for each function such that it is feasible to find good distinguishers, but at the same time, it is not too simple, i.e. randomly chosen Boolean function has a low probability of being a good distinguisher. We can use BoolTest for this purpose.

We can also notice that the change in *p-value* between the rounds is not too abrupt. This is helpful as we have multiple rounds of the same function, where the non-random patterns get gradually weaker.

Chapter 3

Approximated Distinguishers

In BoolTest, distinguishers are constructed by evaluating many simple distinguishers and creating new, more complex ones based on the previous results. In this chapter, we aim to find a different approach to constructing distinguishers. We create just a single distinguisher based on the information we collect about the vectorial Boolean function we are investigating.

First, we approximate the function as a vector of Boolean polynomials by finding low-degree monomials of the polynomial representation of the function. We then create a linear combination of the polynomials with as few low-degree monomials as possible, i.e. a polynomial with no monomials of degree one and as few monomials of degree two as possible. Based on which polynomials were combined, we construct a distinguisher as a linear combination of the corresponding bits. As we construct and test only one distinguisher, we do not need any adjustment of the resulting *p-value* or the use of halving as the evaluation method.

This approach is based on a hypothesis that if we construct a distinguisher d that combines output bits of a cryptographic function F in such a way that low-degree monomials cancel out, it would significantly influence the probability $P(d(F(X)) = 1)$ that the distinguisher returns 1 for a block produced by F when compared to the probability $P(d(X) = 1)$ that the distinguisher returns 1 for arbitrary data. Let us show the idea behind the method in a simple example. Let the investigated function be $F(x_0, x_1, x_2) = (x_2 + x_0x_1,\ x_1 + x_0x_1,\ x_2) = (y_0, y_1, y_2)$. By combining the first and third bit, we obtain $(x_2 + x_0x_1) + (x_2) = x_0x_1$, with the corresponding distinguisher $d = y_0 + y_2$. We constructed the distinguisher as a sum of the first and third bit because the corresponding sum of polynomials representing the bits results in a polynomial with no monomial of degree one. We can compute the expected probability of the distinguisher returning 1 on data from the function as $P(d(F(X)) = 1) = P(x_0x_1 = 1) = 0.25$. On the other hand, the probability of returning 1 for random data is $P(d(X) = 1) = P(x_0 + x_2 = 1) = 0.5$. Since there is a significant difference

between the probabilities, the distinguisher works well on data produced by F.

In practice, we usually can not work with the exact polynomial representation of the function as the polynomials are too complex. Therefore, we use just an approximation of the function. However, since our hypothesis supposes that it is enough to eliminate the low-degree monomials, it is sufficient to know just the low-degree monomials of the polynomial representation of the function.

3.1 Function Approximation

As the first step of our construction, we need to find the polynomial representation of the investigated function limited to a given degree deg. We present two approaches. The first one works fully in a black-box manner and finds the polynomials by using queries to the function F. The second approach expects that we know the exact polynomials representing a round function and allows us to use this knowledge to approximate multiple iterations of this round function.

3.1.1 Black-box Approximation

For each possible monomial of degree at most deg, we can use queries to the cryptographic function to find out if the polynomial representation of an output bit of the function contains the given monomial. This way, we can monomial-by-monomial find the approximation of the examined function.

An algorithm to determine if a given monomial is contained in a Boolean function corresponding to an output bit of a black-box function is described in Algorithm 1. Let us discuss an example. We want to figure out if some black-box polynomial f contains, for example, the monomial $m = x_0x_1x_2$. For the monomial m to be evaluated to 1, we need to query f with input, where bits corresponding to the variables of m are set to 1 and all other variables to 0 so that they do not influence the result. However, all the monomials consisting of a subset of variables of m would be evaluated to 1 as well if they are part of f and would influence the result of our query. The solution to this problem comes directly from Theorem 1, presented and proved in [21]. To cancel out the influence of all the monomials consisting of subsets of variables of m, we can, for each such subset S, compute a vector v_S such that all variables from S are set to 1 and other variables are set to 0. Then $\sum_{S \subseteq m} f(v_s) \equiv 1 \ (mod\ 2)$ if and only if $m \in f$.

During the evaluation of the presence in a polynomial for all the monomials up to some degree, we can reuse calculations from lower-degree monomials for monomials of higher degrees. For each output bit of the cryptographic function, we perform exactly the same operations and calls to the black-box function. Therefore, we can avoid these unnecessary duplicate queries to the

Algorithm 1: Monomial's presence in polynomial

Input: A black-box polynomial f_i, a size n of input of f_i, a monomial m

Output: Information about monomial's presence in the polynomial

```
1  count := 0;
2  for variables ⊆ m do
3  |  input := Array() ;
4  |  for k ∈ 0...n do
5  |  |  if x_k ∈ variables then
6  |  |  |  input.push(1) ;
7  |  |  else
8  |  |  |  input.push(0) ;
9  |  if f_i(input) = 1 then
10 |  |  count := count + 1 ;
11 if count is odd then
12 |  return true;
13 else
14 |  return false;
```

function by computing all the approximations at once. This way, we can approximate polynomials $f_0, ...f_{n-1}$ in $O(k \cdot \sum_{i=0}^{deg} \binom{size}{i} + 2^{deg} \cdot size \cdot \sum_{i=0}^{deg} \binom{size}{i})$, where k is the cost of one request to the black-box function, $size$ is the function's output size in bits, and deg is the maximal degree of monomials, we are approximating. The first summand in the complexity corresponds to all the requests to the black-box function needed. The second summand represents the complexity of constructing the polynomials based on the outputs of the requests.

3.1.2 Round-by-round Approximation

When we know the implementation of the function we are investigating (e.g. code implementing a hash function), we can try a different approach. We can look directly at the implementation and deduce the polynomials from the operations applied to the input data. In the case of multiple rounds, we can deduce the polynomials of one round and then use them to build an approximation of multiple rounds.

Figure 3.1 shows a concrete example of the following procedure. Let $(a_0^n, ..., a_{k-1}^n)$ be the polynomial approximation up to a degree deg of n rounds of a cryptographic function F and let f_i be the polynomial representing i-th output bit of one round of the same function. Then, we can substitute variables $x_0, ..., x_{n-1}$ in f_i with polynomials $a_0^n, ..., a_{k-1}^n$, respectively. After

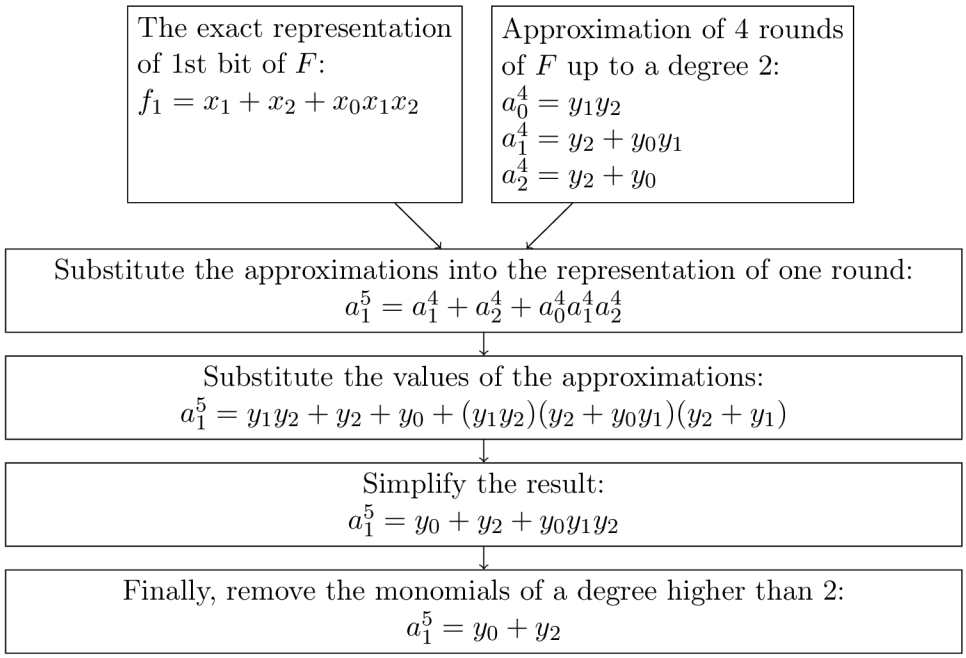


Figure 3.1: Example of substitution of approximated bits of 4 rounds of a function into an exact representation of one round of first bit, to obtain an approximation of 5 rounds of the first bit.

removing monomials with degree $> deg$, we obtain an approximation a_i^{n+1} of the i-th bit of $n+1$ rounds of the function.

It is important that f_i is the exact polynomial representing the i-th bit; otherwise, some monomials can be lost during the substitution. Moreover, it is necessary to substitute the approximations of n rounds into the polynomials representing 1 round and not the other way around. While the other way is possible as well, it is not advised since we would need to compute everything exactly and only at the end of the procedure remove the monomials of high degrees.

The complexity of this approach is $O(size \cdot \sum_{i=0}^{deg_1} \binom{size}{i} \cdot (\sum_{i=0}^{deg} \binom{size}{i})^{deg_1})$, where deg_1 is the maximal degree of monomials in the polynomial representation of one round of the function.

When we compare the complexity of this and the previous technique, it is clear that the black-box approach is more practical in most cases. The round-by-round technique can be more efficient only in cases when the calls to the function are extremely expensive, polynomials representing one round have a really low degree, the polynomials are sparse, or we want an approximation up to a very high degree. In our case, the calls to the function are computationally efficient, we need approximation of only low degrees, and

we do not expect any special properties of the polynomials. Therefore, in this work, we use the black-box approximation.

3.2 Elimination of Low-degree Monomials

When we have the cryptographic function approximated as a vector of polynomials of a degree at most *deg*, we can use them to create the distinguisher. Now, we want to find a linear combination of the polynomials such that it has as few monomials of low degrees as possible, e.g. eliminate all monomials of degree one and as many as possible of degree two.

First, we construct a matrix M based on the polynomials. The matrix contains a column for each possible monomial up to a degree *deg* and rows for each of the polynomials. The value of a cell is 1 if a polynomial contains a given monomial and 0 otherwise. For example, polynomials $f_0 = x_1 + x_0x_1 + x_1x_2$, $f_1 = x_0 + x_1 + x_1x_2$, $f_2 = x_0 + x_0x_1 + x_0x_2$ can be transformed into the following matrix:

$$\begin{array}{c} \\ f_0 \\ f_1 \\ f_2 \end{array} \begin{array}{c} \begin{array}{cccccc} x_0 & x_1 & x_2 & x_0x_1 & x_0x_2 & x_1x_2 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

We apply Gaussian elimination to this matrix and try to eliminate as many low-degree monomials as possible. Finally, we take the last row, which has the most low-degree monomials eliminated, and examine which rows were combined to obtain it. If the indices of the combined rows were $i_1, i_2, ..., i_k$, we take the polynomial $y_{i_1} + y_{i_2} + ... + y_{i_k}$ as the distinguisher. The disadvantage of this construction is that the produced distinguishers are always linear.

