# VBF Library Reference Manual

***Release 1.0***

**Jose Antonio Alvarez Cubero**

**May 21, 2020**

## Contents

VBF

# User Manual and Analysis of Cryptographic algorithms



WRITTEN BY: JOSÉ ANTONIO ÁLVAREZ CUBERO

Table of Contents:

# 1 Preface



## 1.1 Authors


Written by: **Jose Antonio Alvarez Cubero** <jaacubero@gmail.com>
Revised by: **Pedro J. Zufiria** <pedro.zufiria@upm.es>

## 1.2 Dedications

"An appreciative thank you to my wife Isabel and my daughter Sofia for their support. I would like to thank my amazing family, without whom, none of the things I do would matter." - **Jose Antonio Alvarez Cubero**

## 1.3 Acknowledgements

Special thanks to Pedro J. Zufiria, Full Professor at Universidad Politecnica de Madrid (UPM) who supported the realization of this book. We would like to thank you for your influence and support both prior and during the book elaboration.

We are truly grateful to the ACM Transactions on Mathematical Software Journal editors and reviewers for their contributions.

## 1.4 Book Writing Methodology

The GitHub Pages (https://pages.github.com/) were used for writing this book. GitHub Pages are public webpages hosted and published through GitHub. The html pages were built with Sphinx using a theme provided by Read the Docs (https://readthedocs.org/). Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license. The Read the Docs theme features beautiful typography and a nice blue color scheme. It looks great on mobile, and provides a menu of all the pages on the left-hand side.

## 1.5 Who Should Read This Book?

### 1.5.1 Expected Audience

The intended audience for this book is those with a general need to understand how to install and use the VBF library. While experts in Cryptography may experience the largest benefit from this content, the materials included herein may be of use to a much wider audience.

## 1.6 Book Objectives

The objectives of this book are those that allows the reader to:

- Give a brief explanation of the installation of VBF library.
- Provide a thematic index of all of VBF's features.
- Show many examples that illustrate their use.

## 1.7 Organization of this Book

### 1.7.1 Section 1: Introduction

The introduction provides a brief description of the functions available in the VBF library and gives a general overview of the VBF's programming interface and conventions. It also illustrates how to obtain and install the library, and introduces some mathematical preliminaries and notations.

### 1.7.2 Section 2: Using the library

This chapter describes how to compile programs that use VBF and how to evaluate new algorithms.

### 1.7.3 Section 3: Representations and characterizations

This chapter presents a review of theory relevant to the study of the typical forms of Vector Boolean function representations and chacterizations. We will consider representations those that uniquely represents a Vector Boolean function. Characterizations does not uniquely determine the Vector Boolean function in contrast to the previous matrices but provide some useful information in the context of cryptography.

Representations included in this chapter are the Truth Table (TT), the polynomials in Algebraic Normal Form (Pol) and ANF Table (ANF), the Image (Char), Component functions Truth Table(LTT), Sequence vectors of Component functions CTT, the Trace Representation (Trace) and Affine function Representation. A definition for all these representations are given and the relationships among them and their various properties are also discussed.

Characterizations such as Linear Profile (LP), Differential Profile (DP), Autocorrelation Spectrum (AC), Linear Structures (LS) are introduced. A definition for all these representations are given and the relationships among them and the above representations and their various properties are also discussed.

The basic concepts of linear and differential cryptanalysis are introduced in terms of the Linear Profile and Differential Profile, together with other properties related with these attacks, such as linear potential, differential potential, linear or differential relations associated with a specific value.

Affine equivalence analysis of Boolean functions by means of VBF library is described. It is showed how to obtain the Frequency distribution of the absolute values of the Walsh Spectrum and of the Autocorrelation Spectrum.

It is possible to check randomness of a Vector Boolean function outputs with VBF by means of its cycle structure, and the analysis of the presence of fixed points or negated fixed points.

Finally, some other representations useful in block ciphers are described such as the Permutation Vector (Per), Expansion and Compression DES permutations and DES-like S-box representations. The description of each representation and characterization is complemented with the description of the methods in VBF related to them. Most of the member functions of V BF have an in-line definition, for instance: `void TT(NTL::mat_GF2& X, VBF& F)` is also defined as inline `NTL::mat_GF2 TT(VBF& F)`.

### 1.7.4 Section 4: Cryptographic Criteria

This chapter defines some properties relevant for cryptographic applications and explains how to use the package to compute them. They are defined in relation to the representation or transform from which they are derived.

Those properties are criteria or those which provide useful information in cryptanalysis. Among the criteria we find nonlinearity, r-th order nonlinearity, linearity distance, balancedness, correlation immunity, resiliency (i.e. balancedness and correlation immunity), propagation criterion, global avalanche criterion, algebraic degree and algebraic immunity. Other properties described are the maximum possible nonlinearity or the maximum possible linearity distance achievable by a Vector Boolean function with the same number of inputs, the type of function in terms of nonlinearity.

### 1.7.5 Section 5: Constructions for Vector Boolean functions

In this chapter, some basic constructions for Vector Boolean functions supported by the VBF class are described. Some of them correspond to secondary constructions, which build (n,m) variable vector Boolean functions from (n,m) variable ones (with n  n,m  m). The direct sum has been used to construct resilient and bent Boolean functions [Carlet:04]. The concatenation can be used to obtain resilient functions or functions with maximal nonlinearity. The concatenation of polynomials in ANF can be used to obtain functions of high nonlinearity with n variables from functions with high nonlinearity with n variables (n < n). Adding coordinate functions and bricklayering are constructions used to build modern ciphers such as CAST [CAST:256], DES [DES:77] and AES [DaemenR:02]. Additionally, VBF provides operations for identification if two vector Boolean functions are equal, the sum of two vector Boolean functions, the composition of two vector Boolean functions and the inverse of a Vector Boolean function.

### 1.7.6 Section 6: Analysis of AES competition cryptographic algorithms

In January 1997, the US National Institute of Standards and Technology (NIST) announced the start of an initiative to develop a new encryption standard: the AES. The AES selection process was open in which 15 candidates were accepted for the first evaluation round and 5 finalists were announced in the second round. On October 2, 2000, NIST officially announced that Rijndael would become the AES. In this chapter, a number of cryptographic algorithms from the AES (Advanced Encryption Standard) candidates accepted for the first evaluation round process are analysed.

### 1.7.7 Section 7: Analysis of CRYPTEC project cryptographic algorithms

CRYPTREC is the Cryptography Research and Evaluation Committees set up by the Japanese Government to evaluate and recommend cryptographic techniques for government and industrial use. It is comparable in many respects to the European Union's NESSIE project and to the Advanced Encryption Standard process run by NIST in the U.S.. In this chapter, certain cryptographic algorithms from CRYPTEC project candidates are analysed.

### 1.7.8 Section 8: Analysis of NESSIE project cryptographic algorithms

The NESSIE call includes a request for a broad set of algorithms providing date confidentiality, data authentication, and entity authentication. These algorithms include block ciphers, stream ciphers, hash functions, MAC algorithms, digital signature schemes, and public-key encryption and identification schemes. In this chapter, several cryptographic algorithms from NESSIE (New European Schemes for Signature, Integrity, and Encryption) research project candidates are analysed.

### 1.7.9 Section 9: Analysis of other cryptographic algorithms

There are some block ciphers that are relevant either because its wide use or because of its importance in the development of cryptanalysis techniques. In this chapter, some cryptographic algorithms from other block ciphers are analysed.

### 1.7.10 Section 10: Design of cryptographically robust Vector Boolean functions

This chapter is devoted to the practical implementation of robust Boolean functions as fundamental components for S-box design. First, the design of such functions is formalized as a Multi-Objective Combinatorial Optimization (MOCO) problem.

### 1.7.11 Section 11: FAQs

Frequently Asked Questions.

### 1.7.12 Section 12: Bibliography

This chapter includes all citations referenced along the Reference Manual.

### 1.7.13 Section 13: Appendix

This chapter includes example programs using VBF library.

# 2 Introduction



The Vector Boolean Function Library (VBF) is a collection of C++ classes designed for analyzing Vector Boolean Functions (functions that map a Boolean vector to another Boolean vector) from a cryptographic perspective. This implementation uses the NTL library from Victor Shoup, modifying some of the general purpose modules of this library (to make it better suited to cryptography), and adding new modules that complement the existing ones. The class representing a Vector Boolean Function can be initialized by several data structures such as Truth Table, Trace representation, Algebraic Normal Form (ANF) among others. The most relevant cryptographic criteria for both block and stream ciphers can be evaluated with VBF. It allows to obtain some interesting cryptologic characterizing features such as linear structures, frequency distribution of the absolute values of the Walsh Spectrum or Autocorrelation Spectrum, among others. In addition, operations such as equality checking, composition, inversion, sum, direct sum, concatenation, bricklayering (parallel application of Vector Boolean Functions as employed in Rijndael cipher), and adding coordinate functions of two Vector Boolean Functions can be executed.

## 2.1 Functions available in VBF

The library covers a wide range of topics for analyzing cryptographic properties of Vector Boolean Functions. Methods are available for the following areas:

1. Vector Boolean Function representations and characterizations
2. Cryptographic criteria calculation
3. Constructions and operations over Vector Boolean functions

The use of these methods is described in this manual. Each chapter provides detailed definitions of the methods, followed by example programs.

## 2.2 Conventions used in this manual

This manual contains many examples which can be typed at the keyboard. A command entered at the terminal is shown like this,

```
$ command
```

The first character on the line is the terminal prompt, and should not be typed. The dollar sign $ is used as the standard prompt in this manual, although some systems may use a different character.

The examples assume the use of the GNU operating system. There may be minor differences in the output on other systems. The commands for setting environment variables use the Bourne shell syntax of the standard GNU shell (bash).

## 2.3 Software implementation

The package included consists of:

1. Derived classes inherited from NTL base classes which add new functions on top of them:

```
pol.h, vbf_GF2EX.h, vbf_GF2X.h, vbf_ZZ.h, vbf_mat_GF2.h, vbf_mat_RR.h, vbf_mat_ZZ.h,␣
↪vbf_tools.h,
vbf_vec_GF2.h, vbf_vec_GF2E.h, vbf_vec_RR.h, vbf_vec_ZZ.h, vec_pol.h
```

2. Main class (*VBF.h*) with the functions and other *.h* files,
3. A makefile to ease the compilation of example (*Makefile*),
4. A set of files associated with the decimal representation of KASUMI [KASUMI:05] S-boxes (S7.dec and S9.dec).

The Output files can be found within "KASUMI Analysis" in the "Analysis of other cryptographic algorithms" menu at **'http://vbf.rtfd.io'**_.

## 2.4 System requirements

The VBF library can be easily installed in a matter of minutes on just about any platform, including virtually any 32- or 64-bit machine running any flavor of Unix, as well as PCs running Windows, and Macintoshes. VBF achieves this portability by avoiding esoteric C++ features, and by avoiding assembly code; it should therefore remain usable for years to come with little or no maintenance, even as processors and operating systems continue to change and evolve.

## 2.5 Installation

We are going to illustrate the installation of the package in an Unix or Unix-like platforms (including Linux distributions).

1. **Download the lastest version of the library** from VBF source code URL and place it in the working directory. You should see the example program, input files and the "*.h" files.
2. **Obtain NTL library source code**. To obtain the source code and documentation for NTL, download ntl-xxx.tar.gz from Download NTL library, placing it in a different directory.
3. **Run the configuration script**. Working in the directory where you placed the NTL library, do the following (here, "xxx" denotes the desired version number of NTL; any version of NTL can be employed):

```
$ cd ntl-xxx/src
$ ./configure
```

The execution of *configure* generates the file "makefile" and the file "../include/NTL/config.h", based upon the values assigned to the variables on the command line. In the example above no arguments were employed. The most important variables are: "CC" for choosing the C compiler, "CXX" for choosing the C++ compiler, "PREFIX" for choosing the directory in which to install NTL library components.

---

**Disable GMP**

If you really do not want to use GMP, you can pass the option *NTL_GMP_LIP=off* to configure. More information on A Tour of NTL: Obtaining and Installing NTL for UNIX

---

4. **Build NTL:**

```
$ make
$ make check
$ make install
```

The *make* execution in the directory *src/* compiles all the source files and creates a library *ntl.a* in the same directory. Some testing programs are run by means of the command *make check*. Lastly, *make install* performs among other actions the copy of the library file *ntl.a* into */usr/local/lib/libntl.a* by default. It is not necessary to execute *make check* in each NTL building as it takes a long time. In order to execute *make install*, it is necessary to have privileged user permissions as some directories creation, file deletion, changes of file attributes, and copies of files are done.

Do not forget to use an account with appropriate permissions: *root* for instance.

## 2.6 Preliminaries

The mathematical theory of Vector Boolean Functions starts with the formal definition of vector spaces whose elements (vectors) have binary elements. Let $< \mathrm{GF}(2), +, \cdot >$ be the finite field of order 2, where $\mathrm{GF}(2) = \mathbb{Z}_2 = \{0, 1\}, '+'$ the 'integer addition modulo 2' and $'\cdot'$ the 'integer multiplication modulo 2'. $\mathrm{V_n}$ is the vector space of n-tuples of elements from $\mathrm{GF}(2)$. The *direct sum* of $\mathbf{x} \in \mathrm{V_{n_1}}$ and $\mathbf{y} \in \mathrm{V_{n_2}}$ is defined as $\mathbf{x} \oplus \mathbf{y} = (x_1, \ldots, x_{n_1}, y_1, \ldots, y_{n_2}) \in \mathrm{V_{n_1+n_2}}$. The *inner product* of $\mathbf{x}, \mathbf{y} \in \mathrm{V_n}$ is denoted by $\mathbf{x} \cdot \mathbf{y}$, and the inner product of real vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is denoted by $\langle \mathbf{x}, \mathbf{y} \rangle$.

One can now define binary functions between this type of vector spaces, whose linearity analysis (for robustness-against-attacks purposes) will become very important. $f : \mathrm{V_n} \to \mathrm{GF}(2)$ is called a *Boolean function* and $\mathcal{F}_n$ is the set of all Boolean functions on $\mathrm{V_n}$. $\mathcal{L}_n$ is the set of all linear Boolean functions on $\mathrm{V_n} : \mathcal{L}_n = \{l_{\mathbf{u}} \, \forall \, \mathbf{u} \in \mathrm{V_n} \mid l_{\mathbf{u}}(\mathbf{x}) = \mathbf{u} \cdot \mathbf{x}\}$ and $\mathcal{A}_n$ is the set of all affine Boolean functions on $\mathrm{V_n}$.

It is possible to characterize Boolean functions via alternative and very useful associated mappings. In the following, some of these mappings are presented. The real-valued mapping $\chi_{\mathbf{u}}(\mathbf{x}) = (-1)^{\sum_{i=1}^{i=n} u_i x_i} = (-1)^{\mathbf{u} \cdot \mathbf{x}}$ for $\mathbf{x}, \mathbf{u} \in \mathrm{V_n}$ is called a *character*. The character form of $f \in \mathcal{F}_n$ is defined as $\chi_f(\mathbf{x}) = (-1)^{f(\mathbf{x})}$. The Truth Table of $\chi_f$ is called as the *(1,-1)-sequence vector* or *sequence vector* of $f$ and is denoted by $\xi_f \in \mathbb{R}^{2^n}$.

Let $f \in \mathcal{F}_n$ be a Boolean function; the *Walsh Transform* of $f$ at $\mathbf{u} \in \mathrm{V_n}$ is the n-dimensional Discrete Fourier Transform and can be calculated as follows:

$$\hat{\chi}_f(\mathbf{u}) = \langle \xi_f, \xi_{l_{\mathbf{u}}} \rangle = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{f(\mathbf{x}) + \mathbf{u}\mathbf{x}}$$

The *autocorrelation* of $f \in \mathcal{F}_n$ with respect to the shift $\mathbf{u} \in \mathrm{V_n}$ is a measure of the statistical dependency among the involved variables (indicating robustness against randomness-based attacks). It is the cross-correlation of $f$ with itself, denoted by $r_f(\mathbf{u}) : \mathrm{V_n} \to \mathbb{Z}$ and defined by[1]:

$$r_f(\mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} \chi_f(\mathbf{x}) \chi_f(\mathbf{x} + \mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{f(\mathbf{x}) + f(\mathbf{u} + \mathbf{x})}$$

The *directional derivative* of $f \in \mathcal{F}_n$ in the direction of $\mathbf{u} \in \mathrm{V_n}$ is defined by:

$$\Delta_{\mathbf{u}} f(\mathbf{x}) = f(\mathbf{x} + \mathbf{u}) + f(\mathbf{x}), \;\; \mathbf{x} \in \mathrm{V_n}$$

We shall call the linear kernel of $f$ the set of those vectors $\mathbf{u}$ such that $\Delta_{\mathbf{u}} f$ is a constant function. The linear kernel of any Boolean function is a subspace of $\mathrm{V_n}$. Any element $\mathbf{u}$ of the linear kernel of $f$ is said to be a linear structure of $f$.

Given $f \in \mathcal{F}_n$, a nonzero function $g \in \mathcal{F}_n$ is called an *annihilator* of $f$ if $fg = 0$.

We now extend the scope of the study by considering functions between any pair of binary-valued vector spaces. $F : \mathrm{V_n} \to \mathrm{V_m}, \; F(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))$ is called a *Vector Boolean function* and $\mathcal{F}_{n,m}$ is the set of all Vector

[1] Most authors omit the factor $\frac{1}{2^n}$

Boolean Functions $F : \mathrm{V_n} \to \mathrm{V_m}$. Each $f_i : \mathrm{V_n} \to \mathrm{GF}(2)\ \forall\, i \in \{1, \ldots, m\}$ is a coordinate function of $F$. The *indicator function* of $F \in \mathcal{F}_{n,m}$, denoted by $\theta_F : \mathrm{V_n} \times \mathrm{V_m} \to \{0, 1\}$, is defined in [ChabaudV:94] as:

$$\theta_F(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{y} = F(\mathbf{x}) \\ 0 & \text{if } \mathbf{y} \neq F(\mathbf{x}) \end{cases}$$

Again, several mappings associated with a Vector Boolean Functions can be defined, in similar terms to the binary functions case. Hence, the character form of $(\mathbf{u}, \mathbf{v}) \in \mathrm{V_n} \times \mathrm{V_m}$ can be defined as follows: $\chi_{(\mathbf{u},\mathbf{v})}(\mathbf{x}, \mathbf{y}) = (-1)^{\mathbf{u}\cdot\mathbf{x}+\mathbf{v}\cdot\mathbf{y}}$. Similarly, let $F \in \mathcal{F}_{n,m}$ be a Vector Boolean function; its *Walsh Transform* is the two-dimensional Walsh Transform defined by:

$$\hat{\theta}_F(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{x}\in\mathrm{V_n}} \sum_{\mathbf{y}\in\mathrm{V_m}} \theta_F(\mathbf{x}, \mathbf{y})\chi_{(\mathbf{u},\mathbf{v})}(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{x}\in\mathrm{V_n}} (-1)^{\mathbf{u}\mathbf{x}+\mathbf{v}F(\mathbf{x})}$$

Also, the *autocorrelation* of $F \in \mathcal{F}_{n,m}$ with respect to the shift $(\mathbf{u}, \mathbf{v}) \in \mathrm{V_n} \times \mathrm{V_m}$ is the cross-correlation of $F$ with itself, denoted by $r_F(\mathbf{u}, \mathbf{v}) : \mathrm{V_n} \times \mathrm{V_m} \to \mathbb{Z}$, so that [fse-Nyberg:94]:

$$r_F(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{x}\in\mathrm{V_n}} \chi_{\mathbf{v}F}(\mathbf{x} + \mathbf{u})\chi_{\mathbf{v}F}(\mathbf{x}) = \sum_{\mathbf{x}\in\mathrm{V_n}} (-1)^{\mathbf{v}F(\mathbf{x}+\mathbf{u})+\mathbf{v}F(\mathbf{x})}$$

Let $F \in \mathcal{F}_{n,m}$ and $\mathbf{u} \in \mathrm{V_n}$, then the *difference Vector Boolean function* of $F$ in the direction of $\mathbf{u} \in \mathrm{V_n}$, denoted by $\Delta_{\mathbf{u}}F \in \mathcal{F}_{n,m}$ is defined as follows: $\Delta_{\mathbf{u}}F(\mathbf{x}) = F(\mathbf{x} + \mathbf{u}) + F(\mathbf{x}),\ \mathbf{x} \in \mathrm{V_n}$. If the following equality is satisfied: $\Delta_{\mathbf{u}}F(\mathbf{x}) = \mathbf{c},\ \ \mathbf{c} \in \mathrm{V_n}\ \forall\, \mathbf{x} \in \mathrm{V_n}$ then $\mathbf{u} \in \mathrm{V_n}$ is called a linear structure of $F$.

Finally, we define the simplifying notation for the maximum of the absolute values of a set of real numbers $\{a_{\mathbf{uv}}\}_{\mathbf{u},\mathbf{v}}$, characterized by vectors $\mathbf{u}$ and $\mathbf{v}$, as: $\max\,(a_{\mathbf{uv}}) = \max_{(\mathbf{u},\mathbf{v})}\,\{|a_{\mathbf{uv}}|\}$. Using the same simplifying notation, we can define the $\overset{*}{\max}\,(\cdot)$ operator on a set of real numbers $\{a_{\mathbf{uv}}\}_{\mathbf{u},\mathbf{v}}$, as: $\overset{*}{\max}\,(a_{\mathbf{uv}}) = \max_{(\mathbf{u},\mathbf{v})\neq(\mathbf{0},\mathbf{0})}\{|a_{\mathbf{uv}}|\}$. This notation will be used in some criteria definitions.