The process can be improved by considering the inputs used during our experiments. When we use a CTR strategy to generate inputs, some variables are always set to zero. In such a case, we can omit all monomials containing these variables from the approximation of the cryptographic functions, as they would be evaluated to zero for any of the inputs. With this simplification, the Gaussian elimination often ends with a significantly simpler polynomial in the last row.

3.3 Evaluation

Our hypothesis is that a distinguisher that combines output polynomials of a function in such a way that as many low-degree monomials as possible are eliminated should have significantly different outputs on blocks produced by this function when compared to random blocks.

Table 3.1: Results of presented method on 10 MB of data generated using CTR strategy with 10 permutations from Chapter 2.

permutation	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9
rounds	8	7	8	15	11	12	12	11	8	8
p-value	0.5	0.8	0.2	0.06	0.5	0.2	0.001	0.5	0.8	0.7

In this section, we evaluate this method to find out whether our hypothesis holds or not. For the evaluation of this method, we use the permutations from the previous chapter. Each of the permutations is used with a different number of rounds, chosen based on the results of BoolTest so that a strong non-random pattern is present in the data. Inputs for the functions are generated using the CTR strategy. Therefore, we use the optimization that all the monomials with at least one variable always set to 0 are ignored during the construction of the distinguisher.

The results are shown in Table 3.1. Distinguishers for all 10 functions resulted in a *p-value* larger or equal to 0.001, and in most cases, the *p-value* is considerably larger. Clearly, our method failed for these functions as the distinguishers found by BoolTest had significantly lower *p-values*. However, there are some use cases where the method works really well. One such example is the pseudorandom number generator Xorshift64.

3.3.1 Xorshift

Xorshift is a family of very simple and fast pseudorandom number generators invented by George Marsaglia [22]. As the name suggests, the only operations they use are `XOR` and bitwise `SHIFT`.

We primarily focus on Xorshift64, which is a specific sub-class of the algorithms. Algorithm 2 shows an example of the Xorshift64 algorithm. To obtain different algorithms from this family, both the direction of the `SHIFT` and the shift amount can be changed.

Algorithm 2: Xorshift64

Input: A 64-bit state s
Output: New state
1 $s := s \oplus (s << 18)$;
2 $s := s \oplus (s >> 31)$;
3 $s := s \oplus (s << 11)$;
4 **return** s

Xorshift produces a new value in a pseudorandom sequence by applying the function described by the algorithm to the last 64-bit integer of the

sequence. For the first call to the function, a randomly generated 64-bit seed is used.

Panneton and L'ecuyer analyzed the Xorshift pseudorandom number generators and revealed several weaknesses of the generators [23]. Vigna followed up in 2016 by testing all the variants of Xorshift64 generators using TestU01's SmallCrush test suit [24]. He showed that all the variants systematically fail the MatrixRank test. He sorted the variants based on the number of tests they failed in total and tested the best variants also using the BigCrush test suit. The variant of Xorshift64 that is described by Algorithm 2 is the one that turned out to be the most resilient to the statistical tests. We use this variant to test our method for constructing distinguishers.

From the fact that Xorshift uses only `XOR` and bitwise `SHIFT`, it is clear that the polynomials representing the output bits of the function are linear. So, when we use the black-box approach of approximating the polynomials with a degree set to at least 1, we are actually getting the exact polynomials. The polynomials of the Xorshift64 that we examine are shown in Appendix B.

As the input of the function is not chosen by us but randomly generated by a previous call to the generator, we can not use the optimization that some monomials are ignored. After straightforwardly applying the elimination of low-degree monomials, we end up with a polynomial x_{63} and a corresponding distinguisher $y_1 + y_{12} + y_{23} + y_{32} + y_{34} + y_{43} + y_{45} + y_{54} + y_{56} + y_{63}$. From the construction, it is clear that the distinguisher always returns the value of the last bit. The expected probability of the distinguisher returning 1 for random input is 0.5. Since Xorshift has decent statistical properties, the probability of the last bit being 1 is also close to 0.5. Therefore, this distinguisher is expected to fail to detect any statistical abnormalities. So, we need to slightly improve the approach to eliminate all the monomials.

We know that each block is constructed using the previous one as an input. Hence, if we construct the distinguisher for a pair of consecutive blocks, we can think of the first block as an input and bit i can be described simply as a polynomial x_i. When we express the bits of the first block in this way and the bits of the second block as the approximated polynomials, we can use the elimination again. This time, the resulting polynomial is 0, exactly as we hoped. The distinguisher $d = y_1 + y_{12} + y_{23} + y_{32} + y_{34} + y_{43} + y_{45} + y_{54} + y_{56} + y_{63} + y_{127}$ returns 0 for any input that consists of anything in the first part and output of the Xorshift, when inputted with the first part, in the second part. As shown in Table 3.2, the *p-value* obtained for this distinguisher with as little as 50 blocks is $\sim 6 \cdot 10^{-8}$ and declines when more pairs of blocks are used. After a manual investigation of the polynomial representation of the Xorshift64, we discovered that even shorter distinguishers can achieve the same results, e.g. $d_2 = y_{53} + y_{86} + y_{104} + y_{117}$.[1]

[1]Be aware that we load bits in a different order than BoolTest. For use with Booltest, d_2 would be $d_{2b} = y_{77} + y_{46} + y_{16} + y_{13}$.

Table 3.2: *P-values* of the distinguisher d for different amounts of data generated by a particular version of Xorshift64 with any seed.

#output blocks	size	*p-value*
20	320 B	$2 \cdot 10^{-3}$
50	800 B	$6 \cdot 10^{-8}$
100	1.6 KB	$2 \cdot 10^{-15}$
1000	16 KB	$6 \cdot 10^{-151}$

As we showed, this distinguisher construction works great for functions expressible by linear polynomials. It could be further expanded to produce distinguishers of higher than linear order by adding polynomials corresponding to multiplications of 2 or more bits of the output into the matrix for elimination. However, the number of combinations of 2 or more bits is high, and we would also need an approximation of the function up to a higher degree. These improvements in the approach would be extremely computationally inefficient. As we showed, for more complex functions, this approach does not produce good distinguishers. Therefore, in the next chapter, we explore relations between good distinguishers and the function for which they are constructed.

Chapter 4

Distinguisher Analysis

As the previous technique worked only for very specific types of functions, we decided to study the common traits of good distinguishers. In this chapter, we present experiments that help to understand how the properties (e.g. degree or number of monomials) of a Boolean function influence its suitability as a distinguisher. The first experiment compares distinguishers constructed using `AND` to those constructed using `XOR`. In the second experiment, we use a more general approach and test distinguishers of any possible form.

4.1 Random Sampling

Boolean functions in ANF are naturally constructed using the operations `XOR` and `AND`. BoolTest does it exactly this way. BoolTest first creates monomials using `AND` and then combines them using `XOR`. In this experiment, we would like to determine which one of the operations by itself creates stronger distinguishers for the outputs of our permutation functions. The results can help us to identify how to construct distinguishers for outputs of cryptographic functions and which operation we should prefer.

We examine distinguishers constructed as addition (`XOR`) or multiplication (`AND`) of 1, 2, 3, 5, and 10 variables. For example, if we randomly sample 3 variables y_0, y_3, y_6, the distinguisher constructed using addition is $d_1 = y_0 + y_3 + y_6$, and the distinguisher constructed using multiplication is $d_2 = y_0 y_3 y_6$. We randomly sampled 32 distinguishers of each kind. These distinguishers are then evaluated on the 10 permutations from Chapter 2. Table 4.1 shows results in the form of the minimal *p-value*.

We can notice that with addition, the *p-value* rapidly increases with summands added. Already, for three summands, all the lowest *p-values* are higher or equal to 10^{-3}. On the other hand, with multiplication, the increase in *p-value* is significantly slower, and even the distinguishers formed as multiplication of 5 variables can discover strong non-randomness. It makes sense that with a higher number of variables, the minimal *p-value* increases.

Table 4.1: The minimal *p-values* of 32 randomly sampled distinguishers created using only `XOR` or `AND` operation with 1, 2, 3, 5 or 10 variables evaluated on random permutations from Chapter 2. 1VAR means that the distinguisher consists of just 1 variable; other distinguisher types are labelled with an operation and a number of variables. The value 0 indicates *p-value* $< 10^{-300}$.

	perm0	perm1	perm2	perm3	perm4	perm5	perm6	perm7	perm8	perm9
1VAR	1e-293	3e-40	1e-47	7e-18	9e-38	2e-49	1e-59	5e-60	0	6e-19
XOR2	3e-69	3e-69	3e-30	4e-2	2e-3	1e-2	4e-19	2e-2	4e-36	6e-12
XOR3	3e-2	3e-2	1e-3	4e-3	9e-2	1e-2	9e-3	1e-2	3e-3	3e-2
XOR5	2e-2	1e-2	7e-3	7e-2	6e-2	8e-3	6e-2	4e-2	1e-2	6e-3
XOR10	2e-2	4e-2	9e-3	8e-2	1e-2	4e-2	9e-3	1e-2	3e-2	7e-2
AND2	0	6e-28	4e-19	2e-70	4e-18	8e-14	8e-56	4e-18	2e-76	3e-22
AND3	9e-76	3e-13	4e-8	1e-39	1e-32	3e-27	1e-39	3e-13	3e-115	9e-16
AND5	5e-13	4e-8	6e-5	4e-17	3e-14	5e-7	9e-14	3e-9	8e-19	5e-7
AND10	2e-3	7e-3	1e-1	2e-3	5e-2	1e-2	1e-2	3e-3	1e-2	7e-2

Distinguishers with a higher number of variables have a higher chance of containing variables that cancel each other's bias or variables that are not biased and hide the bias of other variables. However, this should influence both types of distinguishers similarly. We conclude that, at least for the permutations we constructed, it seems better to construct distinguishers as single monomials than as linear polynomials with many monomials. This observation partially explains the failure of our technique tested in Chapter 3. With the subsequent experiment, we analyze functions and distinguishers of more general forms.

4.2 Evaluation of All Distinguishers

In the first experiment from Section 4.1, we tested just a few functions and a tiny proportion of distinguishers. It would be interesting to explore all the possible functions and distinguishers on a given number of bits. This way, we could possibly figure out the relations between sources of data and distinguishers that perform well on such data. Such a relation can be used to propose a new distinguisher construction, which could provide us with stronger distinguishers.