## 2.7 Design Philosophy

The core of VBF library is the VBF class which represents Vector Boolean Functions whose data members and member functions make use of the NTL modules listed in Table NTL Modules. However, some new cryptography-related member functions were added to the previous modules. Besides, new modules which are not present in NTL, are defined and they are listed in Table New Modules.

| NTL modules used in VBF | |
|---|---|
| **CLASS NAME** | **DESCRIPTION** |
| `GF2` | Galois Field of order 2 denoted by $\mathrm{GF}(2)$ |
| `vec_GF2` | Vectors over $\mathrm{GF}(2)$ |
| `mat_GF2` | Matrices over $\mathrm{GF}(2)$ |
| `RR` | Arbitrary-precision floating point numbers |
| `vec_RR` | Vectors over reals |
| `mat_RR` | Matrices over reals |
| `ZZ` | Signed, arbitrary length integers |
| `vec_ZZ` | Vectors over integers |
| `mat_ZZ` | Matrices over integers |
| `GF2X` | Implements polynomial arithmetic modulo 2 |
| `GF2E` | Polynomials in $F_2[X]$ modulo a polynomial $P$ |
| `GF2EX` | Polynomials over GF2E |
| `vec_GF2E` | Vectors over GF2E |

Note that the modulus *P* in GF2E may be any polynomial with degree greater than 0, not necessarily irreducible. Objects of the class GF2E are represented as a GF2X of degree less than the degree of *P*. GF2EX can be used, for example, for arithmetic in $\mathrm{GF}(2^{\mathrm{n}})[\mathrm{X}]$.

| New modules created for VBF | |
|---|---|
| **CLASS NAME** | **DESCRIPTION** |
| `pol` | Polynomial in ANF of a Boolean Function |
| `vec_pol` | Polynomials in ANF of a Vector Boolean Function |

The main file in the library, called *VBF.h* has the definitions of the objects described in the next chapters.

# 3 Using the library

- *An Example Program*
- *Compiling*
- *How to evaluate new algorithms*

This chapter describes how to compile programs that use VBF and how to evaluate new algorithms.

## 3.1 An Example Program

The following program demonstrates the use of the library to analyze Vector Boolean Functions represented in decimal representation of its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::vec_long vec_F;
   NTL::vec_ZZ   c;
   NTL::mat_GF2 A, T;
   NTL::mat_ZZ  W, LP, DP;
   NTL::mat_ZZ  Ac;
   long a;
   int  n;
   char file[33];

   // Load VBF definitions

   sprintf(file,"%s.dec",argv[1]);
   ifstream input(file);
   if(!input) {
      cerr << "Error opening " << file << endl;
      return 0;
   }
   input >> vec_F;
   n = atoi(argv[2]);
   F.putDecTT(vec_F,n);
```

(continues on next page)

(continued from previous page)

```
   input.close();

   sprintf(file,"%s.anf",argv[1]);
   ofstream output(file);
   if(!output) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   A = ANF(F);
   cout << "Argument Dimension = " << F.n() << endl;
   cout << "Argument space has " << F.spacen() << " elements."<< endl;
   cout << "Image Dimension = " << F.m() << endl;
   cout << "Image space has " << F.spacem() << " elements." << endl << endl;
   cout << "Writing Algebraic Normal Form to file: " << file << endl;
   cout << "[Columns = Image components]" << endl;
   output << A << endl;
   output.close();

   sprintf(file,"%s.tt",argv[1]);
   ofstream output1(file);
   if(!output1) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   T = TT(F);
   cout << endl << "Writing Truth Table to file: " << file << endl;
   cout << "[Columns = Image components]" << endl;
   output1 << T << endl;
   output1.close();

   sprintf(file,"%s.wal",argv[1]);
   ofstream output2(file);
   if(!output2) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   W = Walsh(F);
   cout << endl << "Writing Walsh Spectrum to file: " << file <<endl;
   output2 << W << endl;
   output2.close();

   sprintf(file,"%s.lp",argv[1]);
   ofstream output3(file);
   if(!output3) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   LP = LAT(F);
   cout << endl << "Writing Linear Profile to file: " << file << endl;
   cout << "[To normalize divide by " << LP[0][0] << "]" << endl;
   output3 << LP << endl;
   output3.close();
```

(continues on next page)

(continued from previous page)

```
   sprintf(file,"%s.dp",argv[1]);
   ofstream output4(file);
   if(!output4) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   DP = DAT(F);
   cout << endl << "Writing Differential Profile to file: " << file << endl;
   cout << "[To normalize divide by " << DP[0][0] << "]" << endl;
   output4 << DP << endl;
   output4.close();

   sprintf(file,"%s.pol",argv[1]);
   ofstream output5(file);
   if(!output5) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   cout << endl << "Writing the polynomials in ANF to file: " << file << endl;
   Pol(output5,F);
   output5.close();

   sprintf(file,"%s.ls",argv[1]);
   ofstream output6(file);
   if(!output6) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   cout << endl << "Writing Linear structures to file: " << file << endl;
   LS(output6,F);
   output6.close();

   sprintf(file,"%s.ac",argv[1]);
   ofstream output7(file);
   if(!output7) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   Ac = AC(F);
   cout << endl << "Writing Autocorrelation Spectrum to file: " << file << endl;
   output7 << Ac << endl;
   output7.close();

   sprintf(file,"%s.cy",argv[1]);
   ofstream output8(file);
   if(!output8) {
      cerr << "Error opening " << file << endl;
      return 0;
   }

   cout << endl << "Writing Cycle Structure to file: " << file << endl;
   printCycle(output8,F);
   output8.close();
```

(continues on next page)

(continued from previous page)

```
  cout << endl <<  "Nonlinearity: " << nl(F) << endl;
  nlr(a,F,2);
  cout << "Second order Nonlinearity: " << a << endl;
  cout << "Linearity distance: " << ld(F) << endl;
  cout << "Algebraic degree: " << deg(F) << endl;
  cout << "Algebraic immunity: " << AI(F) << endl;
  cout << "Absolute indicator: " << maxAC(F) << endl;
  cout << "Sum-of-squares indicator: " << sigma(F) << endl;
  cout << "Linear potential: " << lp(F) << endl;
  cout << "Differential potential: " << dp(F) << endl;
  cout << "Maximum Nonlinearity (if n is even): " << nlmax(F) << endl;
  cout << "Maximum Linearity distance: " << ldmax(F) << endl;

  int type;
  typenl(type, F);

  if (type == BENT) {
    cout << "It is a bent function" << endl;
  } else if (type == ALMOST_BENT) {
    cout << "It is an almost bent function" << endl;
  } else if (type == LINEAR) {
    cout << "It is a linear function" << endl;
  }

  cout << "The fixed points are: " << endl;
  cout << fixedpoints(F) << endl;
  cout << "The negated fixed points are: " << endl;
  cout << negatedfixedpoints(F) << endl;
  cout << "Correlation immunity: " << CI(F) << endl;
  if (Bal(F))
  {
    cout << "It is a balanced function" << endl;
  } else
  {
    cout << "It is a non-balanced function" << endl;
  }
  cout << "The function is PC of degree " << PC(F) << endl;

 return 0;
}
```

A set of files associated with the decimal representation of KASUMI S-boxes (S7.dec and S9.dec) are in the "Example" directory. If we use as input of the program above "S7.dec" (S7 Decimal representation), the output files would be:

1. S7.ac (Autocorrelation Spectrum)
2. S7.anf (ANF Table)
3. S7.cy (Cycle structure)
4. S7.dp (Differential Profile)
5. S7.lp (Linear Profile)
6. S7.ls (Linear structures): It is an empty vector because there is no linear structures
7. S7.pol (Polynomial representation)
8. S7.tt (Truth Table)

9. S7.wal (Walsh Spectrum)

The same applies to S9 S-box analysis.

## 3.2 Compiling

There is only one library header files called "VBF.h". You should include a statement like this in the program that make use of VBF library,

```
#include "VBF.h"
```

If the directory is not installed on the standard search path of your compiler you will also need to provide its location to the preprocessor as a command line flag. The default location of the 'NTL' directory is '/usr/local/include/NTL'. A typical compilation command for a source file 'ex.cpp' with the GNU C++ compiler g++ included in a Makefile is,

```
GPP=g++
LIBS=-lntl
NTLINC= -I/usr/local/include -L/usr/local/lib

ex: ex.cpp VBF.h
     $(GPP) $(NTLINC) -Wall ex.cpp -o ex.exe $(LIBS)
```

This results in an executable file 'ex.exe' if the following command is executed:

```
$ make ex
```

In order to execute the example program included in the "Example" program with S7.dec and S9.dec, the following commands must be executed:

```
$ ./ex.exe S7 7
$ ./ex.exe S9 9
```

## 3.3 How to evaluate new algorithms

In order to evaluate an algorithm, we need to obtain a representation of this algorithm that can be used to initialize a VBF class. These representations are the Truth Table, Hexadecimal representation (only for Boolean functions), Decimal representation of its Truth Table, its trace together with the irreducible polynomial, Polynomials in ANF, ANF Table, Characteristic Function, Walsh Spectrum, permutation representation, Expansion and Compression DES vector representation, DES S-Box representation.

As an example we are going to describe the procedure followed to evaluate FI function in KASUMI algorithm. We used an implementation of KASUMI in c as you can see below:

```
/*-----------------------------------------------------------------------
 *                                                Kasumi.c
 *-----------------------------------------------------------------------
 *
 *      A sample implementation of KASUMI, the core algorithm for the
 *      3GPP Confidentiality and Integrity algorithms.
 *
 *      This has been coded for clarity, not necessarily for efficiency.
 *
 *      This will compile and run correctly on both Intel (little endian)
 *      and Sparc (big endian) machines. (Compilers used supported 32-bit ints).
 *
```

(continues on next page)

(continued from previous page)

```
 *      Version 1.1             08 May 2000
 *
 *-----------------------------------------------------------------------*/

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "VBF.h"

#include "Kasumi.h"

/*--------- 16 bit rotate left ------------------------------------------*/

#define ROL16(a,b)  (u16)((a<<b)|(a>>(16-b)))

/*------- unions: used to remove "endian" issues ------------------------*/

typedef union {
        u32 b32;
        u16 b16[2];
        u8  b8[4];
} DWORD;

typedef union {
        u16 b16;
        u8  b8[2];
} WORD;

/*-------- globals: The subkey arrays ----------------------------------*/

static u16 KLi1[8], KLi2[8];
static u16 KOi1[8], KOi2[8], KOi3[8];
static u16 KIi1[8], KIi2[8], KIi3[8];


/*-----------------------------------------------------------------------
 *      FI()
 *              The FI function (fig 3).  It includes the S7 and S9 tables.
 *              Transforms a 16-bit value.
 *-----------------------------------------------------------------------*/

static u16 FI( u16 in, u16 subkey )
{
        u16 nine, seven;
        static u16 S7[] = {
                54, 50, 62, 56, 22, 34, 94, 96, 38, 6, 63, 93, 2, 18,123, 33,
                55,113, 39,114, 21, 67, 65, 12, 47, 73, 46, 27, 25,111,124, 81,
                53, 9,121, 79, 52, 60, 58, 48,101,127, 40,120,104, 70, 71, 43,
                20,122, 72, 61, 23,109, 13,100, 77, 1, 16, 7, 82, 10,105, 98,
                117,116, 76, 11, 89,106, 0,125,118, 99, 86, 69, 30, 57,126, 87,
                112, 51, 17, 5, 95, 14, 90, 84, 91, 8, 35,103, 32, 97, 28, 66,
                102, 31, 26, 45, 75, 4, 85, 92, 37, 74, 80, 49, 68, 29,115, 44,
                64,107,108, 24,110, 83, 36, 78, 42, 19, 15, 41, 88,119, 59, 3};
        static u16 S9[] = {
                167,239,161,379,391,334,  9,338, 38,226, 48,358,452,385, 90,397,
                183,253,147,331,415,340, 51,362,306,500,262, 82,216,159,356,177,
```

(continues on next page)

(continued from previous page)

```
                175,241,489, 37,206, 17,  0,333, 44,254,378, 58,143,220, 81,400,
                 95,  3,315,245, 54,235,218,405,472,264,172,494,371,290,399, 76,
                165,197,395,121,257,480,423,212,240, 28,462,176,406,507,288,223,
                501,407,249,265, 89,186,221,428,164, 74,440,196,458,421,350,163,
                232,158,134,354, 13,250,491,142,191, 69,193,425,152,227,366,135,
                344,300,276,242,437,320,113,278, 11,243, 87,317, 36, 93,496, 27,
                487,446,482, 41, 68,156,457,131,326,403,339, 20, 39,115,442,124,
                475,384,508, 53,112,170,479,151,126,169, 73,268,279,321,168,364,
                363,292, 46,499,393,327,324, 24,456,267,157,460,488,426,309,229,
                439,506,208,271,349,401,434,236, 16,209,359, 52, 56,120,199,277,
                465,416,252,287,246,  6, 83,305,420,345,153,502, 65, 61,244,282,
                173,222,418, 67,386,368,261,101,476,291,195,430, 49, 79,166,330,
                280,383,373,128,382,408,155,495,367,388,274,107,459,417, 62,454,
                132,225,203,316,234, 14,301, 91,503,286,424,211,347,307,140,374,
                 35,103,125,427, 19,214,453,146,498,314,444,230,256,329,198,285,
                 50,116, 78,410, 10,205,510,171,231, 45,139,467, 29, 86,505, 32,
                 72, 26,342,150,313,490,431,238,411,325,149,473, 40,119,174,355,
                185,233,389, 71,448,273,372, 55,110,178,322, 12,469,392,369,190,
                  1,109,375,137,181, 88, 75,308,260,484, 98,272,370,275,412,111,
                336,318,  4,504,492,259,304, 77,337,435, 21,357,303,332,483, 18,
                 47, 85, 25,497,474,289,100,269,296,478,270,106, 31,104,433, 84,
                414,486,394, 96, 99,154,511,148,413,361,409,255,162,215,302,201,
                266,351,343,144,441,365,108,298,251, 34,182,509,138,210,335,133,
                311,352,328,141,396,346,123,319,450,281,429,228,443,481, 92,404,
                485,422,248,297, 23,213,130,466, 22,217,283, 70,294,360,419,127,
                312,377,  7,468,194,  2,117,295,463,258,224,447,247,187, 80,398,
                284,353,105,390,299,471,470,184, 57,200,348, 63,204,188, 33,451,
                 97, 30,310,219, 94,160,129,493, 64,179,263,102,189,207,114,402,
                438,477,387,122,192, 42,381,  5,145,118,180,449,293,323,136,380,
                 43, 66, 60,455,341,445,202,432, 8,237, 15,376,436,464, 59,461};

        /* The sixteen bit input is split into two unequal halves,  *
         * nine bits and seven bits - as is the subkey                    */

        nine  = (u16)(in>>7);
        seven = (u16)(in&0x7F);

        /* Now run the various operations */

        nine  = (u16)(S9[nine]  ^ seven);
        seven = (u16)(S7[seven] ^ (nine & 0x7F));

        seven ^= (subkey>>9);
        nine  ^= (subkey&0x1FF);

        nine  = (u16)(S9[nine]  ^ seven);
        seven = (u16)(S7[seven] ^ (nine & 0x7F));

        in = (u16)((seven<<9) + nine);

        return( in );
}


/*----------------------------------------------------------------------
 * FO()
 *              The FO() function.
```

(continues on next page)

(continued from previous page)

```
 *              Transforms a 32-bit value.  Uses <index> to identify the
 *              appropriate subkeys to use.
 *----------------------------------------------------------------------*/
static u32 FO( u32 in, int index )
{
        u16 left, right;
        u16 l,r;

        /* Split the input into two 16-bit words */

        left  = (u16)(in>>16);
        right = (u16) in;

        l = left;
        r = right;

        /* Now apply the same basic transformation three times         */

        left ^= KOi1[index];

        left  = FI( left, KIi1[index] );

        left ^= right;

        right ^= KOi2[index];

        right  = FI( right, KIi2[index] );

        right ^= left;

        left ^= KOi3[index];

        left  = FI( left, KIi3[index] );

        left ^= right;

        in = (((u32)right)<<16)+left;

        return( in );
}

/*----------------------------------------------------------------------
 * FL()
 *              The FL() function.
 *              Transforms a 32-bit value.  Uses <index> to identify the
 *              appropriate subkeys to use.
 *----------------------------------------------------------------------*/
static u32 FL( u32 in, int index )
{
        u16 l, r, a, b;

        /* split out the left and right halves */

        l = (u16)(in>>16);
        r = (u16)(in);

        /* do the FL() operations                        */
```

(continues on next page)

(continued from previous page)

```
        a  = (u16) (l & KLi1[index]);
        r ^= ROL16(a,1);

        b  = (u16)(r | KLi2[index]);
        l ^= ROL16(b,1);

        /* put the two halves back together */

        in = (((u32)l)<<16) + r;

        return( in );
}


/*-----------------------------------------------------------------------
 * Kasumi()
 *              the Main algorithm (fig 1).  Apply the same pair of operations
 *              four times.  Transforms the 64-bit input.
 *----------------------------------------------------------------------*/
void Kasumi( u8 *data )
{
        u32 left, right, temp;
        DWORD *d;
        int n;

        /* Start by getting the data into two 32-bit words (endian correct) */

        d = (DWORD*)data;

        left  = (((u32)d[0].b8[0])<<24)+(((u32)d[0].b8[1])<<16)
+(d[0].b8[2]<<8)+(d[0].b8[3]);
        right = (((u32)d[1].b8[0])<<24)+(((u32)d[1].b8[1])<<16)
+(d[1].b8[2]<<8)+(d[1].b8[3]);
        n = 0;
        do{
                temp = FL( left, n   );
                temp = FO( temp,  n++ );
                right ^= temp;
                temp = FO( right, n   );
                temp = FL( temp,   n++ );
                left ^= temp;
        }while( n<=7 );

        /* return the correct endian result */
        d[0].b8[0] = (u8)(left>>24);            d[1].b8[0] = (u8)(right>>24);
        d[0].b8[1] = (u8)(left>>16);            d[1].b8[1] = (u8)(right>>16);
        d[0].b8[2] = (u8)(left>>8);             d[1].b8[2] = (u8)(right>>8);
        d[0].b8[3] = (u8)(left);                d[1].b8[3] = (u8)(right);
}

/*-----------------------------------------------------------------------
 * KeySchedule()
 *              Build the key schedule.  Most "key" operations use 16-bit
 *              subkeys so we build u16-sized arrays that are "endian" correct.
 *----------------------------------------------------------------------*/
void KeySchedule( u8 *k )
```

(continues on next page)

(continued from previous page)

```
{
        static u16 C[] = {
                0x0123,0x4567,0x89AB,0xCDEF, 0xFEDC,0xBA98,0x7654,0x3210 };
        u16 key[8], Kprime[8];
        WORD *k16;
        int n;

        /* Start by ensuring the subkeys are endian correct on a 16-bit basis */

        k16 = (WORD *)k;
        for( n=0; n<8; ++n )
                key[n] = (u16)((k16[n].b8[0]<<8) + (k16[n].b8[1]));

        /* Now build the K'[] keys */

        for( n=0; n<8; ++n )
                Kprime[n] = (u16)(key[n] ^ C[n]);

        /* Finally construct the various sub keys */

        for( n=0; n<8; ++n )
        {
                KLi1[n] = ROL16(key[n],1);
                KLi2[n] = Kprime[(n+2)&0x7];
                KOi1[n] = ROL16(key[(n+1)&0x7],5);
                KOi2[n] = ROL16(key[(n+5)&0x7],8);
                KOi3[n] = ROL16(key[(n+6)&0x7],13);
                KIi1[n] = Kprime[(n+4)&0x7];
                KIi2[n] = Kprime[(n+3)&0x7];
                KIi3[n] = Kprime[(n+7)&0x7];
        }
}
```

In the main procedure, we defined an algorithm to obtain the Truth Table of FI function for the key values that are between “first” and “last” parameters.

```
int main(int argc, char *argv[])
{
   using namespace VBFNS;

   u16 l,k;
   long i,j,first,last;
   std::stringstream number;
   char file[33];
   NTL::vec_GF2 vn,vs;

   first = atoi(argv[1]);
   last = atoi(argv[2]);

   for (i = first; i <= last; i++)
   {
     sprintf(file,"%ld.tt",i);
     ofstream output(file);
     if(!output)
     {
        cerr << "Error opening " << file << endl;
```

(continues on next page)

(continued from previous page)

```
        return 0;
     }

     output << "[";

     number << i;
     number >> std::hex >> k;

     for (j = 0; j < 65536; j++)
     {
        number << j;
        number >> std::hex >> l;

        l  = FI( l, k );

        vn = to_vecGF2(l,16);

        output << vn << endl;
     }

     output << "]" << endl;
     output.close();
   }

}
```

# 4 Representations and characterizations

This chapter presents a review of theory relevant to the study of the typical forms of Vector Boolean function representations and chacterizations. We will consider representations those that uniquely represents a Vector Boolean function. Characterizations does not uniquely determine the Vector Boolean function in contrast to the previous matrices but provide some useful information in the context of cryptography.

Representations included in this chapter are the Truth Table (TT), the polynomials in Algebraic Normal Form (Pol) and ANF Table (ANF), the Image (Char), Component functions Truth Table(LTT), Sequence vectors of Component functions CTT, the Trace Representation (Trace) and Affine function Representation. A definition for all these representations are given and the relationships among them and their various properties are also discussed.

Characterizations such as Linear Profile (LP), Differential Profile (DP), Autocorrelation Spectrum (AC), Linear Structures (LS) are introduced. A definition for all these represen- tations are given and the relationships among them and the above representations and their various properties are also discussed.

The basic concepts of linear and differential cryptanalysis are introduced in terms of the Linear Profile and Differential

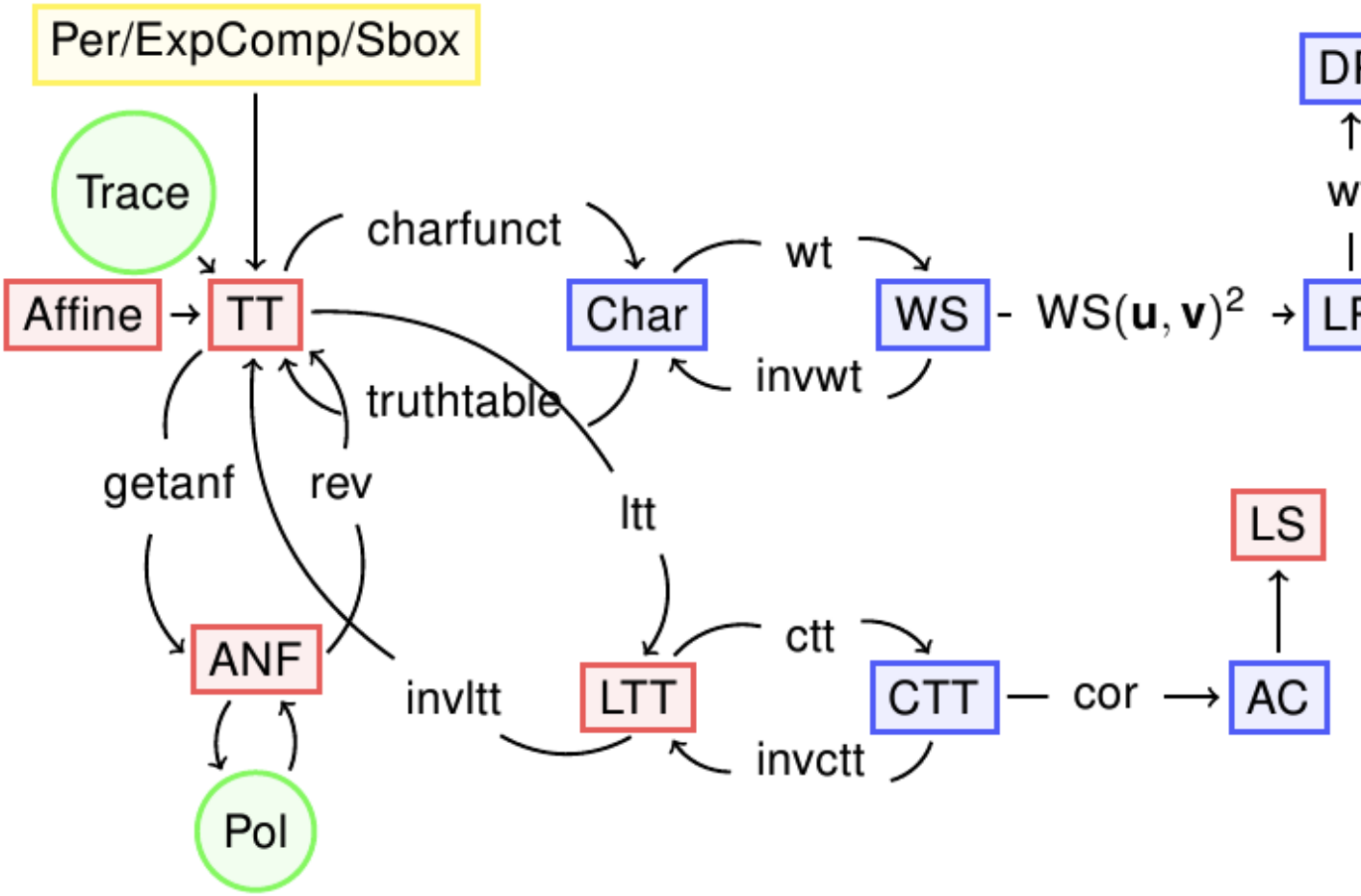
Profile, together with other properties related with these attacks, such as: linear potential, differential potential, linear or differential relations associated with a specific value.

Affine equivalence analysis of Boolean functions by means of VBF library is described. It is showed how to obtain the Frequency distribution of the absolute values of the Walsh Spectrum and of the Autocorrelation Spectrum.

It is possible to check randomness of a Vector Boolean function outputs with VBF by means of its cycle structure, and the analysis of the presence of fixed points or negated fixed points.

Finally, some other representations useful in block ciphers are described such as the Permutation Vector (Per), Expansion and Compression DES permutations and DES-like S-box representations. The description of each representation and characterization is complemented with the description of the methods in VBF related to them. Most of the member functions of V BF have an in-line definition, for instance: `void TT(NTL::mat_GF2& X, VBF& F)` is also defined as inline `NTL::mat_GF2 TT(VBF& F)`.

The figure summarizes the relationships among the different representations.




The representations which are Boolean matrices are coloured in red, those which are Integer matrices are coloured in blue, those that are vector of integers are coloured in yellow and those which are polynomial are coloured in green.

## 4.1 Truth Table

### 4.1.1 Description

A Vector Boolean function $F \in \mathcal{F}_{n,m}$ can be uniquely represented by its Truth Table which is a matrix with $2^n$ rows and *m* columns whose elements are the values of *F* taken on all possible vector of $\mathrm{V_n}$ ordered lexicographically.

Let $F \in \mathcal{F}_{n,m}$, if we take into account the one-to-one mapping of $\mathrm{V_n}$ onto the set of integers, we are able to define any vector Boolean function by the corresponding set of values:

$$F(\boldsymbol{\alpha_i}) \in \mathrm{V_m} \;\; \forall\, i \in \{0, \ldots, 2^n - 1\}$$

The matrix with $2^n$ rows and $m$ columns will be referred as the *Truth Table* of *F* and will be generally written as $\mathsf{TT}_F$:

$$\mathsf{TT}_F = \begin{bmatrix} f_1(\boldsymbol{\alpha_0}) & \ldots & f_m(\boldsymbol{\alpha_0}) \\ f_1(\boldsymbol{\alpha_1}) & \ldots & f_m(\boldsymbol{\alpha_1}) \\ \ldots & \ldots & \ldots \\ f_1(\boldsymbol{\alpha_{2^n-1}}) & \ldots & f_m(\boldsymbol{\alpha_{2^n-1}}) \end{bmatrix}$$

each $\boldsymbol{\alpha_i} = (x_1, \ldots, x_n) \in \mathrm{V_n} \;\; i \in \{1, \ldots, 2^n - 1\}$ is a vector whose decimal equivalent is $dec(\boldsymbol{\alpha_i}) = i = \sum_{j=1}^{n} x_j 2^{n-j}$, and all the vectors of $\mathrm{V_n}$ can be listed so that $\boldsymbol{\alpha_0} < \boldsymbol{\alpha_1} < \cdots < \boldsymbol{\alpha_{2^n-1}}$.

As a total order is defined over the assignments (inputs) of the Vector Boolean Function, the Truth Table can be uniquely represented by this matrix. Any function *F* can be uniquely described by its Truth Table $\mathsf{TT}_F \in \mathsf{M}_{2^n \times m}(\mathrm{GF}(2))$ (or by the Truth Tables of its coordinate functions $\mathsf{TT}_{f_i} \;\; i \in \{1, \ldots, m\}$) and it holds that:

$$\begin{array}{rcl} \gamma : \mathcal{F}_{n,m} & \rightarrow & \mathsf{M}_{2^n \times m}(\mathrm{GF}(2)) \\ F & \rightarrow & \mathsf{TT}_F \end{array}$$

is an isomorphism between the vector spaces $\mathcal{F}_{n,m}$ and $\mathsf{M}_{2^n \times m}(\mathrm{GF}(2))$, so that $\#\mathcal{F}_{n,m} = 2^{2^n \cdot m}$.

The Truth Table for an *n*-variable Boolean function $f$ should be in lexicographical form, i.e., $\mathsf{TT}_f = (f(0), f(1), f(2), \ldots, f(2^{n-1}))$. Since the Truth Table length might be too large, we represent it in hexadecimal rather than in binary notation. The hexadecimal Truth Table is obtained by replacing each four bits by their corresponding hexadecimal form. For instance, to enter $\mathsf{TT}_f = (0, 0, 1, 1, 1, 1, 1, 1)$ one should just write $\mathsf{TT}_f = 3f$.

The distance between two Vector Boolean functions $F, G \in \mathcal{F}_{n,m}$ is defined as the number of bits that are different in their respective Truth Tables:

$$d(F, G) = \sum_{\mathbf{x} \in \mathrm{V_n}} d\left(F(\mathbf{x}), G(\mathbf{x})\right)$$

where $d\left(F(\mathbf{x}), G(\mathbf{x})\right)$ is the Hamming distance between the two vectors $F(\mathbf{x}), G(\mathbf{x}) \in \mathrm{V_m}$.

The weight of a Vector Boolean function $F \in \mathcal{F}_{n,m}$ is equal to the distance between *F* and the corresponding zero Vector Boolean function $0 \in \mathcal{F}_{n,m}$ where $0(\mathbf{x}) = \mathbf{0} \;\forall\, \mathbf{x} \in \mathrm{V_n}$.

In order to obtain certain characterizations (such as Autocorrelation Spectrum), it is important to take into account two additional representations related to the Truth Table: $\mathsf{LTT}$ and $\mathsf{CTT}$.

We will denote by $\mathsf{LTT}$ of $F \in \mathcal{F}_{n,m}$ the matrix whose columns are the Truth Tables of the $2^m$ component functions of *F*. We will denote by $\mathsf{CTT}$ of *F* the matrix whose columns are the sequence vectors of the $2^m$ component functions of *F* (Sometimes it is called the Polarity Truth Table).

### 4.1.2 Library

A VBF class can be initialized by a Boolean Matrix representing the Truth Table with the following method:

```
void puttt(const NTL::mat_GF2& T)
```

To obtain the Truth Table of a Vector Boolean function the following method must be used:

```
void TT(NTL::mat_GF2& X, VBF& F)
```

A VBF class can be initialized by a collection of strings separated by carriage returns defined by *s* with the following method:

```
void putHexTT(istream& s)
```

Each row must be the hexadecimal representation of the Truth Table of the coordinate functions of a Vector Boolean function. To obtain the Truth Table in hexadecimal representation the following method must be used:

```
void getHexTT(ostream& s)
```

Analogously a VBF class can be initialized by a collecting of strings with binary representation of the Truth Table of coordinate functions:

```
void putBinTT(istream& s)
```

To obtain its Truth Table in binary representation the following method must be used:

```
void getBinTT(ostream& s)
```

A VBF class can be initialized by a Boolean vector representing the decimal representation of the Truth Table of a Vector Boolean Function defined by a vector of outputs in lexicographic order, called *d*, and knowing the number of component Boolean functions *m*:

```
void putDecTT(const NTL::vec_long& d,const long& m)
```

To obtain the Truth Table in decimal representation the following method must be used:

```
NTL::vec_long getDecTT() const
```

To obtain the weight of a Vector Boolean function *F* the following method must be used:

```
void weight(long& w, VBF& F)
```

A VBF class can be initialized by a Boolean Matrix representing the Truth Table of their component functions with the following method:

```
void putltt(const NTL::mat_GF2& L)
```

To obtain the Truth Table of the component functions of a Vector Boolean function the following method must be used:

```
void LTT(NTL::mat_GF2& X, VBF& F)
```

A VBF class can be initialized by a Boolean Matrix representing its Polarity Truth Table with the following method:

```
void putctt(const NTL::mat_ZZ& C)
```

To obtain the Polarity Truth Table of a Vector Boolean function the following method must be used:

```
void CTT(NTL::mat_ZZ& X, VBF& F)
```

### Example

The Truth Table of the NibbleSub S-box is the following:

```
[[1 1 1 0]
[0 1 0 0]
[1 1 0 1]
[0 0 0 1]
[0 0 1 0]
[1 1 1 1]
[1 0 1 1]
[1 0 0 0]
[0 0 1 1]
[1 0 1 0]
[0 1 1 0]
[1 1 0 0]
[0 1 0 1]
[1 0 0 1]
[0 0 0 0]
[0 1 1 1]
]
```

If we use a file with this matrix as the input of the following program, we can obtain its hexadecimal, binary and decimal representation, as well as the Truth Tables of the components functions and its Polarity Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input)
   {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The hexadecimal representation is: " << endl;
   F.getHexTT(cout);

   cout << endl << "The binary representation is: " << endl;
   F.getBinTT(cout);

   cout << endl << "The decimal representation is: " << endl
   << F.getDecTT() << endl;

   cout << endl << "The Truth Table of the component functions is: "
   << endl << LTT(F) << endl;
```

(continues on next page)

(continued from previous page)

```
   cout << endl << "The Polarity Truth Table is: "
   << endl << CTT(F) << endl;

   return 0;
}
```

The output of this program would be:

```
The hexadecimal representation is:
a754
e439
8ee1
368d

The binary representation is:
1010011101010100
1110010000111001
1000111011100001
0011011010001101

The decimal representation is:
[14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7]

The Truth Table of the component functions is:
[[0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0]
[0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0]
[0 0 1 1 0 0 1 1 1 1 0 0 1 1 0 0]
[0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0]
[0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0]
[0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0]
[0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1]
]

The Polarity Truth Table is:
[[1 1 -1 -1 -1 -1 1 1 -1 -1 1 1 1 1 -1 -1]
[1 1 1 1 -1 -1 -1 -1 1 1 1 1 -1 -1 -1 -1]
[1 -1 1 -1 -1 1 -1 1 -1 1 -1 1 1 -1 1 -1]
[1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1]
[1 1 -1 -1 1 1 -1 -1 1 1 -1 -1 1 1 -1 -1]
[1 -1 -1 1 -1 1 1 -1 -1 1 1 -1 1 -1 -1 1]
[1 -1 -1 1 1 -1 -1 1 -1 1 1 -1 -1 1 1 -1]
[1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1]
[1 -1 -1 1 1 -1 -1 1 1 -1 -1 1 1 -1 -1 1]
[1 1 -1 -1 1 1 -1 -1 -1 -1 1 1 -1 -1 1 1]
[1 1 -1 -1 -1 -1 1 1 1 1 -1 -1 -1 -1 1 1]
[1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1]
[1 -1 1 -1 -1 1 -1 1 1 -1 1 -1 -1 1 -1 1]
```

(continues on next page)

(continued from previous page)

```
[1 -1 1 -1 1 -1 1 -1 -1 1 -1 1 -1 1 -1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 -1 -1 1 -1 1 1 -1 1 -1 -1 1 -1 1 1 -1]
]
```

## 4.2 Trace representation

### 4.2.1 Description

We identify a Boolean function in *n* variables with a function from $\mathrm{GF}(2^{\mathrm{n}})$ to $\mathrm{GF}(2)$ and Vector Boolean function in *n* variables with a function from $\mathrm{GF}(2^{\mathrm{n}})$ to $\mathrm{GF}(2^{\mathrm{n}})$.

A *trace* is a function over a finite field $\mathrm{GF}(2^{\mathrm{n}})$ defined as: $tr(\mathbf{x}) = \sum_{i=0}^{2^n-1} x^i$

Since there is is an isomorphism between $\mathrm{V_n}$ and $\mathrm{GF}(2^{\mathrm{n}})$, it is possible to identify the trace function with a Boolean function in *n* variables. Analogously, a Vector Boolean function can be identified with trace as follows:

When *m*=*n*, we endow $\mathrm{V_n}$ with the structure of the field $\mathrm{GF}(2^{\mathrm{n}})$. Any $F \in \mathcal{F}_{n,n}$ admits a unique *univariate polynomial representation* over $\mathrm{GF}(2^{\mathrm{n}})$, of degree at most $2^n - 1$:

$$F(\mathbf{x}) = \sum_{i=0}^{2^n-1} \delta_i x^i,\ \delta_i \in \mathrm{GF}(2^{\mathrm{n}})$$

A general way to derive this polynomial representation is given by a Lagrange interpolation from the knowledge of the irreducible polynomial of degree *n* over $\mathrm{GF}(2)$ associated with the field $GF(2^n)$ and the Truth Table of *F*.

The *interpolation attack* [JakobsenK:97] is efficient when the degree of the univariate polynomial representation of the S-box over $\mathrm{GF}(2^{\mathrm{n}})$ is low or when the distance of the S-box to the set of low univariate degree functions is small. This attack exploits the low degree of the algebraic relation between some input (respective output) and intermediate data to infer some keybits relating the output (respective input) and the intermediate data.

### 4.2.2 Library

A VBF class can be initialized giving its trace *f* and the irreducible polynomial *g* with the following methods:

```
void putirrpol(GF2X& g)
void puttrace(string& f)
```

To obtain a Vector Boolean function trace representation the following method must be used:

```
void Trace(GF2EX& f, VBF& F)
```

and to print the trace representation use the following method:

```
void print(NTL_SNS ostream& s, GF2EX& f, const long& m)
```

#### Example

The following program provides the Trace representation over $\mathrm{GF}(2^{\mathrm{n}})$ of a Vector Boolean function with Truth Table in a file with extension ".tt". $\mathrm{GF}(2^{\mathrm{n}})$ is constructed with the irreducible polynomial whose corresponding GF2X representation is in a file with extension ".irr". The class GF2X implements polynomial arithmetic modulo 2 and a polynomial is represented as a coefficient vector.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;
   GF2X g;
   GF2EX f;
   int d;
   char file[33];

   sprintf(file,"%s.irr",argv[1]);
   ifstream input1(file);
   if(!input1) {
      cerr << "Error opening " << file << endl;
      return 0;
   }
   input1 >> g;
   F.putirrpol(g);
   input1.close();

   sprintf(file,"%s.tt",argv[1]);
   ifstream input(file);
   if(!input) {
      cerr << "Error opening " << file << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The trace representation is " << endl;
   f = Trace(F);
   d = deg(g);
   print(cout,f,d);

   return 0;
}
```

In this cipher, $\mathrm{GF}(2^8)$ is constructed with the irreducible polynomial $g(\mathbf{x}) = \mathbf{x}^8 + \mathbf{x}^4 + \mathbf{x}^3 + \mathbf{x} + \mathbf{1}$. The inputs of this program would be the Truth Table of $S_{RD}$ in a file with extension ".tt" and the corresponding GF2X representation of $g$ : [110110001] in a file with extension ".irr". The output of the program would be a GF2EX which represents polynomials over GF2E, and so can be used, for example, for arithmetic in $\mathrm{GF}(2^{\mathrm{n}})$:

$$05 \cdot x^{254} + 09 \cdot x^{253} + f9 \cdot x^{251} + 25 \cdot x^{247} + f4 \cdot x^{239} + 01 \cdot x^{223} + b5 \cdot x^{191} + 8f \cdot x^{127} + 63$$

The coefficients are elements of $\mathrm{GF}(2^8)$.

## 4.3 Polynomials in ANF

### 4.3.1 Description

Any vector Boolean function $F \in \mathcal{F}_{n,m}$ can be uniquely represented by $m$ multivariate polynomials over $\mathrm{GF}(2)$ (called coordinate functions) where each variable has power at most one. Each of these polynomials can be expressed as a sum of all distinct *k*th-order product terms $(0 < k \leq n)$ of the variables in the form:

$$\begin{aligned} f(x_1, \ldots, x_n) \quad &= a_0 + a_1 x_1 + \cdots + a_n x_n + a_{12} x_1 x_2 + \cdots + a_{n-1,n} x_{n-1} x_n + \cdots \\ &+ a_{12\ldots n} x_1 x_2 \ldots x_n = \textstyle\sum_{\mathsf{I} \in \mathsf{P}(N)} a_{\mathsf{I}} \left( \prod_{i \in \mathsf{I}} x_i \right) = \sum_{\mathsf{I} \in \mathsf{P}(N)} a_{\mathsf{I}} x^{\mathsf{I}}, \;\; a_{\mathsf{I}} \in \mathrm{GF}(2) \end{aligned}$$

where $\mathsf{P}(N)$ denotes the power set of $N = \{1, \ldots, n\}$. This representation of $f$ is called the *algebraic normal form (ANF)* of $f$. The algebraic normal form is thus a set of multivariate polynomials and the constant functions (those obtained by decomposition) are the coefficients of the $2^n$ products of input variables (i.e. monomials).

### 4.3.2 Library

A VBF class can be initialized giving its Polynomials in ANF with the following method:

```
void putpol(vec_pol& p)
```

To obtain its representation as Polynomials in ANF, the following method must be used:

```
void Pol(NTL_SNS ostream& s, VBF& F)
```

### Example

The following program provides the Polynomials in ANF Vector Boolean function from its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   Pol(cout,F);

   return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
1+x4+x2+x2x3+x2x3x4+x1+x1x2+x1x2x3
1+x3x4+x2+x2x4+x1+x1x3+x1x3x4
1+x4+x3+x3x4+x2x4+x2x3+x1x4+x1x3+x1x2+x1x2x4+x1x2x3
x3+x2x4+x1+x1x4+x1x3x4
```

which corresponds to the coordinate functions of *NibbleSub* as follows:

$$
\begin{aligned}
f_1(NibbleSub) &= 1 + x_4 + x_2 + x_2x_3 + x_2x_3x_4 + x_1 + x_1x_2 + x_1x_2x_3 \\
f_2(NibbleSub) &= 1 + x_3x_4 + x_2 + x_2x_4 + x_1 + x_1x_3 + x_1x_3x_4 \\
f_3(NibbleSub) &= 1 + x_4 + x_3 + x_3x_4 + x_2x_4 + x_2x_3 + x_1x_4 + x_1x_3 + x_1x_2 + x_1x_2x_4 + x_1x_2x_3 \\
f_4(NibbleSub) &= x_3 + x_2x_4 + x_1 + x_1x_4 + x_1x_3x_4
\end{aligned}
$$

## 4.4 ANF table

### 4.4.1 Description

*ANF table* of *F*, denoted by $\mathsf{ANF}_F \in \mathsf{M}_{2^n \times m}(\mathrm{GF}(2))$, represents the $2^n$ coefficients of the polynomials of each of the m coordinate functions in *ANF*.