We want to enumerate all vectorial Boolean functions on some number of bits and, for each of them, compute the *Z-score* for all the possible distinguishers. However, there are $(2^n)^{2^n}$ functions on n bits and $2^{(2^n)}$ Boolean functions that can be used as a distinguisher. Therefore, we work with functions on only 3 or 4 bits at most. Already, for the 3 bits, there are

Table 4.2: Example of enumeration of all distinguishers and classes of functions on 3 bits and their corresponding *Z-scores*. The full table is part of the electronic attachments.

	d_0	d_1	d_2	...	d_{125}	d_{126}	d_{127}
00000000	0	7.5	-1.1	...	1.6	-4.9	1.1
00000001	0	6.4	0	...	0.8	-4.1	1.1
00000002	0	6.4	-1.1	...	1.6	-4.1	1.1
...	...	...	...	...	...	...	...
66777777	0	-1.1	-1.1	...	-3.3	-3.3	-5.3
67777777	0	-1.1	-1.1	...	-4.1	-4.1	-6.4
77777777	0	-1.1	-1.1	...	-4.9	-4.9	-7.5

4 294 967 296 combinations of functions and distinguishers, and for 4 bits, there are more than 10^{24}.

As functions on 3 bits have just 8 inputs, we can use all the inputs to generate outputs on which we evaluate the distinguishers. Table 4.2 demonstrates the resulting *Z-scores* for each combination of a function in a row and a distinguisher in a column. We first describe how we enumerate the distinguishers and how the label for each distinguisher corresponds to a Boolean function, and then we explain how we represent the vectorial functions.

In the table, there are only 128 distinguishers, even though there are $2^{2^3} = 256$ Boolean functions on 3 bits. It is because we do not differentiate two distinguishers if they differ only in the absolute term. We do this because we compare the quality of distinguishers based on the absolute values of their *Z-scores*, and the absolute term changes only the sign of the *Z-score* and not its value. Now, let us explain how a given label d_i corresponds to a Boolean function. As we have 8 possible input vectors ($Y_0 = 000$,...,$Y_7 = 111$) for the distinguishers, we can encode the distinguishers as 8-bit numbers, with each bit representing the output for one vector. For example, the distinguisher d_{68} corresponds to the binary number 01000100. Such distinguisher returns 1 for inputs 010 and 110 since it has 1 on the third and seventh positions from the right and returns 0 for all the other inputs as all other positions in the binary representation are 0.

The encoding of the vectorial functions is slightly more complicated. There are 8 binary inputs and outputs, which we can represent with their corresponding decimal representation 0...7. We represent a function as a vector of outputs, where k-th value is the output for the k-th input, e.g. the identity function would be represented as a vector 01234567 and the vector 53764215, corresponds to a function F such that $F(0) = 5$, $F(1) = 3$ $F(2) = 7$ and so on. However, since we use all the inputs to evaluate the distinguisher, we always use the whole vector as the multiset of inputs for the distinguisher. Therefore, the vectorial functions with the same number of each value in the

vector have identical *Z-scores* on all distinguishers. To make the enumeration more efficient and simplify the later processing of the data, we merge these functions with identical multisets of outputs into classes represented by one vector with a nondecreasing order of values. For example, the vectors consisting of three 0s, two 1s and three 2s (i.e. 22211000, 22210100,...) are represented by the vector 00011222. This way, we have 6 435 classes of functions instead of the original 16 777 216 functions.

With 823 680 combinations, it is still difficult to aggregate the data so that some interesting patterns appear. We tried several ways of processing the data to find something interesting. We looked at the best distinguisher for each class of functions. We grouped the distinguishers for each data based on their *Z-score*. We substituted the Boolean functions from the vectorial boolean function into a distinguisher and compared the number of monomials of such substituted distinguishers and their *Z-score*. None of these attempts showed any interesting property of the good distinguishers. Finally, we found that when we create a histogram for each distinguisher, where the bins represent the frequency of absolute values of *Z-score*, we can group the Boolean functions into 5 groups, in which all the functions share the same histogram.

Table 4.3 shows the groups and number of classes of functions for which the distinguishers produce each possible absolute value of the *Z-score*. For simplicity, the *Z-scores* are rounded to one decimal place; therefore, some *Z-scores* are merged together in the table. The Boolean functions in each group have one thing in common: they have the same distance from their closest balanced function. Table 4.4 shows the properties of each group. The first group contains constant functions, group 2 contains functions that return 1 for 1 or 7 distinct inputs, and groups 3 and 4 have distance from balanced functions 2 and 1, respectively. Finally, group 5 contains balanced functions.

Apparently, the least balanced functions (when we ignore the constant ones) can, for some data, yield the most extreme *Z-scores*. On the other hand, they also have the highest probability of returning a *Z-score* of 0. With higher proximity to balancedness, the maximal *Z-score* declines, and so does the number of *Z-scores* equal to 0. The reason why the least balanced

Table 4.3: The counts of classes of functions for which distinguishers from different groups produce a given *Z-score*.

Z-score	7.5	6.4	5.3	4.9	4.3	4.1	3.7	3.2	2.9	2.8	2.4	2.2	2.1	1.6	1.5	1.4	1.1	0.8	0.7	0
Group1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6435
Group2	1	7	28	0	84	0	0	210	0	0	0	0	462	0	0	0	3927	0	0	1716
Group3	0	0	0	9	0	48	147	0	0	0	336	0	0	1917	0	0	0	2592	0	1386
Group4	0	0	0	0	0	0	45	0	180	0	0	915	0	0	1725	0	0	0	2310	1260
Group5	0	0	0	0	0	0	0	0	0	330	0	0	960	0	0	1680	0	0	2240	1225

Table 4.4: Description of groups of Boolean functions (distinguishers) on 3 bits by the number of 1s they return and the average *p-values* of the groups.

	Returned 1s	Avg. *p-value*
Group 1	0/8	1.00
Group 2	1/7	0.40
Group 3	2/6	0.32
Group 4	3/5	0.29
Group 5	4	0.34

functions can have the highest *Z-scores* is due to their low expected probability of returning 1 (or 0). When such a distinguisher meets with a vectorial function that frequently outputs the value for which the distinguisher returns 1 (or 0), the resulting *Z-score* is high. At first, these observations seem trivial and rather useless for functions with larger blocks as the blocks in practice never repeat for different inputs. However, we do not need to look at whole blocks. Such frequent combinations of bit values can be looked for on just a subset of the block's bits. We should seek as common combinations of bit values as possible on as many bits as possible. If we can find a frequent combination of bit values, a distinguisher that returns 1 just for this combination should perform well. If the combination is on a large number of bits, the distinguisher will have an extremely low expected probability of returning 1, which helps to make the *Z-score* large.

We decided to figure out which group of distinguishers is the best performing when picking a distinguisher randomly. We computed the average *p-values* for each of the groups as e^{log_avg}, where log_avg is the average of logarithms of the *p-values*. The averages are shown in Table 4.4. The average *p-value* is lowest for functions with distance 1 from balanced. With bigger distances of the functions from balanced, the average *p-value* gradually increases, but the group of balanced functions has one of the highest average *p-values*.

In Appendix C, we show the same data for functions on 4 bits. The general pattern is there the same. If we want to reach a high *Z-score*, it seems that we should start with almost balanced functions and gradually make them less balanced as long as the *Z-score* improves. This way, we can find a distinguisher, with a large distance from balanced functions, that exploits unexpectedly high occurrence of some values in the data. We use these principles in the next chapter to design a BoolTest-like heuristic for finding distinguishers of high degrees.

Chapter 5

CoolTest

In this chapter, we present the randomness-testing tool, CoolTest. Based on the observations in the previous chapter, we design a construction that is more effective in finding good distinguishers than the one currently used by BoolTest. First, we present a three-phase algorithm. We continue by explaining the main weakness of this approach and some ways to mitigate this weakness. Then, we describe details about the implementation of the tool. After that, we test the implementation on data generated by 12 hash functions and block ciphers and compare the results of CoolTest with BoolTest.

5.1 Distinguisher Construction

As we discussed in Chapter 4, a good approach to distinguisher construction is to start with an almost balanced function and then gradually make the function less and less balanced. This could be done by using single monomials. We could start with a monomial of a low degree. Then, we could iteratively multiply the monomial by new variables to obtain less and less balanced functions.

What we described above is almost the algorithm that we use. However, restricting the distinguishers just to monomials would be too limiting. Therefore, we decided to use a bit more general type of distinguishers. The distinguishers that the algorithm searches through are created by multiplying together multiple variables and some of the variables can be negated, e.g. $x_1 \cdot (x_3 + 1) \cdot (x_5 + 1)$ (note that negation in ANF is expressed by $+1$). A simple way to interpret this distinguisher is that it returns 1 if and only if the second bit of an input is 1 and the fourth and sixth is 0. Since these Boolean functions are constructed as a generalization of monomials by adding negations, we call them *n-monomials*.

Our distinguisher construction consists of three phases. In the first two phases, we use the distinguishers in the form of *n-monomials*. In the last phase, we convert these distinguishers into more complex ones. We devote

a subsection to each of the phases and describe them comprehensively, but first, let us present a brief overview of the phases.

1. First, we evaluate all *n-monomials* of a given degree.
2. Then, we take a pool of the best k distinguishers from the first phase. We iteratively add new variables (or their negations) to the distinguishers as long as their *Z-score* improves.
3. Finally, we take the best distinguishers from Phase 2 and strengthen them by combining some of them together using logical `OR`. Alternatively, we take the single best distinguisher from Phase 2 and make a histogram of values of the variables used in the distinguisher. Based on the histogram, we try to create the best distinguisher possible on the given bits.

5.1.1 Phase 1 – Brute-force Search

The first phase is similar to BoolTest. BoolTest, in the first phase, searches through all the monomials of the form $x_{i_1} \cdot x_{i_2} \cdot ... \cdot x_{i_{deg}}$ of some degree deg. CoolTest tests all the distinguishers created from the single monomials by negating any subset of its variables. For example, if we have the monomial x_1x_2 by negating any subset of variables, we would get these polynomials: x_1x_2, $(x_1+1)x_2$, $x_1(x_2+1)$ and $(x_1+1)(x_2+1)$. So, for a given degree deg and block size n, CoolTest iterates through $2^{deg} \cdot \binom{n}{deg}$ instead of BoolTest's $\binom{n}{deg}$ distinguishers.

These distinguishers are evaluated and sorted based on their *Z-score*. Then, k distinguishers with the highest *Z-score* are kept for the second step of the algorithm. Notice that, unlike BoolTest, CoolTest does not choose the distinguishers with the highest *absolute values* of *Z-score*, but only the *positive* maximal values. At first, it might seem unintuitive that potentially good distinguishers with negative *Z-scores* are ignored. However, a negative *Z-score* means that the pattern occurs less frequently than expected; e.g. if the pattern has an expected probability of occurrence 0.5 and occurs 400 times in 1000 samples, the *Z-score* is $(400 - 500)/\sqrt{0.5 \cdot 0.5 \cdot 1000} = -6.3$. We want to maximize the difference between the expected and the real number of occurrences of a given pattern, and during Phase 2, the algorithm tries to lower the expected probability as much as possible, i.e. with each added variable, the probability is reduced by a factor of 2. For example, commonly, the distinguishers from the second phase have more than 10 variables, and already for 10 variables the expected probability of the distinguisher returning 1 is less than 0.001. Therefore, the infrequent patterns are likely to lead to weaker distinguishers than the frequent patterns.