The ANF table of *F*, denoted by $\mathsf{ANF}_F \in \mathsf{M}_{2^n \times m}(\mathrm{GF}(2))$, is defined by ${\mathsf{ANF}_F}^i = \mathsf{ANF}_{f_i} \;\; i \in \{1, \ldots, m\}$ where ${\mathsf{ANF}_F}^i$ is the *i*-th column of $\mathsf{ANF}_F$.

The ANF Table can be derived from the Truth Table by a binary matrix transformation called the Algebraic Normal Form Transformation (implemented in the VBF library with *getanf* method). The Truth Table can be obtained from the ANF Table using a method we call *rev*.

### 4.4.2 Library

A VBF class can be initialized giving its ANF table with the following method:

```
void putanf(const NTL::mat_GF2& A)
```

To obtain its representation as ANF table, the following method must be used:

```
void ANF(NTL::mat_GF2& X, VBF& F)
```

#### Example

The following program provides the ANF Table of a Vector Boolean function from its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
```

(continues on next page)

(continued from previous page)

```
        cerr << "Error opening " << argv[1] << endl;
        return 0;
    }
    input >> T;
    F.puttt(T);
    input.close();

    cout << "The ANF Table is:" << endl;
    cout << ANF(F) << endl;

    return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The ANF Table is:
[[1 1 1 0]
[1 0 1 0]
[0 0 1 1]
[0 1 1 0]
[1 1 0 0]
[0 1 1 1]
[1 0 1 0]
[1 0 0 0]
[1 1 0 1]
[0 0 1 1]
[0 1 1 0]
[0 1 0 1]
[1 0 1 0]
[0 0 1 0]
[1 0 1 0]
[0 0 0 0]
]
```

## 4.5 Characteristic Function

### 4.5.1 Description

The *characteristic or indicator* function of $F \in \mathcal{F}_{n,m}$, denoted by $\theta_F : \mathrm{V_n} \times \mathrm{V_m} \to \{0,1\}$, is defined by: $\theta_F(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{y} = F(\mathbf{x}) \\ 0 & \text{if } \mathbf{y} \neq F(\mathbf{x}) \end{cases}$

The Image of $F$ can be represented by a matrix whose rows are indexed by $\mathbf{x} \in \mathrm{V_n}$ and whose columns are indexed by $\mathbf{y} \in \mathrm{V_m}$ in lexicographic order, denoted by $\mathsf{Img(F)} \in \mathsf{M}_{2^n \times 2^m}(\mathrm{GF}(2))$ and defined as follows:

$$\mathsf{Img(F)} = \begin{bmatrix} \theta_F(\boldsymbol{\alpha_0},\boldsymbol{\alpha_0}) & \dots & \theta_F(\boldsymbol{\alpha_0},\boldsymbol{\alpha_{2^m-1}}) \\ \theta_F(\boldsymbol{\alpha_1},\boldsymbol{\alpha_0}) & \dots & \theta_F(\boldsymbol{\alpha_1},\boldsymbol{\alpha_{2^m-1}}) \\ \dots & \dots & \dots \\ \theta_F(\boldsymbol{\alpha_{2^n-1}},\boldsymbol{\alpha_0}) & \dots & \theta_F(\boldsymbol{\alpha_{2^n-1}},\boldsymbol{\alpha_{2^m-1}}) \end{bmatrix}$$

where $\theta_F(\mathbf{x},\mathbf{y})$ is the value of the indicator function at $(\mathbf{x},\mathbf{y})$.

It is clear that all the rows of the matrix $\mathsf{Img(F)}$ have one element equal to one and the rest is zero, that is $\forall\, i \in \{1,\dots,2^n\}$:

$$\begin{array}{rcl} \mathsf{Img(F)}_i & = & \begin{bmatrix} a_{i1} & \dots & a_{i2^m} \end{bmatrix} \\ \text{where } (\exists! j \in \{1,\dots,2^m\}\ a_{ij} = 1) & \wedge & (a_{ik} = 0\ \forall\, k \neq j \in \{1,\dots,2^m\}) \end{array}$$

The Image of $F$ can be derived from the Truth Table by a method implemented in the VBF library called *charfunct*. The Truth Table can be obtained from the Characteristic function using a method we call *truthtable*.

### 4.5.2 Library

A VBF class can be initialized giving its Image with the following method:

```
void putchar(const NTL::mat_ZZ& C)
```

To obtain its representation as Image, the following method must be used:

```
void Charact(NTL::mat_ZZ& C, VBF& F)
```

#### Example

The following program provides the Image of a Vector Boolean function from its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input)  {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The Image is:" << endl;
   cout << Charact(F) << endl;

   return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The Image is:
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
]
```

This matrix can be easily interpreted with the aid of the figure in which the rows and columns are indexed with the corresponding vector:

| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 0001 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0010 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| 0011 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0100 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0101 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0110 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 0111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1000 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 1001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 1010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1011 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |
| 1100 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1101 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| 1110 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

You can see for instance that the output of *0000* is *1110*.

## 4.6 Walsh Spectrum

### 4.6.1 Description

Linear and affine functions are considered as cryptographically weak functions. It is important to measure if a Vector Boolean function has some similarity with these functions. The similarity is measured by means of correlation. The values of Walsh Spectrum provide a measure of the correlation of the Vector Boolean function with the different Vector Boolean Linear functions.

Let $\mathsf{H_n}$ be the Walsh-Hadamard matrix of order $2^n$, then the vectors associated with its columns constitute an orthogonal basis for $\mathbb{R}^{2^n}$ over $\mathbb{R}$ so that:

$$\mathbf{x}\mathsf{H_n} = \mathbf{y}, \forall\, \mathbf{x}, \mathbf{y} \in \mathbb{R}^{2^n}$$

Let $f \in \mathcal{F}_n$, its sequence $\xi_f \in \mathbb{R}^{2^n}$ can be defined as a linear combination of the sequences of all the linear functions over $\mathrm{V_n}$, as they coincide with the rows of $\mathsf{H_n}$.

$$\xi_f = a_{\boldsymbol{\alpha_0}} \xi_{l_{\boldsymbol{\alpha_0}}} + \cdots + a_{\boldsymbol{\alpha_{2^n-1}}} \xi_{l_{\boldsymbol{\alpha_{2^n-1}}}}$$

where $a_{\mathbf{u}} = \frac{1}{2^n} \langle \xi_f, \xi_{l_{\mathbf{u}}} \rangle$

Let a Boolean function $f \in \mathcal{F}_n$, the *Walsh Transform* of *f* at $\mathbf{u} \in \mathrm{V_n}$ is the *n*-dimensional Discrete Fourier Transform and can be calculated as follows:

$$\mathcal{W}_f(\mathbf{u}) = \hat{\chi}_f(\mathbf{u}) = \mathcal{W}\{\xi_f\}(\mathbf{u}) = \langle \xi_f, \xi_{l_{\mathbf{u}}} \rangle = \sum_{\mathbf{x} \in \mathrm{V_n}} \chi_f(\mathbf{x}) \chi_{\mathbf{u}}(\mathbf{x})$$

or, as it is most often written as:

$$\mathcal{W}_f(\mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{f(\mathbf{x}) + \mathbf{u} \cdot \mathbf{x}}$$

As a result, the Walsh Transform of $f \in \mathcal{F}_n$ at $\mathbf{u}$ is the coefficient of the sequence of *f* ($\xi_f$) with respect to the basis constituted by the sequences of linear functions, scaled by a factor of $\frac{1}{2^n}$. If $\mathcal{W}_f$ is the Walsh transform of *f*, we say that $\xi_f$ and $\mathcal{W}_f$ form a Transform pair and write:

$$\xi_f \quad \overset{W}{\longleftrightarrow} \quad \mathcal{W}_f \;\; (\xi_f \text{ corresponds to } \mathcal{W}_f)$$

The *Walsh Spectrum* of *f* can be represented by a matrix whose rows are indexed by $\mathbf{u} \in \mathrm{V_n}$ in lexicographic order, denoted by $\mathsf{WS}(f) \in \mathsf{M}_{2^n \times 1}(\mathbb{R})$ and defined as follows:

$$\mathsf{WS}(f) = \begin{bmatrix} \hat{\chi}_f(\boldsymbol{\alpha_0}) & \ldots & \hat{\chi}_f(\mathbf{u}) & \ldots & \hat{\chi}_f(\boldsymbol{\alpha_{2^n-1}}) \end{bmatrix}^T$$

where $\hat{\chi}_f(\mathbf{u})$ is the value of the spectrum at $\mathbf{u}$. A Boolean function is uniquely determined by its Walsh Spectrum.

Let the vector Boolean function $F \in \mathcal{F}_{n,m}$, the *Walsh Transform* of *F* is the two-dimensional Walsh Transform defined by:

$$\mathcal{W}_F(\mathbf{u}, \mathbf{v}) \quad = \quad \hat{\theta}_F(\mathbf{u}, \mathbf{v}) = \mathcal{W}\{\mathsf{Img(F)}\}(\mathbf{u}, \mathbf{v}) = \textstyle\sum_{\mathbf{x} \in \mathrm{V_n}} \sum_{\mathbf{y} \in \mathrm{V_m}} \theta_F(\mathbf{x}, \mathbf{y}) \chi_{(\mathbf{u}, \mathbf{v})}(\mathbf{x}, \mathbf{y})$$

or, as it is most often written as:

$$\mathcal{W}_F(\mathbf{u}, \mathbf{v}) = \hat{\theta}_F(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{\mathbf{u} \cdot \mathbf{x} + \mathbf{v} \cdot F(\mathbf{x})}$$

The *Walsh Spectrum* of *F* can be represented by a matrix whose rows are indexed by $\mathbf{u} \in \mathrm{V_n}$ and whose columns are indexed by $\mathbf{v} \in \mathrm{V_m}$ in lexicographic order, denoted by $\mathsf{WS}(F) \in \mathsf{M}_{2^n \times 2^m}(\mathbb{R})$ and defined as follows:

$$\mathsf{WS}(F) = \begin{bmatrix} \hat{\theta}_F(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_0}) & \ldots & \hat{\theta}_F(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_{2^m-1}}) \\ \hat{\theta}_F(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_0}) & \ldots & \hat{\theta}_F(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_{2^m-1}}) \\ \cdots & \cdots & \cdots \\ \hat{\theta}_F(\boldsymbol{\alpha_{2^n-1}}, \boldsymbol{\alpha_0}) & \ldots & \hat{\theta}_F(\boldsymbol{\alpha_{2^n-1}}, \boldsymbol{\alpha_{2^m-1}}) \end{bmatrix}$$

where $\hat{\theta}_F(\mathbf{u}, \mathbf{v})$ is the value of the spectrum at $(\mathbf{u}, \mathbf{v})$.

We can deduce that the columns of this matrix are the spectra of the Boolean functions $l_{\mathbf{v}} \circ F$ for all the linear functions $l_{\mathbf{v}} \in \mathcal{L}_m$.

### 4.6.2 Library

A VBF class can be initialized giving its Walsh Spectrum with the following method:

```
void putwalsh(const NTL::mat_ZZ& W)
```

To obtain its representation as Walsh Spectrum the following method must be used:

```
void Walsh(NTL::mat_ZZ& W, VBF& F)
```

### Example

The following program provides the Walsh Spectrum of a Vector Boolean function from its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The Walsh Spectrum is:" << endl;
   cout << Walsh(F) << endl;

   return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The Walsh Spectrum is:
[[16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 -4 -4 0 0 -4 12 4 4 0 0 4 4 0 0]
[0 0 -4 -4 0 0 -4 -4 0 0 4 4 0 0 -12 4]
[0 0 0 0 0 0 0 0 4 -12 -4 -4 4 4 -4 -4]
[0 4 0 -4 -4 -8 -4 0 0 -4 0 4 4 -8 4 0]
[0 -4 -4 0 -4 0 8 4 -4 0 -8 4 0 -4 -4 0]
[0 4 -4 8 4 0 0 4 0 -4 4 8 -4 0 0 -4]
[0 -4 0 4 4 -8 4 0 -4 0 4 0 8 4 0 4]
[0 0 0 0 0 0 0 0 -4 4 4 -4 4 -4 -4 -12]
[0 0 -4 -4 0 0 -4 -4 -8 0 -4 4 0 8 4 -4]
[0 8 -4 4 -8 0 4 -4 4 4 0 0 4 4 0 0]
[0 8 0 -8 8 0 8 0 0 0 0 0 0 0 0 0]
[0 -4 8 -4 -4 0 4 0 4 0 4 8 0 4 0 -4]
[0 4 4 0 -4 8 0 4 -8 -4 4 0 4 0 0 4]
[0 4 4 0 -4 -8 0 4 -4 0 0 -4 -8 4 -4 0]
[0 -4 -8 -4 -4 0 4 0 0 -4 8 -4 -4 0 4 0]
]
```

We can see that the Walsh Spectrum of $f_1(NibbleSub)$ where

$$NibbleSub = (f_1(NibbleSub), f_2(NibbleSub), f_3(NibbleSub), f_4(NibbleSub))$$

corresponds to the Spectrum of $l_{(1,0,0,0)} \circ NibbleSub$. As a consequence, the Walsh Spectrum of $f_1(NibbleSub)$ coincides with the *9*-th column of $\mathsf{WS}(NibbleSub)$, that is, the column indexed by the vector $(1, 0, 0, 0)$.

## 4.7 Linear Profile

### 4.7.1 Description

A complete enumeration of all linear approximations of the S-box is given in the *Linear Profile* (In the literature, an equivalent matrix called Linear Approximation Table is used as well), which is a matrix whose rows are indexed by $\mathbf{u} \in \mathrm{V_n}$ and whose columns are indexed by $\mathbf{v} \in \mathrm{V_m}$ in lexicographic order, denoted by $\mathsf{LP}(F) \in \mathsf{M}_{2^n \times 2^m}(\mathbb{R})$. It holds that $\mathsf{LP}(F)(\mathbf{u}, \mathbf{v}) = |\mathsf{WS}(F)(\mathbf{u}, \mathbf{v})|^2$. The lower bound of the Linear Profile values is *0* and the upper bound is $2^{2n}$.

If we divide each element in the Linear Profile by the value on $\mathsf{LP}(F)(\mathbf{0}, \mathbf{0})$, these values represent the number of matches between the linear equation represented in hexadecimal as "Input Sum" and the sum of the output bits represented in hexadecimal as "Output Sum". Hence, subtracting to these values $\frac{1}{2}$ give the probability bias for the particular linear combination of input and output bits. The hexadecimal value representing a sum, when viewed as a binary value indicates the variables involved in the sum. For a linear combination of input variables represented as $u_1 \cdot x_1 + \cdots + u_n \cdot x_n$ where $u_i \in \mathrm{GF}(2)$, the hexadecimal value represents the binary value $u_1 \ldots u_n$, where $u_1$ is the most significant bit. Similarly, for a linear combination of output bits $v_1 \cdot y_1 + \cdots + v_m \cdot y_m$ where $v_i \in \mathrm{GF}(2)$, the hexadecimal value represents the binary vector $(v_1, \ldots, v_m)$.

In Linear Profiles, we are looking for entries with large value. If all of the entries are small, then the S-box does not have a very linear structure, and it may make Linear Cryptanalysis on the cipher difficult. The *Linear potential* of *F*, defined as $lp(F) = \frac{1}{2^{2n}} \cdot \overset{*}{\max} \left(\mathsf{WS}(F)(\mathbf{u}, \mathbf{v})^2\right)$ is a measure of linearity in Linear Cryptanalysis, and satisfies $2^{-n} \leq lp(F) \leq 1$ so that the lower bound holds if and only if *F* has maximum nonlinearity (*F* is bent) and the upper bound is reached when *F* is linear or affine. This criterion can take values from $\frac{1}{2^n}$ to 1. The larger $lp(F)$ is, the "closer" to a Linear Vector Boolean function is *F*.

### 4.7.2 Library

Note that the Linear Profile does not uniquely determine a Vector Boolean function. Thus, a VBF class cannot be initialized by its Linear Profile. To obtain its representation as Linear Profile, the following method must be used:

```
void LAT(NTL::mat_ZZ& LP, VBF& F)
```

In the VBF library, several methods have been defined in order to analyse the feasibility of Linear Cryptanalysis: Linear potential and Linear relations associated with a specific value of the Linear Profile. The method used to obtain the linear potential is the following:

```
void lp(NTL::RR& x, VBF& F)
```

If we want to obtain the linear expressions associated with the value of the Linear Profile "w", we will use this method:

```
void linear(NTL_SNS ostream& s, VBF& a, ZZ& w)
```

If we want to obtain the probability bias $|p_L - \frac{1}{2}|$ that a linear expression holds with the value of the Linear Profile "w", we will use this method:

```
void ProbLin(NTL::RR& x, VBF& a, NTL::ZZ& w)
```

### Example

The following program finds out the Linear Profile of a Vector Boolean function together with the linear expressions that have the highest value, except from the value in $\mathsf{LP}(F)(\mathbf{0},\mathbf{0})$, their probability, this highest value and the linear potential.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
  using namespace VBFNS;

  VBF     F;
  NTL::mat_GF2 T;
  NTL::ZZ   w;
  NTL::RR   bias;

  ifstream input(argv[1]);
  if(!input) {
   cerr << "Error opening " << argv[1] << endl;
   return 0;
  }
  input >> T;
  F.puttt(T);
  input.close();

  cout << "The Linear Profile is:" << endl;
  cout << LAT(F) << endl;

  w = maxLAT(F);
  cout << endl << "The highest value of the Linear Profile is= "
  << w << endl << endl;

  cout << "The linear expressions that have the highest value are:"
  << endl;
  linear(cout,F,w);

  ProbLin(bias,F,w);
  cout << endl;
  cout << "These expressions hold with probability bias= "
  << bias << endl;

  cout << endl << "The linear potential is= " << lp(F) << endl;

  return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The Linear Profile is:
[[256 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 16 16 0 0 16 144 16 16 0 0 16 16 0 0]
[0 0 16 16 0 0 16 16 0 0 16 16 0 0 144 16]
[0 0 0 0 0 0 0 0 16 144 16 16 16 16 16 16]
[0 16 0 16 16 64 16 0 0 16 0 16 16 64 16 0]
[0 16 16 0 16 0 64 16 16 0 64 16 0 16 16 0]
[0 16 16 64 16 0 0 16 0 16 16 64 16 0 0 16]
[0 16 0 16 16 64 16 0 16 0 16 0 64 16 0 16]
[0 0 0 0 0 0 0 0 16 16 16 16 16 16 16 144]
[0 0 16 16 0 0 16 16 64 0 16 16 0 64 16 16]
[0 64 16 16 64 0 16 16 16 16 0 0 16 16 0 0]
[0 64 0 64 64 0 64 0 0 0 0 0 0 0 0 0]
[0 16 64 16 16 0 16 0 16 0 16 64 0 16 0 16]
[0 16 16 0 16 64 0 16 64 16 16 0 16 0 0 16]
[0 16 16 0 16 64 0 16 16 0 0 16 64 16 16 0]
[0 16 64 16 16 0 16 0 0 16 64 16 16 0 16 0]
]

The highest value of the Linear Profile is= 144

The linear expressions that have the highest value are:
x4=y2+y3+y4
x3=y1+y2+y3
x3+x4=y1+y4
x1=y1+y2+y3+y4

These expressions hold with probability bias= 0.0625

The linear potential is= 0.5625
```

The figure represents the Linear Profile of *NibbleSub* and emphasizes in red the elements which achieve the highest value.

| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 256 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0001 | 0 | 0 | 16 | 16 | 0 | 0 | 16 | 144 | 16 | 16 | 0 | 0 | 16 | 16 | 0 |
| 0010 | 0 | 0 | 16 | 16 | 0 | 0 | 16 | 16 | 0 | 0 | 16 | 16 | 0 | 0 | 144 |
| 0011 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 144 | 16 | 16 | 16 | 16 | 16 |
| 0100 | 0 | 16 | 0 | 16 | 16 | 64 | 16 | 0 | 0 | 16 | 0 | 16 | 16 | 64 | 16 |
| 0101 | 0 | 16 | 16 | 0 | 16 | 0 | 64 | 16 | 16 | 0 | 64 | 16 | 0 | 16 | 16 |
| 0110 | 0 | 16 | 16 | 64 | 16 | 0 | 0 | 16 | 0 | 16 | 16 | 64 | 16 | 0 | 0 |
| 0111 | 0 | 16 | 0 | 16 | 16 | 64 | 16 | 0 | 16 | 0 | 16 | 0 | 64 | 16 | 0 |
| 1000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 1001 | 0 | 0 | 16 | 16 | 0 | 0 | 16 | 16 | 64 | 0 | 16 | 16 | 0 | 64 | 16 |
| 1010 | 0 | 64 | 16 | 16 | 64 | 0 | 16 | 16 | 16 | 16 | 0 | 0 | 16 | 16 | 0 |
| 1011 | 0 | 64 | 0 | 64 | 64 | 0 | 64 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1100 | 0 | 16 | 64 | 16 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 64 | 0 | 16 | 0 |
| 1101 | 0 | 16 | 16 | 0 | 16 | 64 | 0 | 16 | 64 | 16 | 16 | 0 | 16 | 0 | 0 |
| 1110 | 0 | 16 | 16 | 0 | 16 | 64 | 0 | 16 | 16 | 0 | 0 | 16 | 64 | 16 | 16 |
| 1111 | 0 | 16 | 64 | 16 | 16 | 0 | 16 | 0 | 0 | 16 | 64 | 16 | 16 | 0 | 16 |

## 4.8 Differential Profile

### 4.8.1 Description

The first step of Differential Cryptanalysis is to compute the characteristics of inputs and the outputs of the S-boxes, which we will then combine together to form a characteristic for the complete cipher. Consider a $n \times m$ S-box with input $\mathbf{x} = (x_1, \ldots, x_n)$ and output $\mathbf{y} = (y_1, \ldots, y_m)$. All difference pairs of an S-box, $(\Delta\mathbf{x}, \Delta\mathbf{y})$, can be examined and the probability of $\Delta\mathbf{y}$ given $\Delta\mathbf{x}$ can be derived by considering input pairs $(\mathbf{x}^{'}, \mathbf{x}^{''})$ such that $\mathbf{x}^{'} + \mathbf{x}^{''} = \Delta\mathbf{x}$. Since the ordering of the pair is not relevant, for a $n \times m$ S-box we need only consider all $2^n$ values for $\mathbf{x}^{'}$ and then the value of $\Delta\mathbf{x}$ constrains the value of $\mathbf{x}^{''}$ to be $\mathbf{x}^{''} = \mathbf{x}^{'} + \Delta\mathbf{x}$. We can derive the resulting values of $\Delta\mathbf{y}$ for each input pair $(\mathbf{x}^{'}, \mathbf{x}^{''} = \mathbf{x}^{'} + \Delta\mathbf{x})$.