5.1.2 Phase 2 – Greedy Search

In the second phase, we start with a pool of k distinguishers. Then, we continue in iterations, each iteration creating distinguishers of a higher degree than in the previous one.

In a given iteration, we try to find a successor for each distinguisher in the pool. For each distinguisher, we take all the variables that are not yet part of the distinguisher. We create new distinguishers by multiplying the old one by the new variable or a negation of the variable. For a new distinguisher to be chosen as a successor, it needs to:

- have a higher *Z-score* than its predecessor
- not have been already chosen as a successor of any other distinguisher
- have the highest *Z-score* of all such distinguishers

If we successfully find a successor, it goes into the distinguisher pool for the next iteration. If a successor is not found, the old distinguisher is added among the final distinguishers for the third phase and no distinguisher is added to the distinguisher pool for the next round. The second phase stops when, upon finishing an iteration, the distinguisher pool for the next iteration is empty, and therefore, we have k distinguishers selected for the third phase.

Figure 5.1 shows a simple example of one iteration of this process. First, the potential successors of x_0x_2 are evaluated. Two of them have a higher *Z-score* than the original distinguisher. As none of them has been chosen

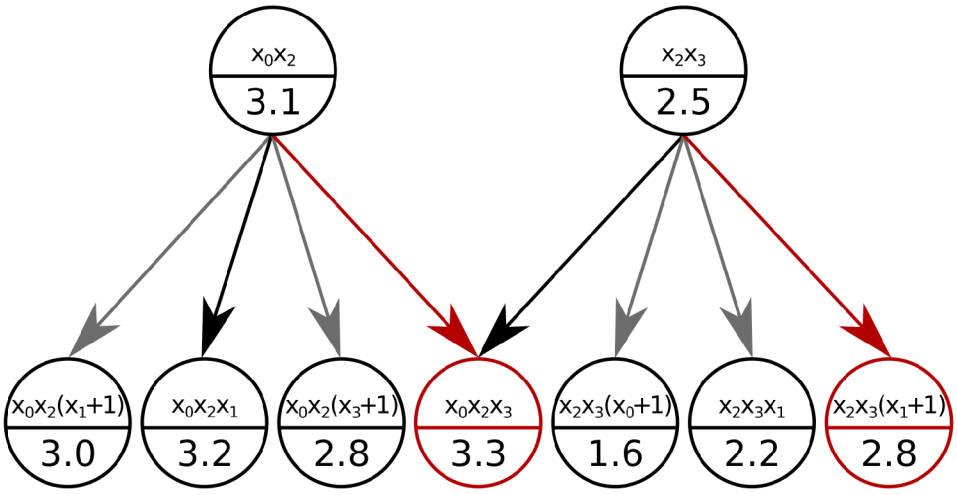


Figure 5.1: One iteration of the CoolTest's greedy search, where each node shows one distinguisher and its *Z-score*. Red arrows point to successors, black arrows point to candidates with a higher *Z-score* than the original distinguisher and grey arrows point to the candidates with a lower *Z-score*.

as a successor of any other distinguisher, $x_0x_2x_3$ is chosen as a successor of x_0x_2 because it has the highest *Z-score*. The process is similar for x_2x_3. However, the best candidate for a successor is again $x_0x_2x_3$. Since it is already a successor of another distinguisher, the second-best candidate is chosen in this case. If the second-best distinguisher had a lower *Z-score* than its predecessor, we would save x_2x_3 as one of the candidates for the third phase. The pool in the next iteration would then have one fewer distinguisher.

5.1.3 Phase 3 – Distinguisher Strengthening

It is possible to directly take the best distinguisher from the second phase and use it as the final distinguisher. However, we decided to try to do a bit more and create a stronger distinguisher. We present here two options for doing so.

The first approach uses the fact that we have more than just the best distinguisher found; we have multiple good candidates for a distinguisher. Therefore, it makes sense to combine them together as we expect the combination to be more robust.

The distinguishers from Phase 2 determine bits and their values, and an input block must perfectly match these values for the distinguisher to return 1. So, it seemed reasonable and natural to combine multiple distinguishers so that it is sufficient that the block matches just one of the combined sub-distinguishers.

To put it together, for all subsets $S \subseteq \{d_1, ...d_k\}$ of size n, where $d_1, .., d_k$ are distinguishers from the second phase, we evaluate the distinguisher $\bigvee_{d \in S} d$, e.g. for $n = 3$ resulting distinguisher could be $x_0x_5x_6x_7$ `OR` $(x_8+1)x_{23}(x_{78}+1)$ `OR` $x_{33}(x_{44}+1)$. Then, we choose the best such distinguisher as a final distinguisher and evaluate it on the second half of the data. The best combination is such that it is made of strong distinguishers, and at the same time, the distinguishers only rarely return 1 for the same block. The advantage of this construction is that it should be robust, i.e. if some of the patterns occur only in the first half of the data, the other patterns can still prove the non-randomness of the second half.

The second way to find a stronger distinguisher focuses solely on the strongest distinguisher from the second phase. The technique is based on [25], which is inspired by [26]. Figure 5.2 shows how this approach works. We take the bit positions from the distinguisher and create a histogram showing the frequency of all possible values of the bits. Using the histogram, we can efficiently evaluate any distinguisher defined on the chosen bits. We sort the bins of the histogram according to their frequencies. A distinguisher is then created by splitting the bins into two groups. For one group, the distin-

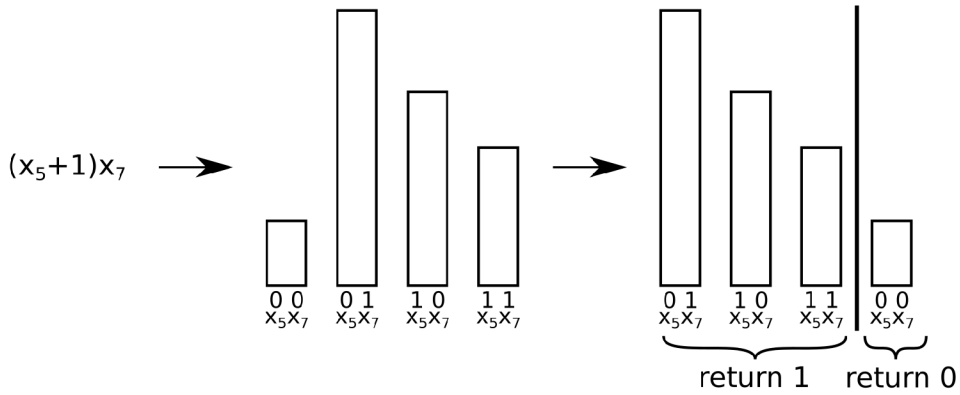


Figure 5.2: Transformation of polynomial distinguisher to a distinguisher based on a histogram.

guisher returns 1, and for the other 0.[1] We try all the possible ways to split the sorted sequence into two groups and evaluate all the distinguishers. The best distinguisher is chosen and then evaluated on the testing part of the data. At the end of this chapter, we present a version of CoolTest, which uses this type of distinguishers in the first two phases.

5.2 Early Stopping

While testing the described algorithm, we discovered an issue that happens mainly for data with significant non-randomness. The issue lies in the second part of the procedure. For some data, the *Z-score* improves for many iterations, resulting in a distinguisher with inappropriately many variables. This causes two problems.

- The greedy search takes more time than would be necessary to find a good distinguisher.
- The obtained distinguisher does not generalize for just slightly different data.

The second is a bigger issue as it means that even though the distinguisher has a very high *Z-score* on the training data, it fails to find the same pattern in the testing data. Therefore, it fails to show that the data are not random. A similar issue was also discovered in [26], where they searched for the best possible distinguisher for given data, but it was often too specific and did not work well for different data from the same source.

[1] Notice that it does not matter for which group we return 1 and for which 0 as long as we use the absolute value of a *Z-score* to compare the distinguishers.

Let us describe a simple example of this issue. Let the training set be 1000 blocks of 128 bits, and one of them be a block of all 0s. If a distinguisher returns 1 for this block and 0 for all other blocks, its *Z-score* on the training set is less than 10^{-16}. However, if the testing data do not contain such a block, the testing *Z-score* would be ~ 0, and the data would be marked as random.

Generally, the solution in our case is to stop the greedy search early enough so that the obtained distinguishers do not overfit the first half of the data. We have discovered and tested several methods to decide when to stop the search. Below, we present the methods. The first uses a validation set to determine when the search should stop. Then, we present multiple similar approaches based on some metric that we can compute from the information we collect during the search without any modifications to the training data.

5.2.1 Validation

Our problem seems similar to overfitting in neural networks. Hence, it is natural to try to use a similar solution. In neural networks, the training data are often split into two parts – training and validation. The validation data are used during training to repeatedly check if an error is decreasing for both training and validation data. If the error starts increasing for the validation data, the training of the model stops [27].

With CoolTest, we can do the same thing. The only difference is that instead of stopping when the error increases, we stop the search when the *Z-score* on validation data decreases. Although this technique seemed promising, it is likely not ideal, especially with low amounts of data. The issue is that we lower the amount of training data. Therefore, the results on the same amount of data can be significantly worse when part of the data is used for validation. Moreover, the search can be stopped too early when the validation *Z-score* unluckily drops a little in one iteration of the search.

5.2.2 Simple Conditions

A simple but effective solution can be to stop the search based on a value of some metric. The most straightforward option is to set a maximal degree for the distinguishers and stop the search if this threshold is reached. This overcomes the issue of the search taking too long. On the other hand, it might be difficult to choose the correct limit so that the search does not stop just before finding a good distinguisher.

The *n-monomials* of higher degrees return 1 for a smaller and smaller number of inputs. Therefore, it could make sense to stop based on the difference between the expected number of 1s returned and the real count. This helps to stop when it would be difficult to distinguish a real improvement from a random one. However, this approach has two issues. One, it does

not help to stop when the bias in the data is too strong, and the difference between expected and actual outcomes of the distinguishers stays large for many rounds. Secondly, it can be tricky to choose the correct value of this parameter.

A wiser metric could be the change between consecutive iterations of the greedy search. Direct comparison of *Z-scores* is unsuitable as the change in *Z-score* is not fully independent of the number of previous iterations. The optimal solution seems to be computing the *p-value* of the change in each iteration. With each iteration, the new distinguisher returns 1 for a subset of inputs for which its predecessor returned 1. We can interpret this as a binomial test, where the sample space is all the blocks for which the predecessor returned 1, and the expected probability of returning 1 for each of the blocks by the successor is 0.5.

However, in practice, this approach does not work as expected. It turns out that for some sources of data (e.g. Java's PRNG `java.util.Random`), the improvements between iterations first significantly drop and then rise up again. So, by setting this limit too low, we would not find a good distinguisher for such functions. On the other hand, when the limit is set low enough for such functions, the same limit for other functions means that the search stops too late. In this work, we primarily use the first two simple conditions, but their suitability may vary based on a given scenario.