We can tabulate the complete differential data for an S-box in the *Differential Profile* (In the literature, an equivalent matrix called Difference Distribution Table is used as well), which the rows represent $\Delta\mathbf{x}$ values and the columns represent $\Delta\mathbf{y}$ values.

If we divide each element in the Differential Profile by the value on $\mathsf{DP}(F)(\mathbf{0}, \mathbf{0})$, these values represent the probability of the corresponding output difference $\Delta\mathbf{y}$ value given the input difference $\Delta\mathbf{x}$, that is $(\Delta\mathbf{x} \Rightarrow \Delta\mathbf{y})$, called *characteristic*. In general, entries in the Differential Profile with fewer bits set in the $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ that have higher probability are desirable.

Let $F \in \mathcal{F}_{n,m}$, if we denote by $D_F(\mathbf{u}, \mathbf{v})$ the set of vectors where the difference Vector Boolean Function of *F* in the direction of $\mathbf{u} \in \mathrm{V_n}$ coincides with $\mathbf{v} \in \mathrm{V_m}$ by:

$$D_F(\mathbf{u}, \mathbf{v}) = \{\mathbf{x} \in \mathrm{V_n} \mid \Delta_{\mathbf{u}} F(\mathbf{x}) = \mathbf{v}\}$$

Let $F \in \mathcal{F}_{n,m}$ where $n \geq m$. The matrix containing all possible values of $\#D_F(\mathbf{u}, \mathbf{v})$ is referred to as its *XOR or Differential Distribution Table*.

Nyberg in [Nyberg:93] introduced the concept of *differential uniformity* as a measure of the resistance to differential crytanalysis as follows:

A Vector Boolean function $F \in \mathcal{F}_{n,m}$ is called differentially $du(F)$-uniform if for all $\mathbf{u} \neq \mathbf{0} \in \mathrm{V_n}$ and $\mathbf{v} \in \mathrm{V_m}$:

$$\#\{\mathbf{x} \in \mathrm{V_n} \mid F(\mathbf{x} + \mathbf{u}) + F(\mathbf{x}) = \mathbf{v}\} \leq du(F)$$

Let $du(F)$ (differential uniformity of *F*) is the largest value in Differential Distribution Table of *F* (not counting the first entry in the first row), namely,

$$du(F) = \max_{(\mathbf{u},\mathbf{v}) \neq (\mathbf{0},\mathbf{0})} \#D_F(\mathbf{u}, \mathbf{v}) = \max_{(\mathbf{u},\mathbf{v}) \neq (\mathbf{0},\mathbf{0})} \#\{\mathbf{x} \in \mathrm{V_n} \mid F(\mathbf{x}) + F(\mathbf{x} + \mathbf{u}) = \mathbf{v}\}$$

Let define the function $\delta_F : \mathrm{V_n} \times \mathrm{V_m} \to \mathbb{Q}$ as: $\delta_F(\mathbf{u}, \mathbf{v}) = \frac{1}{2^n} \#D_F(\mathbf{u}, \mathbf{v})$.

The *Differential Profile* of *F* can be represented by a matrix whose rows are indexed by $\mathbf{u} \in \mathrm{V_n}$ and whose columns are indexed by $\mathbf{v} \in \mathrm{V_m}$ in lexicographic order, denoted by $\mathsf{DP}(F) \in \mathsf{M}_{2^n \times 2^m}(R)$ and defined as follows:

$$\mathsf{DP}(F) = 2^{2n+m} \begin{bmatrix} \delta_F(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_0}) & \ldots & \delta_F(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_{2^m-1}}) \\ \delta_F(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_0}) & \ldots & \delta_F(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_{2^m-1}}) \\ \ldots & \ldots & \ldots \\ \delta_F(\boldsymbol{\alpha_{2^n-1}}, \boldsymbol{\alpha_0}) & \ldots & \delta_F(\boldsymbol{\alpha_{2^n-1}}, \boldsymbol{\alpha_{2^m-1}}) \end{bmatrix}$$

The maximum value of $\delta_F(\mathbf{u}, \mathbf{v})$ is called the *differential potential* of *F*:

$$dp(F) = \max\{\delta_F(\mathbf{u}, \mathbf{v}) \mid \forall\, \mathbf{u} \in \mathrm{V_n}, \mathbf{v} \in \mathrm{V_m}, (\mathbf{u}, \mathbf{v}) \neq (\mathbf{0}, \mathbf{0})\}$$

The differential uniformity of $F \in \mathcal{F}_{n,m}$ and its differential potential are related as: $dp(F) = \frac{1}{2^n} du(F)$.

It is a measure of the robustness against differential cryptanalysis where $2^{-m} \leq dp(F) \leq 1$ and the lower bound holds if and only if *F* is bent and the upper bound is reached when *F* is linear or affine. The differential uniformity of $F \in \mathcal{F}_{n,m}$ and its differential potential are related by $dp(F) = 2^{-n} du(F)$.

### 4.8.2 Library

Note that the Differential Profile does not uniquely determine a Vector Boolean function. Thus, a VBF class cannot be initialized by its Differential Profile. To obtain its representation as Differential Profile, the following method must be used:

```
void DAT(NTL::mat_ZZ& DP, VBF& F)
```

In the VBF library, several methods have been defined in order to analyse the feasibility of differential cryptanalysis: Differential potential and Differential relations associated with a specific value of the Differential profile. The method used to obtain the differential potential is the following:

```
void dp(NTL::RR& x, VBF& F)
```

If we want to obtain the characteristics associated with the value of the Differential Profile "w", we will use this method:

```
void differential(NTL_SNS ostream& s, VBF& a, ZZ& w)
```

If we want to obtain the probability that a characteristic $(\Delta\mathbf{x} \Rightarrow \Delta\mathbf{y})$ holds with the value of the Differential Profile "w", we will use this method:

```
void ProbDif(NTL::RR& x, VBF& a, NTL::ZZ& w)
```

#### Example

The following program finds out the Differential Profile of a Vector Boolean function together with the characteristics that have the highest value, except from the value in $\mathsf{DP}(F)(\mathbf{0},\mathbf{0})$, their probability, this highest value and the differential potential.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;
   NTL::ZZ      w;
   NTL::RR      p;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The Differential Profile is:" << endl;
   cout << DAT(F) << endl;
```

(continues on next page)

(continued from previous page)

```
    w = maxDAT(F);
    cout << endl << "The highest value of the Differential Profile is= "
    << w << endl;

    cout << endl << "The characteristics that have the highest value are:"
    << endl;
    differential(cout,F,w);

    ProbDif(p,F,w);
    cout << endl << "These expressions hold with probability= " << p << endl;

    cout << endl << "The differential potential is= " << dp(F) << endl;

    return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The Differential Profile is:
[[4096 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 512 0 0 0 512 0 512 1024 0 1024 512 0 0]
[0 0 0 512 0 1536 512 512 0 512 0 0 0 0 512 0]
[0 0 512 0 512 0 0 0 0 1024 512 0 512 0 0 1024]
[0 0 0 512 0 0 1536 0 0 512 0 1024 512 0 0 0]
[0 1024 0 0 0 512 512 0 0 0 1024 0 512 0 0 512]
[0 0 0 1024 0 1024 0 0 0 0 0 0 512 512 512 512]
[0 0 512 512 512 0 512 0 0 512 512 0 0 0 0 1024]
[0 0 0 0 0 0 512 512 0 0 0 1024 0 1024 512 512]
[0 512 0 0 512 0 0 1024 512 0 512 512 512 0 0 0]
[0 512 512 0 0 0 0 0 1536 0 0 512 0 0 1024 0]
[0 0 2048 0 0 512 0 512 0 0 0 0 0 512 0 512]
[0 512 0 0 512 512 512 0 0 0 0 512 0 1536 0 0]
[0 1024 0 0 0 0 0 1024 512 0 512 0 512 0 512 0]
[0 0 512 1024 512 0 0 0 1536 0 0 0 0 0 512 0]
[0 512 0 0 1536 0 0 0 0 1024 0 512 0 0 512 0]
]

The highest value of the Differential Profile is= 2048

The characteristics that have the highest value are:
[1 0 1 1]->[0 0 1 0]

These expressions hold with probability= 0.5

The differential potential is= 0.5
```

The figure represents the Differential Profile of *NibbleSub* and emphasizes in blue the elements which achieve the highest value.

| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 4096 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0001 | 0 | 0 | 0 | 512 | 0 | 0 | 0 | 512 | 0 | 512 | 1024 | 0 | 1024 | 512 | 0 |
| 0010 | 0 | 0 | 0 | 512 | 0 | 1536 | 512 | 512 | 0 | 512 | 0 | 0 | 0 | 0 | 512 |
| 0011 | 0 | 0 | 512 | 0 | 512 | 0 | 0 | 0 | 0 | 1024 | 512 | 0 | 512 | 0 | 0 |
| 0100 | 0 | 0 | 0 | 512 | 0 | 0 | 1536 | 0 | 0 | 512 | 0 | 1024 | 512 | 0 | 0 |
| 0101 | 0 | 1024 | 0 | 0 | 0 | 512 | 512 | 0 | 0 | 0 | 1024 | 0 | 512 | 0 | 0 |
| 0110 | 0 | 0 | 0 | 1024 | 0 | 1024 | 0 | 0 | 0 | 0 | 0 | 0 | 512 | 512 | 512 |
| 0111 | 0 | 0 | 512 | 512 | 512 | 0 | 512 | 0 | 0 | 512 | 512 | 0 | 0 | 0 | 0 |
| 1000 | 0 | 0 | 0 | 0 | 0 | 0 | 512 | 512 | 0 | 0 | 0 | 1024 | 0 | 1024 | 512 |
| 1001 | 0 | 512 | 0 | 0 | 512 | 0 | 0 | 1024 | 512 | 0 | 512 | 512 | 512 | 0 | 0 |
| 1010 | 0 | 512 | 512 | 0 | 0 | 0 | 0 | 0 | 1536 | 0 | 0 | 512 | 0 | 0 | 1024 |
| 1011 | 0 | 0 | 2048 | 0 | 0 | 512 | 0 | 512 | 0 | 0 | 0 | 0 | 0 | 512 | 0 |
| 1100 | 0 | 512 | 0 | 0 | 512 | 512 | 512 | 0 | 0 | 0 | 0 | 512 | 0 | 1536 | 0 |
| 1101 | 0 | 1024 | 0 | 0 | 0 | 0 | 0 | 1024 | 512 | 0 | 512 | 0 | 512 | 0 | 512 |
| 1110 | 0 | 0 | 512 | 1024 | 512 | 0 | 0 | 0 | 1536 | 0 | 0 | 0 | 0 | 0 | 512 |
| 1111 | 0 | 512 | 0 | 0 | 1536 | 0 | 0 | 0 | 0 | 1024 | 0 | 512 | 0 | 0 | 512 |

## 4.9 Autocorrelation Spectrum

### 4.9.1 Description

The Autocorrelation provides a useful description of a Vector Boolean function in relation to some cryptographic criteria. It is derived from the sequences of the component functions of the Vector Boolean function and does not uniquely determine the Vector Boolean function itself.

The *directional derivative* of $f \in \mathcal{F}_n$ in the direction of $\mathbf{u} \in \mathrm{V_n}$ is defined as $\Delta_{\mathbf{u}} f(\mathbf{x}) = f(\mathbf{x}+\mathbf{u}) + f(\mathbf{x}), \;\; \mathbf{x} \in \mathrm{V_n}$.

Similarly, the *directional derivative* of the sequence of a Boolean function $\xi_f$ in the direction of $\mathbf{u} \in \mathrm{V_n}$ is defined as: $\Delta_{\mathbf{u}} \chi_f(\mathbf{x}) = \chi_f(\mathbf{x}+\mathbf{u}) \cdot \chi_f(\mathbf{x}), \;\; \mathbf{x} \in \mathrm{V_n}$.

The *autocorrelation* of $f \in \mathcal{F}_n$ with respect to the shift $\mathbf{u} \in \mathrm{V_n}$, $r_f(\mathbf{u})$, is defined by the Polarity Truth Table to be:

$$r_f(\mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} \chi_f(\mathbf{x}) \chi_f(\mathbf{x}+\mathbf{u})$$

From this definition of the autocorrelation function we note two important properties:

- For every Boolean function $r_f(\mathbf{0}) = 2^n$, since $(\chi_f(\mathbf{x}))^2 = 1 \; \forall \mathbf{x} \in \mathrm{V_n}$.
- The value of $r_f(\mathbf{u})$ when $\mathbf{u} \neq \mathbf{0}$ must be proportional to the correlation between $f(\mathbf{x}+\mathbf{u})$ and $f(\mathbf{x})$, i.e.: $r_f(\mathbf{u}) = 2^n \cdot \mathrm{C}\left(f(\mathbf{x}+\mathbf{u}), f(\mathbf{x})\right)$.

The Aucorrelation Spectrum gives an indication of the imbalance of all first order derivatives of the component functions of a Vector Boolean function. As differential cryptanalysis exploits imbalanced derivatives of Vector Boolean functions, the Aucorrelation Spectrum is vital in the analysis.

***Autocorrelation Spectrum* is denoted by :math:'matr{R}(F) in** matr{M}_{2^n times 2^m}(bbbz)'. The columns of the matrix correspond to the Autocorrelation Spectrum of their component functions. The lower bound of the Autocorrelation Spectrum values is $-2^n$ and the upper bound is $2^n$.

### 4.9.2 Linear structures

If the *directional derivative* of $f \in \mathcal{F}_n$ in the direction of $\mathbf{u} \in \mathrm{V_n}$: $\Delta_{\mathbf{u}} f(\mathbf{x}) = f(\mathbf{x}+\mathbf{u}) + f(\mathbf{x})$ is a constant function, then $\mathbf{u}$ is a *linear structure* of $f$ [Lai:95] [Chaum:E85]. The zero vector $\mathbf{0}$ is a trivial linear structure since

$\Delta_{\mathbf{0}} f(\mathbf{x}) = 0 \ \forall \mathbf{x} \in \mathrm{V_n}$. From the point of view of autocorrelation, a vector in $\mathrm{V_n}$ is a linear structure if it satisfies the following:

The vector $\mathbf{u} \in \mathrm{V_n}$ is a linear structure of $f$ if and only if $|r_f(\mathbf{u})| = 2^n$.

The notion of linear structures can be extended for the case of Vector Boolean functions. The definition of a Vector Boolean function that has a linear structure was originally proposed by Chaum [Chaum:E85] and Evertse [Evertse:87]. They defined that a Vector Boolean function $F$ has a linear structure by considering the existence of nontrivial linear structure in any of the component functions of $F$.

$F \in \mathcal{F}_{n,m}$ is said to have a linear structure if there exists a nonzero vector $\mathbf{u} \in \mathrm{V_n}$ together with a nonzero vector $\mathbf{v} \in \mathrm{V_m}$ such that $\mathbf{v} \cdot F(\mathbf{x}) + \mathbf{v} \cdot F(\mathbf{x} + \mathbf{u})$ takes the same value $c \in \mathrm{GF}(2) \ \forall \mathbf{x} \in \mathrm{V_n}$.

$F \in \mathcal{F}_{n,m}$ is said to have a linear structure if there exists a nonzero vector $\mathbf{u} \in \mathrm{V_n}$ together with a nonzero vector $\mathbf{v} \in \mathrm{V_m}$ such that $|r_{\mathbf{v} \cdot F}(\mathbf{u})| = 2^n$.

Nonlinear cryptographic functions used in block ciphers should have no nonzero linear structures [Evertse:87]. The existence of nonzero linear structures, for the functions implemented in stream ciphers, is a potential risk that should also be avoided, despite the fact that such existence could not be used in attacks, so far.

### 4.9.3 Library

To obtain its representation as Autocorrelation Spectrum, the following method must be used:

```
void AC(NTL::mat_ZZ& R, VBF& F)
```

The method used to obtain the linear structures is the following:

```
void LS(NTL_SNS ostream& s, VBF& F)
```

### Example

The following program finds out the Autocorrelation Spectrum of a Vector Boolean function together with its linear structures having as input its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The Autocorrelation Spectrum is:" << endl;
```

(continues on next page)

(continued from previous page)

```
    cout << AC(F) << endl;
    cout << endl << "The linear structures are: " << endl;
    LS(cout,F);

    return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The Autocorrelation Spectrum is:
[[16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
[16 0 0 0 0 0 -8 -8 -8 -8 -8 8 0 0 8 8]
[16 -8 0 -8 -8 0 0 8 8 -8 0 0 -8 8 -8 8]
[16 0 0 0 0 0 0 -16 -8 8 0 0 0 0 -8 8]
[16 0 -8 0 0 -16 0 8 0 8 -8 -8 -8 0 8 8]
[16 0 0 -8 0 0 0 -8 0 -8 8 -8 0 -8 8 8]
[16 -8 0 0 -8 0 -8 8 0 -8 0 0 8 0 -8 8]
[16 0 -8 0 0 0 0 -8 0 8 0 0 0 -8 -8 8]
[16 -8 -8 0 -8 0 0 8 -8 8 0 0 0 0 8 -8]
[16 0 0 8 0 0 0 -8 0 -8 0 0 -8 0 8 -8]
[16 8 0 0 8 0 8 8 -8 -8 0 -8 0 0 -8 -16]
[16 0 -8 -8 0 16 -8 -8 8 8 -8 -8 8 8 -8 -8]
[16 -8 8 -8 -8 0 -8 8 0 8 0 0 0 -8 8 -8]
[16 0 0 0 0 0 8 -8 0 -16 0 0 0 0 8 -8]
[16 8 0 8 8 0 0 8 0 -8 -8 0 0 -8 -16 -8]
[16 0 8 0 0 -16 0 -8 0 8 8 8 -8 0 -8 -8]
]

The linear structures are:
([0 0 1 1],[0 1 1 1])
([0 1 0 0],[0 1 0 1])
([1 0 1 0],[1 1 1 1])
([1 0 1 1],[0 1 0 1])
([1 1 0 1],[1 0 0 1])
([1 1 1 0],[1 1 1 0])
([1 1 1 1],[0 1 0 1])
```

We can notice that *NibbleSub* S-box has seven linear structures which are the following:

The figure represents the Autocorrelation Spectrum of *NibbleSub* and emphasizes in red the values corresponding these linear structures.

| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | |
| 0001 | 16 | 0 | 0 | 0 | 0 | 0 | −8 | −8 | −8 | −8 | −8 | 8 | 0 | 0 | 8 | |
| 0010 | 16 | −8 | 0 | −8 | −8 | 0 | 0 | 8 | 8 | −8 | 0 | 0 | −8 | 8 | −8 | |
| 0011 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | −16 | −8 | 8 | 0 | 0 | 0 | 0 | −8 | |
| 0100 | 16 | 0 | −8 | 0 | 0 | −16 | 0 | 8 | 0 | 8 | −8 | −8 | −8 | 0 | 8 | |
| 0101 | 16 | 0 | 0 | −8 | 0 | 0 | 0 | −8 | 0 | −8 | 8 | −8 | 0 | −8 | 8 | |
| 0110 | 16 | −8 | 0 | 0 | −8 | 0 | −8 | 8 | 0 | −8 | 0 | 0 | 8 | 0 | −8 | |
| 0111 | 16 | 0 | −8 | 0 | 0 | 0 | 0 | −8 | 0 | 8 | 0 | 0 | 0 | −8 | −8 | |
| 1000 | 16 | −8 | −8 | 0 | −8 | 0 | 0 | 8 | −8 | 8 | 0 | 0 | 0 | 0 | 8 | |
| 1001 | 16 | 0 | 0 | 8 | 0 | 0 | 0 | −8 | 0 | −8 | 0 | 0 | −8 | 0 | 8 | |
| 1010 | 16 | 8 | 0 | 0 | 8 | 0 | 8 | 8 | −8 | −8 | 0 | −8 | 0 | 0 | −8 | |
| 1011 | 16 | 0 | −8 | −8 | 0 | 16 | −8 | −8 | 8 | 8 | −8 | −8 | 8 | 8 | −8 | |
| 1100 | 16 | −8 | 8 | −8 | −8 | 0 | −8 | 8 | 0 | 8 | 0 | 0 | 0 | −8 | 8 | |
| 1101 | 16 | 0 | 0 | 0 | 0 | 0 | 8 | −8 | 0 | −16 | 0 | 0 | 0 | 0 | 8 | |
| 1110 | 16 | 8 | 0 | 8 | 8 | 0 | 0 | 8 | 0 | −8 | −8 | 0 | 0 | −8 | −16 | |
| 1111 | 16 | 0 | 8 | 0 | 0 | −16 | 0 | −8 | 0 | 8 | 8 | 8 | −8 | 0 | −8 | |

## 4.10 Affine function and affine equivalence

### 4.10.1 Description

A Boolean linear function is defined as a Boolean function consisting only of the sum of single input variables. Similarly, the set of Boolean affine functions is defined as the set of linear functions and their complements. A mathematical description of the linear and affine Boolean functions is given as follows.