5.3 Implementation

The implementation of CoolTest is available under the MIT license on GitHub[2] as well as in the archive of this work. The structure of the files in the archive is described in Appendix A.

Our implementation is written in Rust, with one exception: we use the Python library SciPy[3] for the computation of *p-value* in binomial tests, as no Rust library offers this function as far as we are concerned. For evaluating distinguishers, we use a similar optimization as BoolTest. CoolTest loads 128 blocks of data into 128-bit integers and, using bitwise operators, evaluates distinguishers on multiple blocks at once.

Although we describe the distinguishers as polynomials, due to efficiency, we do not represent them as such. The *n-monomials* in which we search can be represented as a combination of bit positions and values. Moreover, the properties of the *n-monomials* allow us to overcome one of the biggest limitations of BoolTest – slow computation of the expected probability of returning 1. For *n-monomials*, the expected probability can be computed as $p = 2^{-n}$, where n is the number of distinct variables used in the polynomial.

[2] https://github.com/jirigav/cooltest
[3] https://scipy.org/

This simplicity of computation of the expected probability changes, with the use of distinguisher strengthening. With the histogram technique, it is still straightforward and efficient as it is sufficient to compute the number of bins for which the distinguisher returns 1 and divide it by the total number of bins. However, when we combine the polynomials using `OR`, the computation becomes slightly more complicated. We need to compute the probability of being evaluated as 1 at the same time for each subset of the polynomial's variables. Then, these probabilities are combined together using the inclusion-exclusion principle. To make it computationally less demanding, we add a parameter to limit the number of distinguishers from the second phase from which we are choosing in the third phase.

5.4 Evaluation

We evaluate CoolTest on 12 hash functions and block ciphers. We have chosen the function so that we can compare the results of CoolTest to the results of BoolTest and statistical batteries presented in [5].

In [5], these tools were evaluated on 100 MB of data generated using CTR, LHW, SAC, and RPC strategies. We decided to use the LHW strategy for generating input data as it seems to be the one best suited to both BoolTest and the statistical batteries. Sýs et al. do not specify the exact size of input vectors used for hash functions, nor do they mention what output length they used for hash functions that have it as a parameter. We decided to use the smallest possible output length for each hash function. Based on attempts to reproduce their results, we have chosen to set the length of input vectors to the size of the output for each function. Table 5.1 shows the number of rounds for which each tool suggested that the data are not random, as presented in [5].

We decided to run CoolTest in two configurations, each with a different distinguisher strengthening. We run BoolTest as well to see how it performs in the halving mode. We use the tools with the following settings chosen so that they need approximately the same time to test the data.

- *BoolTest* – BoolTest with `--deg=3` and `--combine-deg=3`, i.e. the final distinguisher could for example be $x_0x_3x_{112}+x_{18}x_{65}x_{77}+x_5x_{23}x_{31}$.
- *CoolTest `OR2`* – CoolTest with `--deg=2`, `--min-difference=100`, `--k=300` and `--patterns-combined=2`, i.e. CoolTest searches through monomials of degree 2 in the first phase, takes 300 of the distinguishers to the second phase and in the third phase combines two distinguishers. The resulting distinguisher can look, for example, like this $x_5(x_{23}+1)x_{68}$ `OR` $(x_{25}+1)x_3x_{110}(x_{78}+1)x_{66}$.[4]

[4] We set `--max-bits=16` in this configuration for 6 rounds of JH as the search would otherwise take too long due to strong non-random patterns.

- *CoolTest hist* – CoolTest with `--deg=2`, `--k=300`, `--max-bits=16`, `--min-difference=100`, `--hist`, i.e. the resulting distinguishers are created using the strengthening based on histograms with a maximum of 16 variables.

Table 5.1: The number of rounds of selected cryptographic functions in which non-randomness was still detected in 100 MB data generated using the LHW strategy, as presented in [5].

Function	NIST STS	Dieharder	TestU01	BoolTest
AES	2	3	3	3
Blowfish	2	3	3	3
DES	4	4	4	5
3-DES	2	2	3	3
Grostl	2	2	2	2
JH	6	6	6	6
Keccak	2	2	2	3
MD5	12	13	20	13
MD6	8	8	8	9
SHA-1	16	16	16	16
SHA-256	12	12	12	13
TEA	3	3	3	4

When executed on a laptop with the Intel Core i7-8550U CPU, all three finish in about 2 minutes with 100 MB of data and block size 128. With larger block sizes, BoolTest generally needs more time, as it searches through a considerably larger number of distinguishers in the first brute-force phase.

For both CoolTest and BoolTest, we use the setting twice with the block size set to the output size of the tested function and double the output size. For comparison, we then use the better of the two results similarly to [5], where multiple settings of BoolTest were used. Moreover, we also test a method that we have proposed in Section 2.2, i.e. randomly shuffling the blocks (or pairs of blocks) before using the tools to enhance performance with the halving strategy. We run each of the experiments in each setting just once.

Table 5.2 shows the results of BoolTest and two variants of CoolTest for original data, and Table 5.3 shows the results when the tools are provided with data with shuffled blocks. In both tables, we show the number of rounds as well as the *p-value* of the discovered distinguishers. To decide if a sufficiently strong pattern was found, we use *significance level* $\alpha = 10^{-4}$. The most interesting observations from the results are the following:

- In most cases, the tools produced stronger distinguishers when provided with the shuffled data. In some cases, they even managed to

Table 5.2: The number of rounds and *p-value* of the detected distinguisher of selected cryptographic functions in which non-randomness was still detected in 100 MB data generated using the LHW strategy. The value 0 indicates any *p-value* $< 10^{-300}$.

	BoolTest		CoolTest OR2		CoolTest hist	
	rounds	*p-value*	rounds	*p-value*	rounds	*p-value*
AES	3	$5 \cdot 10^{-82}$	3	0	3	0
Blowfish	3	$9 \cdot 10^{-37}$	3	$4 \cdot 10^{-285}$	3	$4 \cdot 10^{-285}$
DES	5	$2 \cdot 10^{-35}$	5	$2 \cdot 10^{-9}$	5	$8 \cdot 10^{-144}$
3-DES	3	$8 \cdot 10^{-36}$	3	$9 \cdot 10^{-111}$	3	0
Grostl	2	0	2	0	2	0
JH	6	0	6	0	6	0
Keccak	**2**	**0**	3	$5 \cdot 10^{-18}$	3	$6 \cdot 10^{-22}$
MD5	13	$3 \cdot 10^{-57}$	13	$2 \cdot 10^{-296}$	13	0
MD6	9	$8 \cdot 10^{-73}$	9	$5 \cdot 10^{-222}$	9	$5 \cdot 10^{-222}$
SHA-1	17	10^{-43}	17	$2 \cdot 10^{-72}$	17	$3 \cdot 10^{-218}$
SHA-256	13	10^{-79}	13	0	13	0
TEA	4	10^{-21}	4	$3 \cdot 10^{-189}$	4	$3 \cdot 10^{-189}$

Table 5.3: Shuffled blocks – The number of rounds and *p-value* of the detected distinguisher of selected cryptographic functions in which non-randomness was still detected in 100 MB data generated using the LHW strategy and blockwise shuffled. The value 0 indicates any *p-value* $< 10^{-300}$.

	BoolTest		CoolTest OR2		CoolTest hist	
	rounds	*p-value*	rounds	*p-value*	rounds	*p-value*
AES	3	$5 \cdot 10^{-112}$	3	0	3	0
Blowfish	3	$3 \cdot 10^{-110}$	3	0	3	0
DES	5	$2 \cdot 10^{-51}$	5	$3 \cdot 10^{-237}$	5	0
3-DES	3	10^{-34}	3	$2 \cdot 10^{-144}$	3	0
Grostl	2	0	2	0	2	0
JH	6	0	6	0	6	0
Keccak	3	10^{-11}	3	10^{-27}	3	$3 \cdot 10^{-39}$
MD5	13	$3 \cdot 10^{-48}$	**14**	$\mathbf{10^{-17}}$	**14**	$\mathbf{7 \cdot 10^{-32}}$
MD6	9	$2 \cdot 10^{-105}$	9	$7 \cdot 10^{-237}$	9	$7 \cdot 10^{-237}$
SHA-1	17	$9 \cdot 10^{-39}$	17	$2 \cdot 10^{-63}$	17	$3 \cdot 10^{-247}$
SHA-256	13	$2 \cdot 10^{-37}$	13	0	13	0
TEA	4	$3 \cdot 10^{-29}$	4	10^{-182}	4	10^{-182}

find a good distinguisher for more rounds of the cryptographic function when the shuffling was used. BoolTest found non-randomness in 3 rounds of Keccak in the shuffled data, while it failed to do so with the unshuffled data. Moreover, both variants of CoolTest found non-randomness in shuffled data produced by 14 rounds of MD5, while with unshuffled data, it reached only 13 rounds.

- We managed to find suspicious patterns in data produced by 17 rounds of SHA-1 with all three tools, while Sýs et al. succeeded with BoolTest with at most 16 rounds. At first, this result seems surprising as we use BoolTest in the halving mode, which should have worse results. However, SHA-1 has an unusual output length of 160 bits and since Sýs et al. used the same set of block sizes for all the functions, none of their block sizes matched the output length or its multiple for SHA-1.
- CoolTest systematically produces stronger distinguishers than BoolTest, with the only exception being unshuffled data produced by DES, where one variant of CoolTest produced a worse distinguisher than BoolTest.
- CoolTest in both configurations managed to find non-random patterns in 14 rounds of MD5, while BoolTest succeeded with at most 13 rounds. With all other functions, the number of rounds for which the tools with these configurations can find non-randomness is the same.
- Out of the three tools that we tested, *CoolTest hist* found the strongest distinguisher for all the functions. Interestingly, for some of the functions, *CoolTest hist* and *CoolTest* `OR2` found a distinguisher resulting in exactly the same *p-value*.
- For 11 out of the 12 functions, CoolTest found non-randomness in data produced by at least the same number of rounds of the cryptographic function as the statistical batteries. For 6 of the functions, the number of rounds was higher.

Moreover, with different settings of CoolTest, we were able to find significant non-random patterns in 6 rounds of DES. The main difference in the setting leading to this result was stricter stopping criteria. Therefore, it is likely that finding a better stopping mechanism could further improve the presented results. This remains as a future work.

These experiments show that on outputs of these functions, CoolTest outperforms BoolTest and statistical batteries, with the only exception being TestU01, which detects non-randomness in up to 20 rounds of MD5.

5.4.1 Xorshift

In Chapter 3, we successfully found a good distinguisher for an instance of Xorshift64 that seems most resilient to distinguishing attacks. Based on

the observations from the chapter, we know that a good way to construct a strong distinguisher for this pseudorandom number generator is to set the block size to double the output size and find a linear combination that is always equal to zero.

However, CoolTest can not construct this kind of distinguishers, unlike BoolTest with correctly chosen settings. Therefore, we would like to find out if CoolTest works well even for data with these presumably unfavourable patterns.

Vigna in [24] proposed a methodology of generating seeds for testing pseudorandom generators and tested Xorshift64 using the TestU01 battery with his methodology. We follow this methodology. Therefore, we generate 100 seeds of the form $1 + i\lfloor 2n/100 \rfloor, 0 \leq i < 100$. Vigna discovered that the instance of Xorshift64 we are investigating fails the SmallCrush's MatrixRank test for all of the 100 seeds. SmallCrush uses approximately 200 MB of data from the function to evaluate its tests. We aim to show that CoolTest can discover non-randomness with a significantly lower amount of data.

For this experiment, we chose a different setting of CoolTest than before. We do not use the distinguisher strengthening as we would like the resulting distinguisher to be easier to interpret. As we are going to use the tool on smaller amounts of data, we decided to strengthen the greedy search and increase the number of searched paths from 300 to 500. Other parameters stay the same as in the previous experiments. We again use *significance level* $\alpha = 10^{-4}$.

We used the described seeds to generate 100 files of 10 MB of pseudorandom data. CoolTest is able to find a good distinguisher for all of the 100 data files. The *p-value* for each of the distinguishers was lower than 10^{-29}. Hence, we decided to try if CoolTest can detect non-randomness in an even lower amount of data. We again generated 100 files, this time of size 1 MB. CoolTest, with the described settings, is able to correctly indicate the non-randomness of 52 of the 100 files. This experiment shows that CoolTest is able to find good distinguishers even for low amounts of data from a function that has unfavourable non-random patterns.

5.5 Other Versions

As we presented, CoolTest is really good at finding patterns that are frequent in the examined data. However, it is by design incapable of finding patterns that are infrequent. This did not seem to be an issue during our evaluation of the method, but data with no extremely frequent patterns and, at the same time, patterns that are significantly less frequent than expected could exist. We came up with 3 approaches to modify CoolTest to find the infrequent patterns.

The first solution focuses solely on the infrequent patterns. It iteratively combines infrequent patterns discovered in brute-force search using logical `OR`. The second approach directly generalizes the distinguishers found by CoolTest. It allows an addition of negations to the distinguisher that is being extended during the greedy search. Therefore, it allows a negation of more complex parts of the polynomials and not only single variables. These negations allow us to turn a frequent distinguisher into its opposite, which is infrequent and vice versa. The third idea is to catch both infrequent and frequent patterns more precisely. It comes from the distinguisher strengthening using histograms. We use this histogram version of distinguishers throughout the whole process of CoolTest instead of trying to always find the most biased bin of all the histograms, which is what the original tool does.

We implemented these three approaches, and they are part of the submitted archive. All the approaches are fundamentally more computationally complex than CoolTest. In two of the variants, it is caused by the constructed distinguishers being more complex and difficult to construct and evaluate. In the version which iteratively uses `OR` to combine distinguishers, the complexity lies in the difficulty of computing the expected probability for the distinguishers.

As all the variants are slower than CoolTest, we decided to investigate mainly the histogram variant that offers the most general form of distinguishers. However, all the variants seem to have the potential to produce good distinguishers, and therefore, we describe them in more detail.

5.5.1 `OR` Version

As we described, this version searches only for infrequent patterns. In the brute-force search, it works the same way as CoolTest, but instead of choosing the distinguishers with the highest *Z-score*, it chooses distinguishers with the lowest *Z-score* possible, i.e. with a negative *Z-score*.

In the greedy search, this variant does not extend the distinguisher variable by variable. In each iteration, distinguishers are extended by logical `OR` with the other operand having the same form as the distinguishers in the brute-force search. A distinguisher produced by this procedure could look, for example, like this: $(x_0 + 1)x_1$ `OR` $x_2(x_3 + 1)$ `OR` x_4x_5. Similar to CoolTest, we can try all the possible candidates that are not yet part of the distinguisher. However, we try only the distinguishers that do not share any variables with the distinguisher that is being extended. This way, we can precompute the expected probabilities for all possible forms of the distinguishers and save us from unnecessary computations. Another optimisation is to use just m best distinguishers from the brute-force search as candidates for extension in the greedy search instead of all, even the frequent, distinguishers.

5.5.2 Polynomial Version

We call the second version *the polynomial version* because the distinguishers can no longer be represented as pairs of position and bit values, and we represent them in ANF. This representation allows us to add negations anywhere in the polynomial.

In the brute-force phase, the tested distinguishers are the same as in CoolTest, plus these distinguishers with any other combination of negations. It is unnecessary to use negations on the very top level of the polynomials, as these can be added during the greedy search.

The greedy search follows the same pattern as in CoolTest. The only difference is that for each distinguisher and a fixed variable that we want to add, we have 4 possible successor distinguishers. These distinguishers are created by first negating the distinguisher or the variable or both and then multiplying these two together. For example, if we are looking for the successors of x_0x_1 and we want to add the variable x_7, the possible distinguishers are $x_0x_1x_7$, $(x_0x_1+1)x_7$, $x_0x_1(x_7+1)$ and $(x_0x_1+1)(x_7+1)$. To speed this up, we could try only 2 versions instead of 4 without losing too many good results. If the sign of the *Z-score* of the polynomial and the variable that is being added are the same, negating both or neither of them will likely produce a better distinguisher than negating just one of them. Similarly, when they have different signs of the *Z-score*, it is better to negate just one of them. The logic behind it is that distinguishers with the same signs of their *Z-scores* are both frequent (or infrequent), and thus, by combining them, there is a bigger chance of preserving and reinforcing this property. On the other hand, when we combine frequent and infrequent patterns together, they are likely to at least partially cancel out the bias of each other.

5.5.3 Histogram Version

An alternative version with the biggest potential for better results in exchange for computational efficiency is based on histograms. Recall the histogram construction as a method for distinguisher strengthening. We select n bits, compute a histogram of all 2^n values of the bits and sort them based on their occurrences. Then, we linearly traverse the values and find the best splitting into two groups. In the original CoolTest, we used the histograms only in the third phase. On the other hand, in this version, use them in both brute-force and greedy search.

The brute-force search does not need to change much. CoolTest already computes all the bins and looks for the maximal ones. The histogram version needs to collect the histograms, turn them into distinguishers and find the strongest ones that are then used in the second step.

In the greedy search, we try adding all possible bits to each distinguisher. We recompute the histogram and choose the best splitting. Then, the best such distinguisher is used as a successor. For example, if we use a block size 4 and we are looking for successors of a histogram on the values of bits x_0, x_2, the candidate histograms would be constructed on bits x_0, x_1, x_2 and x_0, x_2, x_3. After the greedy search, we use the distinguisher with the highest absolute value of the *Z-score* for evaluation of the second half of the data. We do not perform any distinguisher strengthening as the distinguisher is already in the histogram form.

The issue of stopping the greedy search is, in this method, even more problematic. The *Z-score* of a successor is always at least the same as the value of its predecessor. Therefore, almost certainly, at least one successor has an improving *Z-score*, and the search never stops by itself. Moreover, some of the stopping mechanisms from CoolTest can not be applied here. For example, the computation of the *p-value* of an improvement between a distinguisher and its successor is due to the complexity of the construction almost impossible. However, validation and setting the maximal number of bits in a distinguisher can still be applied. Limiting the number of bits seems to be the most simple to use and efficient solution. Unfortunately, it is prone to an error as the distinguishers always reach the maximal number of bits, which is likely not the optimal number of bits for given data.

To compare this variant with the original CoolTest and BoolTest, we decided to use just 1 MB of data in order for this tool to run fast. For CoolTest and BoolTest, we use the same settings as in the first evaluation, with CoolTest here being the version with histograms as the strengthening method (*CoolTest hist*). The histogram variant of CoolTest (*Histogram CT*) is used with a brute-force search set to degree 3, 50 distinguishers are used in the greedy search, and the maximal number of variables in a distinguisher is set to 12. As Table 5.4 shows, the only noticeable difference between this variant and the original CoolTest is its ability to find non-randomness in 3 rounds of AES. Other than that, the tools have similar results.

These results show that this variant has some potential, and we believe that it might be useful for finding non-randomness in even smaller amounts of data. However, in order to work well, the stopping mechanism of greedy search needs to be improved, as well as the implementation needs to be optimised.

Table 5.4: The number of rounds and *p-value* of the detected distinguisher of selected cryptographic functions in which non-randomness was still detected in 1 MB data generated using the LHW strategy. The value 0 indicates any *p-value* $< 10^{-300}$.

	BoolTest		CoolTest hist		Histogram CT	
	rounds	*p-value*	rounds	*p-value*	rounds	*p-value*
AES	2	0	2	0	**3**	$\mathbf{10^{-141}}$
Blowfish	2	0	2	0	2	0
DES	4	$2 \cdot 10^{-44}$	5	$2 \cdot 10^{-69}$	5	$2 \cdot 10^{-18}$
3-DES	2	0	2	0	2	0
Grostl	2	10^{-128}	2	0	2	0
JH	6	0	6	0	6	0
Keccak	2	0	2	0	2	0
MD5	12	$4 \cdot 10^{-6}$	12	$6 \cdot 10^{-113}$	12	$7 \cdot 10^{-92}$
MD6	8	$2 \cdot 10^{-18}$	8	$2 \cdot 10^{-110}$	8	$2 \cdot 10^{-151}$
SHA-1	15	$2 \cdot 10^{-19}$	15	$2 \cdot 10^{-79}$	15	$8 \cdot 10^{-54}$
SHA-256	12	$6 \cdot 10^{-5}$	12	$7 \cdot 10^{-117}$	12	$9 \cdot 10^{-12}$
TEA	3	$2 \cdot 10^{-80}$	3	0	3	0

Chapter 6

Future Work

Even though this work provides a functional implementation of a randomness-testing tool that improves on the results of BoolTest, there is still a lot of work to do. As we showed in the last section, there are many ways in which CoolTest can be modified. There are also other options for how to modify the heuristic. For example, it should be possible to base the distinguisher on logical implications and look for the strongest set of implications between bit values. It might be interesting to see how these different modifications compare in terms of time complexity as well as the quality of distinguishers they produce.

There are also many ways in which CoolTest could be improved. We presented that when we use halving, it is a good idea to randomly shuffle blocks of data generated using CTR or LHW strategy. It would be convenient to make this method deterministic for better reproducibility of experiments. The determinization can be achieved by either creating a new data generation strategy that is better suited for the halving method or by identifying the optimal shuffling method to be used consistently.

The halving method could be further modified to produce better results. One such improvement could be used when the block size is set to a multiple of the output size of the data-producing function. When we use such block size, we look primarily for the correlation between consecutive blocks. During the process of finding a good distinguisher for the first half of the data, we could look at every consecutive tuple of blocks, even if they overlap. Therefore, we would have more tuples of blocks, and hence, potentially, we could find better distinguishers. On the other hand, it is not compatible with the random shuffling of blocks. We implemented this functionality to CoolTest, but it remains to be properly tested to see if the correlations between overlapping blocks do not affect the final distinguisher.