A Boolean linear function is defined as the sum of a subset of the input variables, denoted $l_{\mathbf{u}}(\mathbf{x}) = u_1x_1 + u_2x_2 + \cdots + u_nx_n$ where $\mathbf{u} = (u_1, \ldots, u_n) \in \mathrm{V_n}$.

The set of Boolean affine functions are the linear functions and their complements, denoted $l_{\mathbf{u},b}(\mathbf{x}) = l_{\mathbf{u}}(\mathbf{x}) + b$ where $b \in \mathrm{GF}(2)$.

An affine Vector Boolean function is defined in terms of a linear Vector Boolean function and a dyadic shift. A linear Vector Boolean function involves the multiplication of the input vector by a Boolean matrix. A dyadic shift (or translation) involves the complement of a subset of input bits. As such, an affine Vector Boolean function may be defined as the combination of a linear Vector Boolean function and dyadic shift. A mathematical description of the linear and affine Vector Boolean functions is given as follows.

A Vector Boolean function $L_{\mathsf{A},\mathbf{b}} \in \mathcal{F}_{n,m}$ defined as $L_{\mathsf{A},\mathbf{b}}(\mathbf{x}) = \mathbf{x} \cdot \mathsf{A} + \mathbf{b}$ with $\mathbf{x} \in \mathrm{V_n}, \mathsf{A} \in \mathsf{M}_{n\times m}(\mathrm{GF}(2))$ and $\mathbf{b} \in \mathrm{V_m}$ so that if $\mathbf{b} = \mathbf{0}$ then $F$ is linear and if $\mathbf{b} \neq \mathbf{0}$ then $F$ is affine.

#### Affine equivalence of Boolean functions

Equivalence classes provide a powerful tool in both the construction and analysis of Boolean functions for cryptography. In particular, rather than considering the entire space of $2^{2^n}$ functions a reduced view can be found in the consideration of only one function from each equivalence class.

If $g(\mathbf{x}) = f(\mathsf{A}\mathbf{x} + \mathbf{b}) + \mathbf{c}\mathbf{x} + d$

where $\mathsf{A} \in \mathsf{M}_{n\times n}(\mathrm{GF}(2))$ non-singular, $\mathbf{b}, \mathbf{c} \in \mathrm{V_n}$ and $d \in \mathrm{GF}(2)$ and it is an affine transformation. The functions $f$ and $g$ satisfying the previous equation are called equivalent under the action of $AGL(n, 2)$.

Of particular interest in the study of equivalence classes is the effect of the affine transformation on the algebraic degree, the Walsh Spectrum and Autocorrelation Spectrum of a Boolean function.

### Frequency distribution of absolute values of Walsh Spectrum

The effect of the application of an affine transformation to a Boolean function on the Walsh Spectrum is to rearrange the values and hence, the Walsh value distributions are invariant under all affine transformations [Preneel:93]:

$$\hat{\chi}_g(\mathbf{u}) = (-1)^{\mathbf{c}\cdot\mathsf{A}^{-1}\mathbf{b}}(-1)^{\mathbf{u}\cdot\mathsf{A}^{-1}\mathbf{b}}\hat{\chi}_f\left(\left(\mathsf{A}^{-1}\right)\mathbf{u} + \left(\mathsf{A}^{-1}\right)\mathbf{c}\right)$$

Thus nonlinearity is also invariant under affine transformation.

### Frequency distribution of absolute values of Autocorrelation Spectrum

The effect of the application of an affine transformation to a Boolean function on the Autocorrelation Spectrum is to rearrange the values and hence, the Autocorrelation value distributions are invariant under all affine transformations [Preneel:93]:

$$r_g(\mathbf{u}) = (-1)^{\mathbf{u}\cdot\mathbf{c}} r_f(\mathsf{A}\mathbf{u})$$

Thus absolute indicator is also invariant under affine transformation.

## 4.10.2 Library

A VBF class can be initialized for a affine Vector Boolean function giving its corresponding matrix and vector by the following method:

```
void putaffine(const NTL::mat_GF2& A,const NTL::vec_GF2&        b)
```

The method used to obtain the Frequency distribution of the absolute values of the Walsh Spectrum is the following:

```
void printFWH(NTL_SNS ostream& s, VBF& F)
```

The method used to obtain the Frequency distribution of the absolute values of the Autocorrelation Spectrum is the following:

```
void printFAC(NTL_SNS ostream& s, VBF& F)
```

### Example

The following program finds out the Walsh Spectrum, Frequency distribution of the absolute values of the Walsh Spectrum, Autocorrelation Spectrum, and Frequency distribution of the absolute values of the Autocorrelation Spectrum of a Vector Boolean function having as input the matrix $\mathsf{A}$ and the vector $\mathbf{b}$ associated with an affine function where:

$$\mathsf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{b} = (0, 1)$$

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 A;
   NTL::vec_GF2 b;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> A;
   input >> b;
   F.putaffine(A,b);
   input.close();

   cout << "The Walsh Spectrum is:" << endl << Walsh(F) << endl << endl;

   cout << "Frequency distribution of the absolute values of
   the Walsh Spectrum:" << endl;
   printFWH(cout,F);
   cout << endl;

   cout << "The Autocorrelation Spectrum is:" << endl << AC(F) << endl;

   cout << "Frequency distribution of the absolute values of
   the Autocorrelation Spectrum:" << endl;
   printFAC(cout,F);
   cout << endl;

   return 0;
}
```

The output of the program would be the following:

```
The Walsh Spectrum is:
[[4 0 0 0]
[0 0 4 0]
[0 -4 0 0]
[0 0 0 -4]
]

Frequency distribution of the absolute values of the Walsh Spectrum:
(0,3),(4,1)
(0,3),(4,1)
(0,3),(4,1)

The Autocorrelation Spectrum is:
[[4 4 4 4]
[4 4 -4 -4]
[4 -4 4 -4]
```

(continues on next page)

(continued from previous page)

```
[4 -4 -4 4]
]
Frequency distribution of the absolute values of the Autocorrelation Spectrum:
(4,4)
(4,4)
(4,4)
```

## 4.11 Cycle structure, fixed points and negated fixed points

### 4.11.1 Description

The *cycle structure* of an invertible vector Boolean function $F \in \mathcal{F}_{n,n}$ (permutation) describes the number of cycles and their length.

A permutation can also be written in a way that groups together the images of a given number under repeated applications of *F*. For example, the permutation:

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 6 & 4 & 7 & 5 & 9 & 1 & 8 & 2 \end{bmatrix}$$

can be written

$$F = (1347)(269)(5)(8)$$

The first group of numbers in parentheses indicates that 1 gets mapped to 3, 3 gets mapped to 4, 4 gets mapped to 7, and 7 gets mapped back to 1. Each of the other groupings is interpreted in a similar way. These groups of numbers are called cycles, and this notation for permutations is referred to as cycle notation. Following are several facts relating to cycles and cycle notation:

1. A cycle of *k* numbers is referred to as a *k*-cycle or a cycle of length *k*; for example, $(1347)$ is a *4*-cycle or a cycle of length *4*.
2. A cycle of one number indicates that the number is mapped to itself, and 1-cycles are often referred to as *fixed points*. In the example above, there are two fixed points: *5* and *8*.
3. It does not matter which number is written first in a cycle, as long as the order of the numbers is preserved. For example, $(1347) = (4713)$, but $(1347) \neq (1437)$.

A cycle structure with a low number of cycles of high length is considered well suited to be used in cipher design. This fact means that many transpositions are present.

The *fixed points* of *F* are those which belong to the set $\{\mathbf{x} \mid F(\mathbf{x}) = \mathbf{x}\}$. The *negated fixed points* of *F* belong to the set $\{\mathbf{x} \mid F(\mathbf{x}) = \overline{\mathbf{x}}\}$ where $\overline{\mathbf{x}}$ is the inverse of $\mathbf{x}$ or the vector resulting from adding *1* to each of its components.

A cryptographic primitive with a high number of fixed and/or negated fixed points is considered to be not well designed, since it lacks the needed randomness.

### 4.11.2 Library

The method used to obtain the Cycle Structure is the following:

```
void Cycle(NTL::vec_ZZ& v, VBF& F)
```

The method used to print the Cycle structure so that each row has two values separated by a comma: the first one is the Cycle length and the second one is the number of cycles for this length.

```
void printCycle(NTL_SNS ostream& s, VBF& F)
```

The fixed points of $F$ are obtained by this method:

```
NTL::mat_GF2 fixedpoints(VBF& F)
```

The negated fixed points of $F$ are obtained by this method:

```
NTL::mat_GF2 negatedfixedpoints(VBF& F)
```

### Example

The following program prints the cycle structure of a Vector Boolean function having as input its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The Cycle Structure is:" << endl;
   printCycle(cout,F);

   cout << endl << "The fixed points are the following:"
   << endl;
   cout << fixedpoints(F) << endl;

   cout << endl << "The negated fixed points are the following:"
   << endl;
   cout << negatedfixedpoints(F) << endl;

   return 0;
}
```

If we use as input of this program the Truth Table of *NibbleSub*, the output of the program would be the following:

```
The Cycle Structure is:
2,1
14,1

The fixed points are the following:
```

(continues on next page)

(continued from previous page)

```
[]

The negated fixed points are the following:
[[0 0 1 0]
[0 1 1 1]
]
```

which means:

| Cycle structure of *NibbleSub* | |
|---|---|
| Cycle length | Number of cycles |
| 2 | 1 |
| 14 | 1 |

It has no fixed points and 2 negated fixed points which are the following:

```
[0 0 1 0]
[0 1 1 1]
```

This is because $NibbleSub[(1,1,0,1)] = (0,0,1,0)$ and $NibbleSub[(1,0,0,0)] = (0,1,1,1)$.

## 4.12 Permutation matrix

### 4.12.1 Description

If *F* is a Boolean permutation, that is, it is bijective and has the same number of input bits as output bits *(n=m)*, then it can be defined as an array: $F = \begin{bmatrix} F(1) & \dots & F(n) \end{bmatrix}$ where $F(i)$ is the output bit of the input bit *i* for *F*.

### 4.12.2 Library

A VBF class can be initialized giving its permutation vector with the following method:

```
void putper(const NTL::vec_ZZ& v)
```

To obtain its representation as permutation vector, the following method must be used:

```
void PER(NTL::vec_ZZ& v, VBF& F)
```

#### Example

The following program finds out the Truth Table of a Vector Boolean function having as input its Permutation Vector:

```
[ 1 2 3 4 13 14 15 16 9 10 11 12 5 6 7 8 ]
```

For example, you can see bit 13 moves to bit 5, while bit 5 moves to bit 13.

```
#include <iostream>
#include <fstream>
#include "VBF.h"
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::vec_ZZ  a;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> a;
   F.putper(a);
   input.close();

   cout << "The Truth Table is:" << endl;
   cout << TT(F) << endl;

   return 0;
}
```

The first 10 lines of the output of the program would be the following:

```
The Truth Table is:
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
```

## 4.13 DES representations

The VBF library accepts to types of representations of DES [DES:77] components:

1. *Expansion and Compression DES permutations*. It can be used to represent both the Compression Permutation in the Key Transformation of DES and the Expansion Permutation Feistel Function of the DES cipher. The Compression Permutation permutes the order of the bits as well as selects a subset of bits. The Expansion Permutation expands the right half of the data, $R_i$, from 32 bits to 48 bits. Because this operation changes the order of the bits as well as repeating certain bits, it is known as an expansion permutation.

2. *DES S-box Substitution*. Each S-box is a table of 4 rows and 16 columns. Each entry in the box is a 4-bit number. The 6 input bits of the S-box specify under which row and column number to look for the output.

   The input bits specify an entry in the S-box as follows: Consider an S-box input of 6-bits, labeled $b_1, b_2, b_3, b_4, b_5$, and $b_6$. Bits $b_1$ and $b_6$ are combined to form a 2-bit number, from 0 to 3, which corresponds to a row in the table. The middle 4 bits, $b_2$ through $b_5$, are combined to form a 4-bit number, from 0 to 15, which corresponds to a column in the table.

   For example, assume that the input to the first S-box (i.e. bits 1 to 6 of the XOR function) is *110011*. The first and last bits combine to form *11*, which corresponds to row 3 of the first S-box. The middle 4 bits combine to form *1001*, which corresponds to the column 9 of the same S-box. The entry under

row 3, column 9 of S-box 1 is 11 (count rows and columns starting from 0). The value *1110* is substituted for *001011*

The following figures list the eight S-boxes used in DES. Each S-box replaces a 6-bit input with a 4-bit output. Given a 6-bit input, the 4-bit output is found by selecting the row using the outer two bits, and the column using the inner four bits. For example, an input "011011" has outer bits "01" and inner bits "1101"; noting that the first row is "00" and the first column is "0000", the corresponding output for S-box S5 would be "1001" (=9), the value in the second row, 14th column.

| $S_1$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **x0000x** | **x0001x** | **x0010x** | **x0011x** | **x0100x** | **x0101x** | **x0110x** | **x0111x** | **x1000x** | **x1001x** | **x1010x** | **x1011x** | **x1100x** | **x1101x** |
| **0yyyy0** | 14 | 4 | 13 | 1 | 2 | 15 | 11 | 8 | 3 | 10 | 6 | 12 | 5 | 9 |
| **0yyyy1** | 0 | 15 | 7 | 4 | 14 | 2 | 13 | 1 | 10 | 6 | 12 | 11 | 9 | 5 |
| **1yyyy0** | 4 | 1 | 14 | 8 | 13 | 6 | 2 | 11 | 15 | 12 | 9 | 7 | 3 | 10 |
| **1yyyy1** | 15 | 12 | 8 | 2 | 4 | 9 | 1 | 7 | 5 | 11 | 3 | 14 | 10 | 0 |
| $S_2$ | | | | | | | | | | | | | | |
| | **x0000x** | **x0001x** | **x0010x** | **x0011x** | **x0100x** | **x0101x** | **x0110x** | **x0111x** | **x1000x** | **x1001x** | **x1010x** | **x1011x** | **x1100x** | **x1101x** |
| **0yyyy0** | 15 | 1 | 8 | 14 | 6 | 11 | 3 | 4 | 9 | 7 | 2 | 13 | 12 | 0 |
| **0yyyy1** | 3 | 13 | 4 | 7 | 15 | 2 | 8 | 14 | 12 | 0 | 1 | 10 | 6 | 9 |
| **1yyyy0** | 0 | 14 | 7 | 11 | 10 | 4 | 13 | 1 | 5 | 8 | 12 | 6 | 9 | 3 |
| **1yyyy1** | 13 | 8 | 10 | 1 | 3 | 15 | 4 | 2 | 11 | 6 | 7 | 12 | 0 | 5 |
| $S_3$ | | | | | | | | | | | | | | |
| | **x0000x** | **x0001x** | **x0010x** | **x0011x** | **x0100x** | **x0101x** | **x0110x** | **x0111x** | **x1000x** | **x1001x** | **x1010x** | **x1011x** | **x1100x** | **x1101x** |
| **0yyyy0** | 10 | 0 | 9 | 14 | 6 | 3 | 15 | 5 | 1 | 13 | 12 | 7 | 11 | 4 |
| **0yyyy1** | 13 | 7 | 0 | 9 | 3 | 4 | 6 | 10 | 2 | 8 | 5 | 14 | 12 | 11 |
| **1yyyy0** | 13 | 6 | 4 | 9 | 8 | 15 | 3 | 0 | 11 | 1 | 2 | 12 | 5 | 10 |
| **1yyyy1** | 1 | 10 | 13 | 0 | 6 | 9 | 8 | 7 | 4 | 15 | 14 | 3 | 11 | 5 |
| $S_4$ | | | | | | | | | | | | | | |
| | **x0000x** | **x0001x** | **x0010x** | **x0011x** | **x0100x** | **x0101x** | **x0110x** | **x0111x** | **x1000x** | **x1001x** | **x1010x** | **x1011x** | **x1100x** | **x1101x** |
| **0yyyy0** | 7 | 13 | 14 | 3 | 0 | 6 | 9 | 10 | 1 | 2 | 8 | 5 | 11 | 12 |
| **0yyyy1** | 13 | 8 | 11 | 5 | 6 | 15 | 0 | 3 | 4 | 7 | 2 | 12 | 1 | 10 |
| **1yyyy0** | 10 | 6 | 9 | 0 | 12 | 11 | 7 | 13 | 15 | 1 | 3 | 14 | 5 | 2 |
| **1yyyy1** | 3 | 15 | 0 | 6 | 10 | 1 | 13 | 8 | 9 | 4 | 5 | 11 | 12 | 7 |

| $S_5$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | x0000x | x0001x | x0010x | x0011x | x0100x | x0101x | x0110x | x0111x | x1000x | x1001x | x1010x | x1011x | x1100x | x1101x |
| 0yyyy0 | 2 | 12 | 4 | 1 | 7 | 10 | 11 | 6 | 8 | 5 | 3 | 15 | 13 | 0 |
| 0yyyy1 | 14 | 11 | 2 | 12 | 4 | 7 | 13 | 1 | 5 | 0 | 15 | 10 | 3 | 9 |
| 1yyyy0 | 4 | 2 | 1 | 11 | 10 | 13 | 7 | 8 | 15 | 9 | 12 | 5 | 6 | 3 |
| 1yyyy1 | 11 | 8 | 12 | 7 | 1 | 14 | 2 | 13 | 6 | 15 | 0 | 9 | 10 | 4 |
| $S_6$ | | | | | | | | | | | | | | |
| | x0000x | x0001x | x0010x | x0011x | x0100x | x0101x | x0110x | x0111x | x1000x | x1001x | x1010x | x1011x | x1100x | x1101x |
| 0yyyy0 | 12 | 1 | 10 | 15 | 9 | 2 | 6 | 8 | 0 | 13 | 3 | 4 | 14 | 7 |
| 0yyyy1 | 10 | 15 | 4 | 2 | 7 | 12 | 9 | 5 | 6 | 1 | 13 | 14 | 0 | 11 |
| 1yyyy0 | 9 | 14 | 15 | 5 | 2 | 8 | 12 | 3 | 7 | 0 | 4 | 10 | 1 | 13 |
| 1yyyy1 | 4 | 3 | 2 | 12 | 9 | 5 | 15 | 10 | 11 | 14 | 1 | 7 | 6 | 0 |
| $S_7$ | | | | | | | | | | | | | | |
| | x0000x | x0001x | x0010x | x0011x | x0100x | x0101x | x0110x | x0111x | x1000x | x1001x | x1010x | x1011x | x1100x | x1101x |
| 0yyyy0 | 4 | 11 | 2 | 14 | 15 | 0 | 8 | 13 | 3 | 12 | 9 | 7 | 5 | 10 |
| 0yyyy1 | 13 | 0 | 11 | 7 | 4 | 9 | 1 | 10 | 14 | 3 | 5 | 12 | 2 | 15 |
| 1yyyy0 | 1 | 4 | 11 | 13 | 12 | 3 | 7 | 14 | 10 | 15 | 6 | 8 | 0 | 5 |
| 1yyyy1 | 6 | 11 | 13 | 8 | 1 | 4 | 10 | 7 | 9 | 5 | 0 | 15 | 14 | 2 |
| $S_8$ | | | | | | | | | | | | | | |
| | x0000x | x0001x | x0010x | x0011x | x0100x | x0101x | x0110x | x0111x | x1000x | x1001x | x1010x | x1011x | x1100x | x1101x |
| 0yyyy0 | 13 | 2 | 8 | 4 | 6 | 15 | 11 | 1 | 10 | 9 | 3 | 14 | 5 | 0 |
| 0yyyy1 | 1 | 15 | 13 | 8 | 10 | 3 | 7 | 4 | 12 | 5 | 6 | 11 | 0 | 14 |
| 1yyyy0 | 7 | 11 | 4 | 1 | 9 | 12 | 14 | 2 | 0 | 6 | 10 | 13 | 15 | 3 |
| 1yyyy1 | 2 | 1 | 14 | 7 | 4 | 10 | 8 | 13 | 15 | 12 | 9 | 0 | 3 | 5 |

### 4.13.1 Library

A VBF class can be initialized giving its Expansion and Compression DES permutation vector with the following method:

```
void putexp_comp(const NTL::vec_ZZ& v)
```

A VBF class can be initialized giving its DES-like S-box representation matrix with the following method:

```
void putsbox(const NTL::mat_ZZ& S)
```

### Example

The following program prints the Truth Table of a Expansion permutation and of the DES S1 S-box. The inputs are respectively the following:

```
[ 4 1 2 3 4 1 ]
```

```
[[14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7 ]
[ 0 15 7 4 14 2 13 1 10 6 12 11 9 5 3 8 ]
```

(continues on next page)

(continued from previous page)

```
[ 4 1 14 8 13 6 2 11 15 12 9 7 3 10 5 0]
[ 15 12 8 2 4 9 1 7 5 11 3 14 10 0 6 13]]
```

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF F,G;
   NTL::vec_ZZ v;
   NTL::mat_ZZ S;

   ifstream inputv(argv[1]);
   if(!inputv) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   inputv >> v;
   inputv.close();
   F.putexp_comp(v);

   ifstream inputS(argv[2]);
   if(!inputS) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   inputS >> S;
   inputS.close();
   G.putsbox(S);

   cout << "The Truth Table of Expansion Permutation is:"
   << endl << TT(F) << endl;
   cout << endl << "The Truth Table of S1 DES S-box is:"
   << endl << TT(G) << endl;

   return 0;
}
```

The output of the program would be the following (Only a few values of S1 Truth Table is printed for space reasons):

```
The Truth Table of Expansion Permutation is:
[[0 0 0 0 0 0]
[1 0 0 0 1 0]
[0 0 0 1 0 0]
[1 0 0 1 1 0]
[0 0 1 0 0 0]
[1 0 1 0 1 0]
[0 0 1 1 0 0]
[1 0 1 1 1 0]
[0 1 0 0 0 1]
[1 1 0 0 1 1]
[0 1 0 1 0 1]
[1 1 0 1 1 1]
```

(continues on next page)

(continued from previous page)