Another improvement when we use the halving method could be to take more than just one distinguisher to be evaluated on the second half of the data. We, of course, can not just evaluate multiple distinguishers and di-

rectly use the produced *p-values* as we did so far with just one distinguisher. However, a simple solution to this problem would be to use Bonferroni correction [28] to adjust the *significance level* adequately based on the number of distinguishers we evaluate.

The possibility of evaluating multiple distinguishers leads us to a new idea of how CoolTest could solve the problem of stopping the greedy search. In a new version of CoolTest, we could set a number of iterations of the greedy search and have it as the only stopping rule, i.e. the search would not stop even if the *Z-score* decreases between iterations. In each iteration, we would pick the best distinguisher and keep it. These best distinguishers can then be strengthened using the histogram approach or preserved as they are. Then, all the distinguishers would be evaluated on the second half of the data. With this approach, we do not need to worry about choosing the correct parameters for stopping criteria. Moreover, it would be possible to better predict how much time the tool needs to run on a given data. The same technique can be used to improve the histogram version of CoolTest, for which the issue with stopping the search is even more severe.

Besides CoolTest, another tool [25] exists that generates distinguishers in the form of histograms. The main difference is that while CoolTest generates the distinguishers in a bottom-up manner, the other tool works top-down. These two methods could likely be combined and produce more accurate distinguishers.

Conclusion

The goal of this thesis was to explore the relationship between sources of data and distinguishers that work well for such data. Based on that, we aimed to propose and implement a new approach to distinguisher construction.

We presented a construction based on first approximating the source function and then using this approximation to construct the distinguishers. We showed that this approach works really well for the pseudorandom number generator Xorshift64. The discovered distinguisher needs just tens of blocks produced by a particular variant of Xorshift64 to tell that, with a high probability, the data are not random. However, this method turned out to be ineffective for more complex functions.

Therefore, in the fourth chapter, we further explored how to construct good distinguishers. We conducted experiments that examine good distinguishers for our permutation functions and all functions on 3 and 4 bits. We provide the *Z-scores* for all combinations of functions and distinguishers on 3 bits in the attachments. Our observations led to the creation of a new tool for randomness testing similar to BoolTest, which we presented in the fifth chapter. We evaluated the tool on data produced by 12 cryptographic functions. The results showed that for 11 of the 12 functions, our tool, CoolTest, found non-randomness in the same or higher number of rounds as BoolTest and commonly used statistical batteries. For 6 of the functions, CoolTest found a distinguisher for a higher number of rounds than any of the statistical batteries. Moreover, the distinguishers that CoolTest found were for all the functions stronger than the distinguishers found by BoolTest.

The experiments showed that the distinguisher strengthening using histograms is more effective than combining distinguishers using `OR`. Therefore, we advise using CoolTest with the option `--hist`. The block size should be set to the output size (or its multiple) of the function that produced the tested data, e.g. `--block 128`. Other parameters can be left on the default setting. For better results in exchange for execution time, the parameter k can be increased (e.g. to 300 or 500) or decreased for faster but less reliable testing. In the case that the distinguishers are overfitting the first half of the data, the maximal degree of the distinguishers can be limited, e.g. by setting `--max-bits=16`. Another option is to shuffle blocks of the data

before using CoolTest, which can also help against overfitting for some types of data. A tool for shuffling the blocks is part of the attachments.

For easier interpretation of the final distinguishers, the third phase can be suppressed by setting `--patterns-combined=1`. Alternatively, BoolTest can be used as its distinguishers typically have a lower number of variables. We advise using BoolTest also in cases when some specific structure of a distinguisher is desired.

When examining low amounts of data (e.g. 1 MB or less), the Histogram version of CoolTest can likely produce better results than the main version. However, this version is significantly more computationally demanding and choosing the right parameters is not always simple. We plan to simplify the process of choosing the right parameters in the next version of CoolTest, which is outlined in the sixth chapter.

Bibliography

[1] Alberto Rotondi, Paolo Pedroni, and Antonio Pievatolo. "Basic Statistics: Hypothesis Testing". In: *Probability, Statistics and Simulation: With Application Programs Written in R*. Cham: Springer International Publishing, 2022, pp. 259–318. ISBN: 978-3-031-09429-3. DOI: 10.1007/978-3-031-09429-3_7. URL: https://doi.org/10.1007/978-3-031-09429-3_7.

[2] James Sneyd, Rachel M. Fewster, and Duncan McGillivray. "Binomial distribution". In: *Mathematics and Statistics for Science*. Cham: Springer International Publishing, 2022, pp. 561–582. ISBN: 978-3-031-05318-4. DOI: 10.1007/978-3-031-05318-4_29.

[3] Dennis Wackerly, William Mendenhall, and Richard L Scheaffer. *Mathematical statistics with applications*. Cengage Learning, 2014.

[4] Marion Videau. "Distinguishing Attacks". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 358–359. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_345. URL: https://doi.org/10.1007/978-1-4419-5906-5_345.

[5] Marek Sýs et al. "BoolTest: The Fast Randomness Testing Strategy Based on Boolean Functions with Application to DES, 3-DES, MD5, MD6 and SHA-256". In: *E-Business and Telecommunications*. Ed. by Mohammad S. Obaidat and Enrique Cabello. Cham: Springer International Publishing, 2019, pp. 123–149. ISBN: 978-3-030-11039-0. DOI: 10.1007/978-3-030-11039-0_7.

[6] Andrew Rukhin et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. Vol. 22. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2001.

[7] Robert G Brown, Dirk Eddelbuettel, and David Bauer. "Dieharder". In: *Duke University Physics Department Durham, NC* (2018), pp. 27708–0305.

[8] Pierre L'ecuyer and Richard Simard. "TestU01: AC library for empirical testing of random number generators". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (2007), pp. 1–40.

[9] Itsik Mantin and Adi Shamir. "A practical attack on broadcast RC4". In: *Fast Software Encryption: 8th International Workshop, FSE 2001 Yokohama, Japan, April 2–4, 2001 Revised Papers 8*. Springer. 2002, pp. 152–164. DOI: 10.1007/3-540-45473-X_13.

[10] Carlo Brunetta and Pablo Picazo-Sanchez. "Modelling cryptographic distinguishers using machine learning". In: *Journal of Cryptographic Engineering* 12.2 (2022), pp. 123–135. DOI: 10.1007/s13389-021-00262-x.

[11] Bart Preneel, René Govaerts, and Joos Vandewalle. "Hash functions based on block ciphers: A synthetic approach". In: *Advances in Cryptology—CRYPTO'93: 13th Annual International Cryptology Conference Santa Barbara, California, USA August 22–26, 1993 Proceedings 13*. Springer. 1994, pp. 368–378. DOI: 10.1007/3-540-48329-2_31.

[12] Claude Carlet, Yves Crama, and Peter L Hammer. *Vectorial Boolean Functions for Cryptography*. 2010. ISBN: 978-0-521-84752-0.

[13] Thomas Siegenthaler. "Correlation-immunity of nonlinear combining functions for cryptographic applications (Corresp.)" In: *IEEE Transactions on Information theory* 30.5 (1984), pp. 776–780. DOI: 10.1109/TIT.1984.1056949.

[14] Arthur F Webster and Stafford E Tavares. "On the design of S-boxes". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1985, pp. 523–534. DOI: 10.1007/3-540-39799-X_41.

[15] Thomas W Cusick and Pantelimon Stanica. *Cryptographic Boolean functions and applications*. Academic Press, 2017. ISBN: 978-0-1237-4890-4.

[16] Mitsuru Matsui. "Linear cryptanalysis method for DES cipher". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1993, pp. 386–397. DOI: 10.1007/3-540-48285-7_33.

[17] Anne Canteaut. "Fast Correlation Attack". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 450–452. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_348. URL: https://doi.org/10.1007/978-1-4419-5906-5_348.

[18] Oscar S Rothaus. "On "bent" functions". In: *Journal of Combinatorial Theory, Series A* 20.3 (1976), pp. 300–305. DOI: 10.1016/0097-3165(76)90024-8.

[19] Sihem Mesnager and Mesnager. *Bent functions*. Vol. 1. Springer, 2016. ISBN: 978-3-319-32593-4.

[20] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography: Principles and protocols*. Philadelphia, PA: Chapman & Hall/CRC, 2007. DOI: 10.1201/9781420010756.

[21] Itai Dinur and Adi Shamir. "Cube attacks on tweakable black box polynomials". In: *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*. Springer. 2009, pp. 278–299. DOI: 10.1007/978-3-642-01001-9_16.

[22] George Marsaglia. "Xorshift rngs". In: *Journal of Statistical software* 8 (2003), pp. 1–6. DOI: 10.18637/jss.v008.i14.

[23] François Panneton and Pierre L'ecuyer. "On the xorshift random number generators". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 15.4 (2005), pp. 346–361. DOI: 10.1145/1113316.1113319.

[24] Sebastiano Vigna. "An experimental exploration of Marsaglia's xorshift generators, scrambled". In: *ACM Transactions on Mathematical Software (TOMS)* 42.4 (2016), pp. 1–23. DOI: 10.1145/2845077.

[25] Rudolf Lukáč. "BoolTest with complex functions". Master's Thesis. Brno: Masaryk University, Faculty of Informatics, 2024. URL: https://is.muni.cz/th/id334/.

[26] Bikshan Chatterjee et al. "Revisiting BoolTest–On Randomness Testing Using Boolean Functions". In: *International Conference on Cryptology in India*. Springer. 2022, pp. 471–491. DOI: 10.1007/978-3-031-22912-1_21.

[27] Warren S Sarle et al. "Stopped training and other remedies for overfitting". In: *Computing science and statistics* (1996), pp. 352–360.

[28] Hervé Abdi et al. "Bonferroni and Šidák corrections for multiple comparisons". In: *Encyclopedia of measurement and statistics* 3.01 (2007), p. 2007.

Appendix A

Electronic Attachments

Besides this text, the archive of this thesis contains attachments with the following structure:

- Directory `boolean polynomials` containing implementation of permutations from the second chapter and implementation of the distinguisher construction from the third chapter.
- Directory `cooltest` with the implementation of CoolTest presented in the fifth chapter.
- Directory `cooltest_other_variants` with the implementations of other variants of CoolTest.
- Directory `shuffle` with a simple tool for shuffling blocks of data.
- File `alldis3.csv` containing the full version of Table 4.2.
- File `README.md` with a list of contents of the attachments.