```
[0 1 1 0 0 1]
[1 1 1 0 1 1]
[0 1 1 1 0 1]
[1 1 1 1 1 1]
]

The Truth Table of S1 DES S-box is:
[[1 1 1 0]
[0 0 0 0]
[0 1 0 0]
[1 1 1 1]
[1 1 0 1]
[0 1 1 1]
[0 0 0 1]
[0 1 0 0]
[0 0 1 0]
[1 1 1 0]
[1 1 1 1]
[0 0 1 0]
[1 0 1 1]
[1 1 0 1]
[1 0 0 0]
[0 0 0 1]
...
```

## 4.14 Auxiliary functions

In order to compute the matrices described above, some functions have been implemented which allow to obtain some of these matrices from others:

1. A function whose input is an ANF table and its output is the Truth Table: `mat_GF2 rev(const mat_GF2& A, int n, int m)`.
2. A function whose input is the Characteristic Function and its output is the Truth Table: `mat_GF2 truthtable(const mat_ZZ& C, int n, int m)`.
3. A function whose input is the Truth Table and its output is the Characteristic Function: `mat_ZZ charfunct(const mat_GF2& T, int n, int m)`.
4. A function whose input is the Walsh Spectrum and its output is the Characteristic Function (that is the Inverse Walsh Transform): `mat_ZZ invwt(const mat_ZZ& X, int n, int m)`.
5. A matrix representing the linear combinations of Truth Table coordinate functions `void LTT(NTL::mat_GF2& X, VBF& a)`.
6. A matrix representing character form of Truth Table coordinate functions `void CTT(NTL::mat\_GF2\& X, VBF\& a)`.

## 4.15 Summary

Table Representations Table lists the member functions related to methods of vector Boolean functions initialization. Table Characterizations Table lists the member functions related to the characterizations of vector Boolean functions as described above. Most of the member functions of *VBF* have an in-line definition, for instance: `void TT(NTL::mat_GF2& X, VBF& F)` is also defined as `inline NTL::mat_GF2 TT(VBF& F)`.

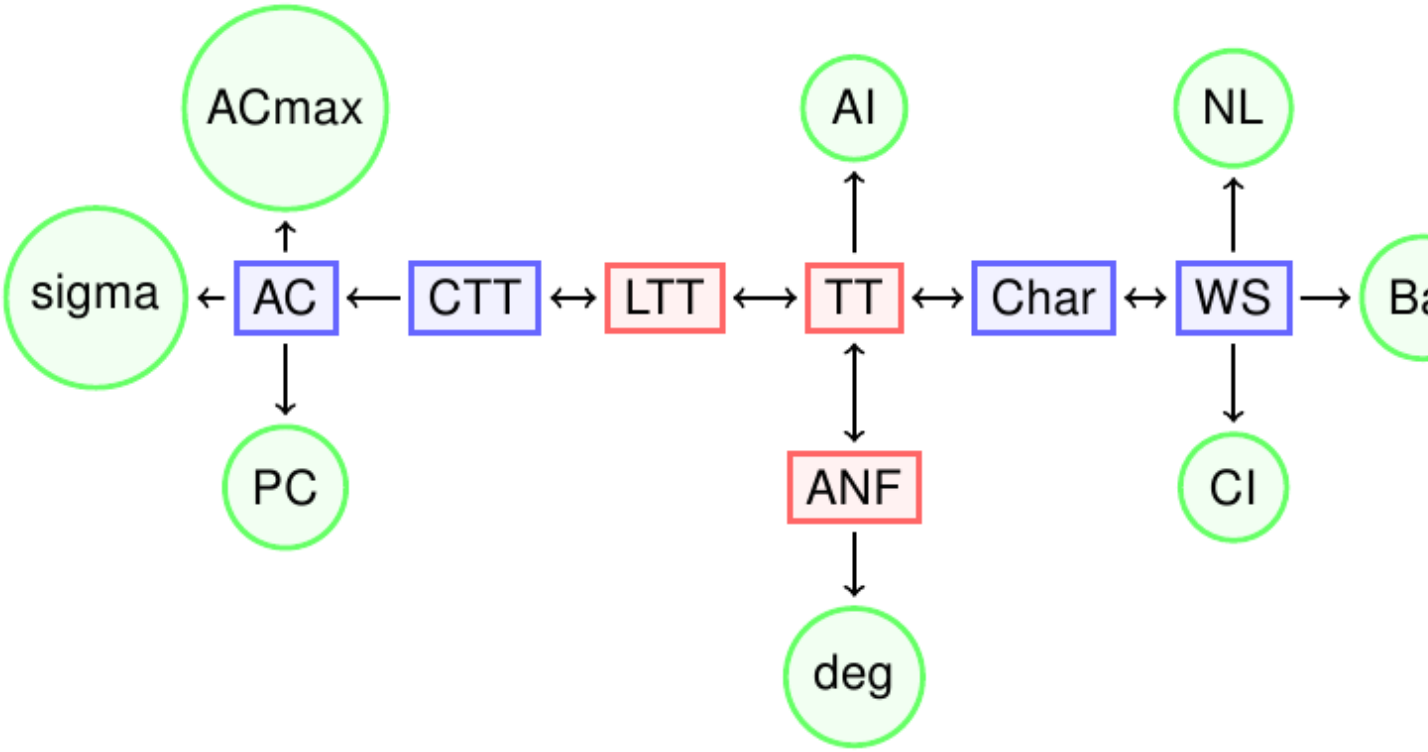
| Representation of VBF | |
|---|---|
| **SYNTAX** | **DESCRIPTION** |
| `void puttt(const NTL::mat_GF2& X)` | $\mathsf{TT}_F = \mathsf{T}$ |
| `void putHexTT(istream& s)` | VBF which has an hexadecimal representation of its truth table defined by *s* |
| `void putDecTT(const NTL::vec_long& d,const long& m)` | VBF with a decimal representation of its truth table defined by *d* and *m* is the number of component Boolean functions |
| `void putltt(const NTL::mat_GF2& L)` | $\mathsf{LTT}_F = \mathsf{L}$ |
| `void putctt(const NTL::mat_ZZ& C)` | $\mathsf{CTT}_F = \mathsf{C}$ |
| `void putirrpol(GF2X& g)  void puttrace(string& f)` | Set *F* by its trace *f* and the irreducible polynomial *g* |
| `void putpol(vec_pol& p)` | Set *F* with Polynomials in *ANF* equals to *p* |
| `void putanf(const NTL::mat_GF2& A)` | $\mathsf{ANF}_F = \mathsf{A}$ |
| `void putchar(const NTL::mat_ZZ& C)` | $\mathsf{Img}(F) = \mathsf{C}$ |
| `void putwalsh(const NTL::mat_ZZ& W)` | $\mathsf{WS}(F) = \mathsf{W}$ |
| `void putper(const NTL::vec_ZZ& v)` | VBF which is a permutation defined by $\mathbf{v}$ |
| `void putexp_comp(const NTL::vec_ZZ& v)` | VBF defined by Expansion and Compression DES vector $\mathbf{v}$ |
| `void putsbox(const NTL::mat_ZZ& S)` | VBF which is a DES S-Box defined by *S* |

| Characterization of VBF | |
|---|---|
| **SYNTAX** | **DESCRIPTION** |
| `void TT(NTL::mat_GF2& X, VBF& F)` | $\mathsf{X} = \mathsf{T}_F$ |
| `void getHexTT(ostream& s)` | *s* is the hexadecimal representation of the truth table of *F* |
| `NTL::vec_long getDecTT() const` | Decimal representation of the truth table |
| `long weight(VBF& F)` | Weight of *F* |
| `void LTT(NTL::mat_GF2& X, VBF& F)` | $\mathsf{X} = \mathsf{LTT}_F$ |
| `void CTT(NTL::mat_ZZ& X, VBF& F)` | $\mathsf{X} = \mathsf{CTT}_F$ |
| `void Trace(GF2EX& f, VBF& F)` | *F* has a trace representation defined by *f* |
| `void Pol(NTL_SNS ostream& s, VBF& F)` | *s* contains the Polynomials in *ANF* of *F* |
| `void ANF(NTL::mat_GF2& X, VBF& F)` | $\mathsf{X} = \mathsf{ANF}_F$ |
| `void Charact(NTL::mat_ZZ& X, VBF& F)` | $\mathsf{X} = \mathsf{Img}(\mathsf{F})$ |
| `void Walsh(NTL::mat_ZZ& X, VBF& F)` | $\mathsf{X} = \mathsf{WS}(F)$ |
| `void LAT(NTL::mat_ZZ& X, VBF& F)` | $\mathsf{X} = \mathsf{LP}(F)$ |
| `void lp(NTL::RR& x, VBF& F)` | $lp(F) = x$ |
| `void linear(NTL_SNS ostream& s, VBF& F, ZZ& x)` | Linear relations associated with the value *x* of the Linear Profile of *F* |
| `void ProbLin(NTL::RR& x, VBF& F, NTL::ZZ& w)` | Probability of Linear relations associated with the value *w* of the Linear Profile of *F* |
| `void DAT(NTL::mat_ZZ& X, VBF& F)` | $\mathsf{X} = \mathsf{DP}(F)$ |
| `void dp(NTL::RR& x, VBF& F)` | $dp(F) = x$ |
| `void differential(NTL_SNS ostream& s, VBF& F, ZZ& x)` | Differential relations associated with the value *x* of the Differential Profile of *F* |
| `void ProbDif(NTL::RR& x, VBF& F, NTL::ZZ& w)` | Probability of characteristics associated with the value *w* of the Differential Profile of *F* |
| `void AC(NTL::mat_ZZ& X, VBF& F)` | $\mathsf{X} = \mathsf{R}(F)$ |
| `NTL::mat_GF2 LS(VBF& F)` | Returns a matrix whose rows are the linear structures |
| `void printFWH(NTL_SNS ostream& s, VBF& F)` | Frequency distribution of the absolute values of the Walsh Spectrum |
| `void printFAC(NTL_SNS ostream& s, VBF& F)` | Frequency distribution of the absolute values of the Autocorrelation Spectrum |
| `void Cycle(NTL::vec_ZZ& v, VBF& F)` | **v** is the Cycle Structure |
| `void printCycle(NTL_SNS ostream& s, VBF& F)` | Print Cycle Structure |
| `NTL::mat_GF2 fixedpoints(VBF& F)` | Return fixed points |
| `NTL::mat_GF2 negatedfixedpoints(VBF& F)` | Return negated fixed points |
| `void PER(NTL::vec_ZZ& v, VBF& F)` | **v** is the permutation vector defined by *F* |

# 5 Cryptographic Criteria

This chapter defines some properties relevant for cryptographic applications and explains how to use the package to compute them. Those properties are criteria or those which provide useful information in cryptanalysis. Among the criteria we find nonlinearity, r-th order nonlinearity, linearity distance, balancedness, correlation immunity, resiliency (i.e. balancedness and correlation immunity), propagation criterion, global avalanche criterion, algebraic degree and algebraic immunity. Other properties described are linear potential, differential potential, linear or differential relations associated with a specific value, linear structures, the maximum possible nonlinearity or the maximum possible linearity distance achievable by a Vector Boolean Function with the same number of inputs, the type of function in terms of nonlinearity, the Frequency distribution of the absolute values of the Walsh Spectrum or the Autocorrelation Spectrum, its cycle structure, the presence of fixed points or negated fixed points.

The figure summarizes the relationships among several representations and the criteria studied in this chapter.




The representations which are Boolean matrices are coloured in red, those which are integer matrices are coloured in blue, and those which are criteria are coloured in green.

## 5.1 Algebraic degree

### 5.1.1 Description

Cryptographic algorithms using Boolean functions to achieve confusion in a cipher (S-boxes in block ciphers, combining of filtering functions in stream ciphers) can be attacked if the functions have low algebraic degree. The algebraic degree is a good indicator of the function's algebraic complexity. The higher the degree of a function, the greater is its algebraic complexity. *Higher order differential attack* [Lai:94] exploits the fact that the algebraic degree of the S-box is low.

The *algebraic degree* of a Vector Boolean function $F \in \mathcal{F}_{n,m}$ is defined as the minimum among the algebraic degrees of all component functions of $F$ [Nyberg:92], namely:

$$deg(F) = \min_{g}\{deg(g) \mid g = \sum_{j=1}^{m} v_j f_j,\ \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}\}$$

where the algebraic order or degree of a Boolean function is the order of the largest product term in the *ANF*. This criterion is obtained by generating the ANF table and then analyzing the degree of all the component functions.

Functions with algebraic degree less than or equal to 1 are called affine. A non-constant affine function for which $F(\mathbf{0}) = \mathbf{0}$ is called linear. We refer to functions of degree two as quadratic and functions of degree three as cubic.

### 5.1.2 Library

The method used to obtain this criterion is the following:

```
void deg(int& d, VBF& F)
```

### Example

The following program provides the algebraic degree of a Vector Boolean function given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The algebraic degree of the function is "
   << deg(F) << endl;

   return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input we will find out that its algebraic degree is 2.

The following figure represents the ANF table of NibbleSub nonzero component functions and emphasizes in red the ANF terms of degree 4. As we can see there are no terms of degree 4 in neither of the component functions of NibbleSub.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0001 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0010 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0011 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0100 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0101 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0110 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0111 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1000 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1001 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1010 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1011 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1100 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1101 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1110 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The following figure represents the ANF table of NibbleSub nonzero component functions and emphasizes in blue the ANF terms of degree 3. As we can see there are no terms of degree 3 in one of the component functions of NibbleSub, which is marked in yellow.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0001 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0010 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0011 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0100 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0101 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0110 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0111 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1000 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1001 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1010 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1011 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1100 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1101 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1110 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The following figure represents the ANF table of NibbleSub nonzero component functions and emphasizes in orange the ANF terms of degree 2. As we can see there are always terms of degree 2 in all the component functions of NibbleSub. Because of this, the algebraic degree of NibbleSub is 2.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0001 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0010 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0011 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0100 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0101 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0110 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0111 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1000 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1001 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1010 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1011 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1100 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1101 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1110 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## 5.2 Nonlinearity

### 5.2.1 Description

In order to provide confusion, cryptographic functions must lie at large Hamming distance to all affine functions. Because of Parseval's Relation, any Vector Boolean function has correlation with some affine functions of its input. This correlation should be small: the existence of affine approximations of the Boolean functions involved in a cipher allows to build attacks on this system (see, [Matsui:93] for block ciphers and [DingXS:91] for stream ciphers).

The nonlinearity of a Boolean function $f \in \mathcal{F}_m$ is defined as the Hamming distance between $f$ and the subspace of affine functions [PieprzykF:88]: $NL(f) = d(f, \mathsf{A}_n)$.

The nonlinearity of a Vector Boolean function $F \in \mathcal{F}_{n,m}$ is defined as the minimum among the nonlinearities of all component functions of $F$ [Nyberg:92]:

$$NL(F) = \min_{\mathbf{v} \neq \mathbf{0}} NL(\mathbf{v} \cdot F) \;\; \mathbf{v} = (v_1, \ldots, v_m) \in \mathrm{V_m}$$

The nonlinearity of $F$ can be expressed in terms of the Walsh coefficients by the following theorem:

Let $F \in \mathcal{F}_{n,m}$, the nonlinearity of $F$ can be calculated in terms of the maximum of the absolute values of its Walsh Spectrum without taking into account the element of its first row and column, as follows:

$$NL(F) = 2^{n-1} - \frac{1}{2} \overset{*}{\max} \left(\mathsf{WS}(F)(\mathbf{u}, \mathbf{v})\right)$$

Let $f \in \mathcal{F}_n$, the nonlinearity of $f$ can be expressed in terms of its Walsh transform as follows:

$$NL(f) = 2^{n-1} - \frac{1}{2} \max_{\mathbf{u} \in \mathrm{V_n}} |\hat{\chi}_f(\mathbf{u})|$$

The *spectral radius* of a Boolean function $f \in \mathcal{F}_n$ is $r(f) = \max_{\mathbf{u} \in \mathrm{V_n}} |\hat{\chi}_f(\mathbf{u})|$.

This criterion is a measure of the distance of a Vector Boolean function and all Affine Vector Boolean functions. If this distance is small, it is possible to mount affine approximations of the Vector Boolean functions involved in a cipher

to build attacks (called *linear attacks*) on a block cipher [Matsui:94]. In the case of stream ciphers, these attacks are called *fast correlation attacks*. Thus, this property is useful to assess the resistance of a Vector Boolean function to linear attacks (including correlation attacks), i.e., attacks where the function $F$ is approximated by an affine function.

### 5.2.2 Library

The method used to obtain the nonlinearity of a Vector Boolean function is the following:

```
void nl(NTL::RR& x, VBF& F)
```

The method used to obtain the spectral radius of a Vector Boolean function is the following:

```
void SpectralRadius(NTL::ZZ& x, VBF& F)
```

The method used to the maximum nonlinearity that can be achieved by a Vector Boolean function with the same number of input bits and output bits is the following:

```
NTL::RR nlmax(VBF& F)
```

The method used to obtain the type of function in terms of nonlinearity is the following:

```
void typenl(int& typenl, VBF& F)
```

### Example 1

The following program provides the nonlinearity of a Vector Boolean function given its Truth Table together with the maximum nonlinearity that can be achieved by a Vector Boolean function with the same number of input bits and output bits.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The spectral radius of the function is " << SpectralRadius(F)
           << endl;
   cout << "The nonlinearity of the function is " << nl(F) << endl;

   cout << "The maximum nonlinearity that can be achieved by
```

(continues on next page)

(continued from previous page)

```
    a Vector Boolean function with the same dimensions is "
          << nlmax(F) << endl;

    return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input, the output would be the following:

```
The spectral radius of the function is 12
The nonlinearity of the function is 2
The maximum nonlinearity that can be achieved by
a Vector Boolean function with the same dimensions is 5
```

The following figure represents the Walsh Spectrum of NibbleSub and emphasizes in blue its maximum absolute values.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | −4 | −4 | 0 | 0 | −4 | 12 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 |
| 0 | 0 | −4 | −4 | 0 | 0 | −4 | −4 | 0 | 0 | 4 | 4 | 0 | 0 | −12 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | −12 | −4 | −4 | 4 | 4 | −4 | −4 |
| 0 | 4 | 0 | −4 | −4 | −8 | −4 | 0 | 0 | −4 | 0 | 4 | 4 | −8 | 4 | 0 |
| 0 | −4 | −4 | 0 | −4 | 0 | 8 | 4 | −4 | 0 | −8 | 4 | 0 | −4 | −4 | 0 |
| 0 | 4 | −4 | 8 | 4 | 0 | 0 | 4 | 0 | −4 | 4 | 8 | −4 | 0 | 0 | −4 |
| 0 | −4 | 0 | 4 | 4 | −8 | 4 | 0 | −4 | 0 | 4 | 0 | 8 | 4 | 0 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | −4 | 4 | 4 | −4 | 4 | −4 | −4 | −1 |
| 0 | 0 | −4 | −4 | 0 | 0 | −4 | −4 | −8 | 0 | −4 | 4 | 0 | 8 | 4 | −4 |
| 0 | 8 | −4 | 4 | −8 | 0 | 4 | −4 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 |
| 0 | 8 | 0 | −8 | 8 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | −4 | 8 | −4 | −4 | 0 | 4 | 0 | 4 | 0 | 4 | 8 | 0 | 4 | 0 | −4 |
| 0 | 4 | 4 | 0 | −4 | 8 | 0 | 4 | −8 | −4 | 4 | 0 | 4 | 0 | 0 | 4 |
| 0 | 4 | 4 | 0 | −4 | −8 | 0 | 4 | −4 | 0 | 0 | −4 | −8 | 4 | −4 | 0 |
| 0 | −4 | −8 | −4 | −4 | 0 | 4 | 0 | 0 | −4 | 8 | −4 | −4 | 0 | 4 | 0 |

From definition we have $NL(NibbleSub) = 2^{4-1} - \frac{1}{2} \cdot 12 = 2$

### Example 2

The following program provides the nonlinearity of a Vector Boolean function given its polynomial representation in ANF together with the maximum nonlinearity that can be achieved by a Vector Boolean function with the same number of input bits and output bits, and the type of function in terms of nonlinearity.