Appendix B

Xorshift Polynomials

Table B.1: Polynomial representation of Xorshift64 used in Chapter 3.

bit	polynomial	bit	polynomial
0	$x_0 + x_{11} + x_{18} + x_{29}$	32	$x_1 + x_{12} + x_{19} + x_{30} + x_{32} + x_{43} + x_{50} + x_{61}$
1	$x_1 + x_{12} + x_{19} + x_{30}$	33	$x_2 + x_{13} + x_{20} + x_{31} + x_{33} + x_{44} + x_{51} + x_{62}$
2	$x_2 + x_{13} + x_{20} + x_{31}$	34	$x_3 + x_{14} + x_{21} + x_{32} + x_{34} + x_{45} + x_{52} + x_{63}$
3	$x_3 + x_{14} + x_{21} + x_{32}$	35	$x_4 + x_{15} + x_{22} + x_{33} + x_{35} + x_{46} + x_{53}$
4	$x_4 + x_{15} + x_{22} + x_{33}$	36	$x_5 + x_{16} + x_{23} + x_{34} + x_{36} + x_{47} + x_{54}$
5	$x_5 + x_{16} + x_{23} + x_{34}$	37	$x_6 + x_{17} + x_{24} + x_{35} + x_{37} + x_{48} + x_{55}$
6	$x_6 + x_{17} + x_{24} + x_{35}$	38	$x_7 + x_{18} + x_{25} + x_{36} + x_{38} + x_{49} + x_{56}$
7	$x_7 + x_{18} + x_{25} + x_{36}$	39	$x_8 + x_{19} + x_{26} + x_{37} + x_{39} + x_{50} + x_{57}$
8	$x_8 + x_{19} + x_{26} + x_{37}$	40	$x_9 + x_{20} + x_{27} + x_{38} + x_{40} + x_{51} + x_{58}$
9	$x_9 + x_{20} + x_{27} + x_{38}$	41	$x_{10} + x_{21} + x_{28} + x_{39} + x_{41} + x_{52} + x_{59}$
10	$x_{10} + x_{21} + x_{28} + x_{39}$	42	$x_{11} + x_{22} + x_{29} + x_{40} + x_{42} + x_{53} + x_{60}$
11	$x_{11} + x_{22} + x_{29} + x_{40}$	43	$x_{12} + x_{23} + x_{30} + x_{41} + x_{43} + x_{54} + x_{61}$
12	$x_{12} + x_{23} + x_{30} + x_{41}$	44	$x_{13} + x_{24} + x_{31} + x_{42} + x_{44} + x_{55} + x_{62}$
13	$x_{13} + x_{24} + x_{31} + x_{42}$	45	$x_{14} + x_{25} + x_{32} + x_{43} + x_{45} + x_{56} + x_{63}$
14	$x_{14} + x_{25} + x_{32} + x_{43}$	46	$x_{15} + x_{26} + x_{33} + x_{44} + x_{46} + x_{57}$
15	$x_{15} + x_{26} + x_{33} + x_{44}$	47	$x_{16} + x_{27} + x_{34} + x_{45} + x_{47} + x_{58}$
16	$x_{16} + x_{27} + x_{34} + x_{45}$	48	$x_{17} + x_{28} + x_{35} + x_{46} + x_{48} + x_{59}$
17	$x_{17} + x_{28} + x_{35} + x_{46}$	49	$x_{18} + x_{29} + x_{36} + x_{47} + x_{49} + x_{60}$
18	$x_{18} + x_{29} + x_{36} + x_{47}$	50	$x_{19} + x_{30} + x_{37} + x_{48} + x_{50} + x_{61}$
19	$x_{19} + x_{30} + x_{37} + x_{48}$	51	$x_{20} + x_{31} + x_{38} + x_{49} + x_{51} + x_{62}$
20	$x_0 + x_{18} + x_{20} + x_{31} + x_{38} + x_{49}$	52	$x_{21} + x_{32} + x_{39} + x_{50} + x_{52} + x_{63}$
21	$x_1 + x_{19} + x_{21} + x_{32} + x_{39} + x_{50}$	53	$x_{22} + x_{40} + x_{53}$
22	$x_2 + x_{20} + x_{22} + x_{33} + x_{40} + x_{51}$	54	$x_{23} + x_{41} + x_{54}$
23	$x_3 + x_{21} + x_{23} + x_{34} + x_{41} + x_{52}$	55	$x_{24} + x_{42} + x_{55}$
24	$x_4 + x_{22} + x_{24} + x_{35} + x_{42} + x_{53}$	56	$x_{25} + x_{43} + x_{56}$
25	$x_5 + x_{23} + x_{25} + x_{36} + x_{43} + x_{54}$	57	$x_{26} + x_{44} + x_{57}$
26	$x_6 + x_{24} + x_{26} + x_{37} + x_{44} + x_{55}$	58	$x_{27} + x_{45} + x_{58}$
27	$x_7 + x_{25} + x_{27} + x_{38} + x_{45} + x_{56}$	59	$x_{28} + x_{46} + x_{59}$
28	$x_8 + x_{26} + x_{28} + x_{39} + x_{46} + x_{57}$	60	$x_{29} + x_{47} + x_{60}$
29	$x_9 + x_{27} + x_{29} + x_{40} + x_{47} + x_{58}$	61	$x_{30} + x_{48} + x_{61}$
30	$x_{10} + x_{28} + x_{30} + x_{41} + x_{48} + x_{59}$	62	$x_{31} + x_{49} + x_{62}$
31	$x_0 + x_{11} + x_{18} + x_{29} + x_{31} + x_{42} + x_{49} + x_{60}$	63	$x_{32} + x_{50} + x_{63}$

Appendix C

Evaluation of All Distinguishers – 4 Bits

The following tables show the same data as presented in the forth chapter, but for functions on 4 bits instead of 3 bits. The general patterns are here the same as for the functions on 3 bits.

Table C.1: Description of groups of Boolean functions on 4 bits by the number of 1s they return and the average *p-values* of the groups.

	Returned 1s	Avg. *p-value*
Group 1	0/16	1.00
Group 2	1/15	0.40
Group 3	2/14	0.32
Group 4	3/13	0.28
Group 5	4/12	0.26
Group 6	5/11	0.25
Group 7	6/10	0.25
Group 8	7/9	0.25
Group 9	8	0.29

Table C.2: The counts of classes of functions for which distinguishers from different groups have a given *Z-score*.

Z-sc	Group1	Group2	Group3	Group4	Group5	Group6	Group7	Group8	Group9
15.5	0	1	0	0	0	0	0	0	0
14.5	0	15	0	0	0	0	0	0	0
13.4	0	120	0	0	0	0	0	0	0
12.4	0	680	0	0	0	0	0	0	0
11.4	0	3060	0	0	0	0	0	0	0
10.6	0	0	17	0	0	0	0	0	0
10.3	0	11628	0	0	0	0	0	0	0
9.8	0	0	224	0	0	0	0	0	0
9.3	0	38760	0	0	0	0	0	0	0
9	0	0	1575	0	0	0	0	0	0
8.3	0	116280	7840	153	0	0	0	0	0
7.7	0	0	0	1768	0	0	0	0	0
7.6	0	0	30940	0	0	0	0	0	0
7.2	0	319770	0	0	0	0	0	0	0
7	0	0	0	10920	0	0	0	0	0
6.9	0	0	0	0	969	0	0	0	0
6.8	0	0	102816	0	0	0	0	0	0
6.4	0	0	0	47775	9792	0	0	0	0
6.2	0	817190	0	0	0	0	0	0	0
6	0	0	298452	0	0	0	0	0	0
5.9	0	0	0	0	0	4845	0	0	0
5.8	0	0	0	165620	53040	0	0	0	0
5.4	0	0	0	0	0	42636	0	0	0
5.3	0	0	775200	0	0	0	0	0	0
5.2	0	1961256	0	0	203840	0	20349	0	0
5.1	0	0	0	482664	0	0	0	0	0
4.9	0	0	0	0	0	201960	0	0	0
4.6	0	0	0	0	621075	0	155040	0	0
4.5	0	0	1831410	1225224	0	0	0	74613	0
4.3	0	0	0	0	0	680680	0	0	0
4.1	0	4457400	0	0	0	0	639540	0	0
4	0	0	0	0	0	0	1589952	488376	490314
3.8	0	0	3979360	2771340	0	1821820	0	0	0
3.6	0	0	0	0	0	0	1884960	0	0
3.5	0	0	0	0	3539536	0	0	2479671	2728704
3.2	0	0	0	5668650	0	4099095	0	0	0
3.1	0	9657700	0	0	0	0	6467395	0	0
3	0	0	8008462	0	0	0	0	7908978	8372160
2.9	0	0	0	0	7001280	0	0	0	0
2.7	0	0	0	0	0	13327743	0	0	0
2.6	0	0	0	10581480	0	0	16589760	0	0
2.5	0	0	0	0	0	0	0	18142740	18604800
2.3	0	0	14976864	0	25508925	0	0	0	0
2.2	0	0	0	0	0	30249120	0	0	0
2.1	0	20058300	0	0	0	0	32191005	0	0
2	0	0	0	0	0	0	0	33021072	33256080
1.9	0	0	0	48527843	0	0	0	0	0
1.7	0	0	0	0	51059840	0	0	0	0
1.6	0	0	0	0	0	51079050	0	0	0
1.5	0	0	93865415	0	0	0	50758400	50501724	50409216
1.3	0	0	0	80545218	0	0	0	0	0
1.2	0	0	0	0	74202144	0	0	0	0
1.1	0	0	0	0	0	70527050	0	0	0
1	0	185539275	0	0	0	0	68322150	67126059	66745536
0.8	0	0	116486720	0	0	0	0	0	0
0.6	0	0	0	98508540	89427072	0	0	0	0
0.5	0	0	0	0	0	84063980	80832960	79081002	78524160
0	300540195	77558760	60174900	52003000	47322730	44442216	42678636	41715960	41409225

Appendix D

Generation of Data

We used CryptoStreams[1] for generating data from cryptographic functions with a limited number of rounds. The following examples are JSON configurations for generating 1 MB of data from 14 rounds of SHA2 and 3 rounds of DES with input generated using the LHW strategy.

```
{
    "notes": "hash",
    "file_name": "SHA2_14r_lhw_1MB",
    "seed": "1fe40505e131963c",
    "tv_size": 32,
    "tv_count": 31250.0,
    "stream": {
        "type": "hash",
        "algorithm": "SHA2",
        "input_size": 32,
        "source": {
            "type": "hw_counter",
            "hw": 4
        },
        "round": 14,
        "hash_size": 32
    }
}

{
    "notes": "block cipher",
    "file_name": "SINGLE-DES_6r_lhw_1MB",
    "seed": "1fe40505e131963c",
    "tv_size": 8,
    "tv_count": 125000,
```

[1] https://github.com/crocs-muni/CryptoStreams

```
    "stream": {
        "type": "block",
        "init_frequency" : "only_once",
        "algorithm": "SINGLE-DES",
        "round": 6,
        "block_size": 8,
        "plaintext": {
            "type": "hw_counter",
            "hw": 6
        },
        "key_size": 7,
        "key": {
            "type" : "pcg32_stream"
        },
        "encryption_mode": true,
        "mode": "ECB",
        "iv": {
            "type": "false_stream"
        }

    }
}
```