```
#include <iostream>
#include <fstream>
```

(continues on next page)

(continued from previous page)

```
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   vec_pol p;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> p;
   F.putpol(p);
   input.close();

   cout << "The nonlinearity of the function is " << nl(F) << endl;
   cout << "The maximum nonlinearity that can be achieved by
   a Vector Boolean function with the same dimensions is "
   << nlmax(F) << endl;

   int type;
   typenl(type, F);

   if (type == BENT) {
     cout << "It is a bent function" << endl;
   } else if (type == ALMOST_BENT) {
     cout << "It is an almost bent function" << endl;
   } else if (type == LINEAR) {
     cout << "It is a linear function" << endl;
   }

   return 0;
}
```

If we use the $x_1x_2 + x_3x_4$ as input, the output would be the following:

```
The nonlinearity of the function is 6
The maximum nonlinearity that can be achieved by
a Vector Boolean function with the same dimensions is 6
It is a bent function
```

As the nonlinearity of this Boolean function is maximal, it is a bent function.

## 5.3 r-th order nonlinearity

### 5.3.1 Description

As well as the affine functions, we can consider that functions with low algebraic degree are weak functions from the cryptographic point of view. A criterion can be defined en terms of the Hamming distance to the Reed-Muller code of order $r(r < n)$.

For every positive integer *r*, the *r-th order nonlinearity* of a Vector Boolean function *F* is the minimum *r*-th order nonlinearity of its component functions. The *r*-th order nonlinearity of a Boolean function equals its minimum Hamming

distance to functions of algebraic degrees at most $r$ (see [carlet2008higher] for details).

$$NL_r(F) = \min_{\mathbf{v}\neq\mathbf{0}\in \mathrm{V_m}} NL_r(\mathbf{v}\cdot F) = \min_{\mathbf{v}\neq\mathbf{0}\in \mathrm{V_m}} \min_{f\in\mathcal{F}_n} d(f, \mathbf{v}\cdot F)$$

Computing r-th order nonlinearity is not an easy task for $r \geq 2$. Unlike the first-order nonlinearity there are no efficient algorithms to compute second-order nonlinearities for $n \geq 11$. VBF library naive exhaustive search is employed for this purpose.

### 5.3.2 Library

The method used to obtain this criterion is the following:

```
void nlr(long& x, VBF& F, int r)
```

This method return -1 if the number of functions to check is too large (greater than the maximum value of a long int variable).

### Example

The following program provides the *2*-nd order nonlinearity of a Vector Boolean function given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;
   long a;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   nlr(a,F,2);
   cout << "The 2-nd order nonlinearity of the function is "
   << a << endl;

   return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input, the output would be the following:

```
The 2-nd order nonlinearity of the function is 0
```

This result is congruent to the fact that its algebraic degree is *2*.

## 5.4 Balancedness

### 5.4.1 Description

The output of a Vector Boolean function $F \in \mathcal{F}_{n,m}$ used in a cipher must be uniformly distributed over $\mathrm{V_m}$ for avoiding statistical dependence between the plaintext and the ciphertext (which can be used in attacks).

$F \in \mathcal{F}_{n,m}$ is balanced (or has balanced output) if each possible output $m$-tuple occurs with equal probability $2^{-m}$. This criterion can be evaluated from the Walsh Spectrum in the following way: $\hat{\theta}_F(\mathbf{0}, \mathbf{v}) = 0, \ \forall \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}$.

$f \in \mathcal{F}_n$ is balanced if and only if the Walsh coefficient at $\mathbf{0}$ is zero. $F \in \mathcal{F}_{n,m}$ is balanced if and only if the first row of its Walsh Spectrum has all its elements equal to zero except from the first entry.

The imbalance of a Boolean function is defined to be $I(f) = |wt(f) - 2^{n-1}| = 2^{n-1}|\mathrm{C}(f, 0)|$ where *0* indicates the constant zero Boolean function.

Imbalance is defined as the minimum Hamming distance to a balanced function and is therefore directly proportional to the magnitude of the correlation with the constant zero Boolean function. Thus, when imbalance is zero, the function is balanced. Balancedness is a fundamental cryptographic criterion as an imbalanced function has suboptimal unconditional entropy, i.e. it is correlated to a constant function.

The significance of the balancedness criterion is that the higher the magnitude of a function's imbalance (deviation from uniform distribution of outputs), the more likelihood of a high probability linear approximation being obtained. This, in turn, represents a weakness in the function in terms of linear cryptanalysis. In particular, a large imbalance may enable the function to be easily approximated by a constant function.

### 5.4.2 Library

This criterion can only take values *0* (meaning *F* is not balanced) or *1* (meaning *F* is balanced). The method used to obtain this criterion is the following:

```
void Bal(int& bal, VBF& F)
```

and there is also an inline function:

```
inline int Bal(VBF& a)
```

### Example

The following program finds out if a Vector Boolean function is balanced given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
```

(continues on next page)

(continued from previous page)

```
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   if (Bal(F)) {
      cout << "It is a balanced function" << endl;
   } else {
      cout << "It is not a balanced function" << endl;
   }

   return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input, the output would be the following:

```
It is a balanced function
```

*NibbleSub* S-box is balanced as each possible 4-tuple occurs with equal probability $\frac{1}{2^4}$.

The following figure represents the Walsh Spectrum of *NibbleSub* and emphasizes in red the first row.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | −4 | −4 | 0 | 0 | −4 | 12 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 |
| 0 | 0 | −4 | −4 | 0 | 0 | −4 | −4 | 0 | 0 | 4 | 4 | 0 | 0 | −12 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | −12 | −4 | −4 | 4 | 4 | −4 | − |
| 0 | 4 | 0 | −4 | −4 | −8 | −4 | 0 | 0 | −4 | 0 | 4 | 4 | −8 | 4 | 0 |
| 0 | −4 | −4 | 0 | −4 | 0 | 8 | 4 | −4 | 0 | −8 | 4 | 0 | −4 | −4 | 0 |
| 0 | 4 | −4 | 8 | 4 | 0 | 0 | 4 | 0 | −4 | 4 | 8 | −4 | 0 | 0 | − |
| 0 | −4 | 0 | 4 | 4 | −8 | 4 | 0 | −4 | 0 | 4 | 0 | 8 | 4 | 0 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | −4 | 4 | 4 | −4 | 4 | −4 | −4 | − |
| 0 | 0 | −4 | −4 | 0 | 0 | −4 | −4 | −8 | 0 | −4 | 4 | 0 | 8 | 4 | − |
| 0 | 8 | −4 | 4 | −8 | 0 | 4 | −4 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 |
| 0 | 8 | 0 | −8 | 8 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | −4 | 8 | −4 | −4 | 0 | 4 | 0 | 4 | 0 | 4 | 8 | 0 | 4 | 0 | − |
| 0 | 4 | 4 | 0 | −4 | 8 | 0 | 4 | −8 | −4 | 4 | 0 | 4 | 0 | 0 | 4 |
| 0 | 4 | 4 | 0 | −4 | −8 | 0 | 4 | −4 | 0 | 0 | −4 | −8 | 4 | −4 | 0 |
| 0 | −4 | −8 | −4 | −4 | 0 | 4 | 0 | 0 | −4 | 8 | −4 | −4 | 0 | 4 | 0 |

As all Walsh Spectrum's values are 0 except from the $\mathbf{0} \in \mathrm{V}_4$, we can conclude that *NibbleSub* is balanced.

## 5.5 Correlation Immunity

### 5.5.1 Description

In stream cipher applications, it is vital that the Boolean function used as the combining function have certain properties. In addition to being balanced, possessing high nonlinearity and high algebraic degree, the function should have correlation immunity greater than zero to resist a *divide and conquer attack* [Siegenthaler:84].

This criterion describes the extent to which input values of a Vector Boolean function $F \in \mathcal{F}_{n,m}$ can be guessed given the output value. Equivalently, we can say that *F* is *t*-CI if its output distribution does not change when we fix *t* variables $x_i$ of its input.

Interest in this criterion came from discovery by Siegenthaler [Siegenthaler:84] in 1984 of an attack on pseudo-random generators using combining functions (used in stream ciphers), called a correlation attack. This attack is based on the idea of finding correlation between the outputs and the inputs, that is, finding S-boxes with low resiliency.

A function $f \in \mathcal{F}_n$ is $t - CI$ if and only if, for every set S of *t* variables, $1 \leq t \leq n$, given the value of *f*, the probability that S takes on any of its $2^t$ assignments of values to the *t* variables is $\frac{1}{2^t}$. If $f$ is $t - CI$ and balanced, then it is *t*-resilient.

$f \in \mathcal{F}_n$ is said to be *t*-CI if for each linear function $l_{\mathbf{u}} = u_1x_1 + \cdots + u_nx_n$ with $1 \leq wt(\mathbf{u}) \leq t$, $f + l_{\mathbf{u}}$ is balanced [XiaoM:88].

$F \in \mathcal{F}_{n,m}$ is an *t*-CI function (or $(n, m, t)$-CI function) if and only if every component function of $F$ is an *t*-CI function. *F* is said to be *t*-resilient (or $(n, m, t)$-resilient function) if it is balanced and *t*-CI [Chen:02].

Let $f \in \mathcal{F}_n$ and $t \in \{1, \ldots, n-1\}$, $f$ is called correlation immune (CI) of order *t* if its Walsh coefficients, at values of the nonzero vector indexes whose weight at most *t*, are zero: $\hat{\chi}_f(\mathbf{u}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq t$. $f$ can also be denoted as $(n, 1, t)$-CI function.[XiaoM:88]_

Let $F \in \mathcal{F}_{n,m}$ and $t \in \{1, \ldots, n-1\}$, *F* is a correlation immune Vector Boolean function of order *t* if its Walsh coefficients, at values of the nonzero vector indexes whose weight at most *t*, are zero: $\hat{\theta}_F(\mathbf{u}, \mathbf{v}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq t,\ \forall\, \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}$. *F* can also be denoted as an *t*-CI function.

From the definition of resiliency we can derive that a balanced Vector Boolean function can be interpreted as a *0*-resilient function.

### 5.5.2 Library

The method used to obtain this criterion is the following:

```
void CI(int& t, VBF& F)
```

#### Example

The following program provides the order of correlation immunity of a Vector Boolean function given its polynomial in ANF.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;
```

(continues on next page)

(continued from previous page)

```
   VBF          F;
   vec_pol p;
   int t;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> p;
   F.putpol(p);
   input.close();

   t = CI(F);
   cout << "It is a (" << F.n() << "," << F.m()
<< "," << t << ")-CI function" << endl;

   return 0;
}
```

If we use the function $f = (1 + x_1)(1 + x_2)(1 + x_3)(1 + x_4) + x_1x_2x_3x_4$ polynomial in ANF as input, the output would be the following:

```
It is a (4,1,1)-CI function
```

The following figure represents the Walsh Spectrum of $f$ and emphasizes in red the rows whose indexes are of weight 1.

| | | |
|---|---|---|
| 0000 | 16 | 12 |
| 0001 | 0 | 0 |
| 0010 | 0 | 0 |
| 0011 | 0 | −4 |
| 0100 | 0 | 0 |
| 0101 | 0 | −4 |
| 0110 | 0 | −4 |
| 0111 | 0 | 0 |
| 1000 | 0 | 0 |
| 1001 | 0 | −4 |
| 1010 | 0 | −4 |
| 1011 | 0 | 0 |
| 1100 | 0 | −4 |
| 1101 | 0 | 0 |
| 1110 | 0 | 0 |
| 1111 | 0 | −4 |

For all this rows, the Walsh values are *0* so $f$ is 1-CI. There are rows whose indexes are of weight 2 and the Walsh values are not *0* so $f$ cannot be 2-CI.

## 5.6 Algebraic immunity

### 5.6.1 Description

A new kind of attacks, called *algebraic attacks*, has been introduced [Courtois:03], [CourtoisM:02], [FaugereA:03]. Algebraic attacks recover the secret key, or at least the initialization of the system, by solving a system of multivariate algebraic equations. A new criterion was introduced in order to identify a cryptographic algorithm's immunity to this kind of attacks.

Denote the Boolean function obtained by the product of the Truth Tables of two Boolean functions $f, g \in \mathcal{F}_n$ by $f \cdot g$ (Note that this product is different from the dot product between two vectors $\mathbf{x}, \mathbf{y}$). The *algebraic immunity (AI)* of $f$ is defined as the lowest degree of the function $g$ for which $f \cdot g = \mathbf{0}$ or $(\mathbf{1} + f) \cdot g = \mathbf{0}$. The function $g$ for which $f \cdot g = \mathbf{0}$ is called an *annihilator* of $f$. Denote the set of all annihilators of $f$ by $\mathsf{An}(f)$. This set is an ideal in the ring of Boolean functions generated by $\mathbf{1} + f$.

A function $f$ should not be used if $f$ or $1 + f$ has a low degree annihilator. If this happens, algebraic attacks [courtois2002cryptanalysis] can be executed.

The component algebraic immunity of any $F \in \mathcal{F}_{n,m}$, denoted by $AI(F)$, is the minimal algebraic immunity of the component functions $\mathbf{v} \cdot F(\mathbf{v})$ of the Vector Boolean function with $\mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}$.

The *algebraic attack* exploits the existence of multivariate equations involving the input to the S-box and its output, that is, finding S-boxes with low algebraic immunity.

### 5.6.2 Library

The method used to obtain this criterion is the following:

```
void AI(int& ai, VBF& F)
```

The method used to the maximum algebraic immunity that can be achieved by a Vector Boolean function with the same number of input bits and output bits is the following:

```
int aimax(VBF& F)
```

### Example

The following program provides the algebraic immunity of a Vector Boolean function given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
```

(continues on next page)

(continued from previous page)

```
    input >> T;
    F.puttt(T);
    input.close();

    cout << "The algebraic immunity of the function is "
    << AI(F) << endl;
    cout << "The maximum algebraic immunity that can be achieved by
    a Vector Boolean function with the same dimensions is "
    << aimax(F) << endl;

    return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input, the output would be the following:

```
The algebraic immunity of the function is 2
The maximum algebraic immunity that can be achieved by a
Vector Boolean function with the same dimensions is 2
```

## 5.7 Global avalanche criterion

### 5.7.1 Description

The *Global avalanche criterion (GAC)* was introduced in [zhang95gac] to measure the overall avalanche characteristics of a Boolean function. Let $F \in \mathcal{F}_{n,m}$, its Global avalanche criterion is defined by two indicators:

1. The *absolute indicator* of $F$, denoted by $AC_{max}(F)$, defines the maximum absolute non-zero value of the Autocorrelation Spectrum:

$$AC_{max}(F) = \max(|\mathsf{AC}(F)(\mathbf{u}, \mathbf{v})|)\ \forall\, \mathbf{u} \neq \mathbf{0} \in \mathrm{V_n},\ \forall\, \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}$$

2. The *sum-of-square indicator*, denoted by $\sigma$, is the second moment of the autocorrelation coefficients:

$$\sigma(F) = \sum_{(\mathbf{u},\mathbf{v}) \in \mathrm{V_n} \times \mathrm{V_m}} \mathsf{AC}(F)(\mathbf{u}, \mathbf{v})^2 = \frac{1}{2^n} \sum_{(\mathbf{u},\mathbf{v}) \in \mathrm{V_n} \times \mathrm{V_m}} \mathsf{WS}(F)(\mathbf{u}, \mathbf{v})^4$$

In order to achieve good diffusion, cryptographic functions should achieve low values of both indicators.

### 5.7.2 Library

The methods used to obtain these criteria are the following:

```
void maxAC(NTL::ZZ& x, VBF& F)
void sigma(NTL::ZZ& x, VBF& F)
```

#### Example

The following program provides the absolute indicator and the sum-of-square indicator of a Vector Boolean function given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "The absolute indicator of the function is "
   << maxAC(F) << endl;
   cout << "The sum-of-square indicator of the function is "
   << sigma(F) << endl;
   cout << "The maximum absolute indicator that can be achieved by
   a Vector Boolean function with the same dimensions is "
   << maxACmax(F) << endl;
  cout << "The maximum sum-of-square indicator that can be achieved by
   a Vector Boolean function with the same dimensions is "
   << sigmamax(F) << endl;
  cout << "The minimum sum-of-square indicator that can be achieved by
   a Vector Boolean function with the same dimensions is "
   << sigmamin(F) << endl;

   return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input, the output would be the following:

```
The absolute indicator of the function is 16
The sum-of-square indicator of the function is 1408
The maximum absolute indicator that can be achieved by a
Vector Boolean function with the same dimensions is 16
The maximum sum-of-square indicator that can be achieved by a
Vector Boolean function with the same dimensions is 4096
The minimum sum-of-square indicator that can be achieved by a
Vector Boolean function with the same dimensions is 256
```

The following figure represents the Autocorrelation Spectrum of *NibbleSub* and emphasizes in red the values in which the maximum is attained.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 1 |
| 16 | 0 | 0 | 0 | 0 | 0 | −8 | −8 | −8 | −8 | −8 | 8 | 0 | 0 | 8 | 8 |
| 16 | −8 | 0 | −8 | −8 | 0 | 0 | 8 | 8 | −8 | 0 | 0 | −8 | 8 | −8 | 8 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | −16 | −8 | 8 | 0 | 0 | 0 | 0 | −8 | 8 |
| 16 | 0 | −8 | 0 | 0 | −16 | 0 | 8 | 0 | 8 | −8 | −8 | −8 | 0 | 8 | 8 |
| 16 | 0 | 0 | −8 | 0 | 0 | 0 | −8 | 0 | −8 | 8 | −8 | 0 | −8 | 8 | 8 |
| 16 | −8 | 0 | 0 | −8 | 0 | −8 | 8 | 0 | −8 | 0 | 0 | 8 | 0 | −8 | 8 |
| 16 | 0 | −8 | 0 | 0 | 0 | 0 | −8 | 0 | 8 | 0 | 0 | 0 | −8 | −8 | 8 |
| 16 | −8 | −8 | 0 | −8 | 0 | 0 | 8 | −8 | 8 | 0 | 0 | 0 | 0 | 8 | − |
| 16 | 0 | 0 | 8 | 0 | 0 | 0 | −8 | 0 | −8 | 0 | 0 | −8 | 0 | 8 | − |
| 16 | 8 | 0 | 0 | 8 | 0 | 8 | 8 | −8 | −8 | 0 | −8 | 0 | 0 | −8 | −1 |
| 16 | 0 | −8 | −8 | 0 | 16 | −8 | −8 | 8 | 8 | −8 | −8 | 8 | 8 | −8 | − |
| 16 | −8 | 8 | −8 | −8 | 0 | −8 | 8 | 0 | 8 | 0 | 0 | 0 | −8 | 8 | − |
| 16 | 0 | 0 | 0 | 0 | 0 | 8 | −8 | 0 | −16 | 0 | 0 | 0 | 0 | 8 | − |
| 16 | 8 | 0 | 8 | 8 | 0 | 0 | 8 | 0 | −8 | −8 | 0 | 0 | −8 | −16 | − |
| 16 | 0 | 8 | 0 | 0 | −16 | 0 | −8 | 0 | 8 | 8 | 8 | −8 | 0 | −8 | − |

The following figure represents the Autocorrelation Spectrum of *NibbleSub* and emphasizes in blue the columns (component functions) in which the maximum sum-of-square is attained.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 0 | 0 | 0 | 0 | 0 | −8 | −8 | −8 | −8 | −8 | 8 | 0 | 0 | 8 | 8 |
| 16 | −8 | 0 | −8 | −8 | 0 | 0 | 8 | 8 | −8 | 0 | 0 | −8 | 8 | −8 | 8 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | −16 | −8 | 8 | 0 | 0 | 0 | 0 | −8 | 8 |
| 16 | 0 | −8 | 0 | 0 | −16 | 0 | 8 | 0 | 8 | −8 | −8 | −8 | 0 | 8 | 8 |
| 16 | 0 | 0 | −8 | 0 | 0 | 0 | −8 | 0 | −8 | 8 | −8 | 0 | −8 | 8 | 8 |
| 16 | −8 | 0 | 0 | −8 | 0 | −8 | 8 | 0 | −8 | 0 | 0 | 8 | 0 | −8 | 8 |
| 16 | 0 | −8 | 0 | 0 | 0 | 0 | −8 | 0 | 8 | 0 | 0 | 0 | −8 | −8 | 8 |
| 16 | −8 | −8 | 0 | −8 | 0 | 0 | 8 | −8 | 8 | 0 | 0 | 0 | 0 | 8 | −8 |
| 16 | 0 | 0 | 8 | 0 | 0 | 0 | −8 | 0 | −8 | 0 | 0 | −8 | 0 | 8 | −8 |
| 16 | 8 | 0 | 0 | 8 | 0 | 8 | 8 | −8 | −8 | 0 | −8 | 0 | 0 | −8 | −1 |
| 16 | 0 | −8 | −8 | 0 | 16 | −8 | −8 | 8 | 8 | −8 | −8 | 8 | 8 | −8 | −8 |
| 16 | −8 | 8 | −8 | −8 | 0 | −8 | 8 | 0 | 8 | 0 | 0 | 0 | −8 | 8 | −8 |
| 16 | 0 | 0 | 0 | 0 | 0 | 8 | −8 | 0 | −16 | 0 | 0 | 0 | 0 | 8 | −8 |
| 16 | 8 | 0 | 8 | 8 | 0 | 0 | 8 | 0 | −8 | −8 | 0 | 0 | −8 | −16 | −8 |
| 16 | 0 | 8 | 0 | 0 | −16 | 0 | −8 | 0 | 8 | 8 | 8 | −8 | 0 | −8 | −8 |
| — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| | 640 | 640 | 640 | 640 | 1024 | 640 | 1408 | 640 | 1408 | 640 | 640 | 640 | 640 | 1408 | 140 |

## 5.8 Linearity distance

### 5.8.1 Description

Functions with non-zero linear structures are considered weak functions from cryptanalytic viewpoint. It is our interest to identify strong Vector Boolean functions which are far from this weak functions. The cryptanalytic value of linear structures lies in their potential to map a nonlinear function to a degenerate function via a linear transformation, which may reduce the size of the keyspace.

S-boxes used in block ciphers should have no nonzero linear structures (see [Evertse:88]). The existence of nonzero linear structures, for the functions implemented in stream ciphers, is a potential risk that should also be avoided, despite the fact that such existence could not be used in attacks, so far.

The *linearity distance* of a Boolean function $f \in \mathcal{F}_n$ is a characteristic defined by the distance to the set of all Boolean functions admitting nonzero linear structures. These include, among others, all the affine functions and all non bent quadratic functions and are defined as follows [MeierS:89]:

$$LD(f) = d(f, \mathsf{LS}_n) = \min_{\mathsf{S} \in \mathsf{LS}_n} d(f, \mathsf{S})$$

where:

$$\mathsf{LS}_n = \{f \in \mathcal{F}_n \mid f \text{ has a linear structure } \neq \mathbf{0}\}$$

*Linearity distance* of a Vector Boolean function, defined as the minimum among the linearity distances of all component functions of *F*, may be computed from the Autocorrelation Spectrum using [CarletBF:08]:

$$LD(F) = \min_{\mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}} LD(\mathbf{v} \cdot F) = 2^{n-2} - \frac{1}{4} \cdot AC_{max}(F)$$

The *differential cryptanalysis* is based on the idea of finding high probable differentials pairs between the inputs and outputs of S-boxes present in the cipher, that is, finding S-boxes with low linearity distance. Differential cryptanalysis [BihamS:90] can be seen as an extension of the ideas of attacks based on the presence of linear structures [Nyberg:91]. If $\mathbf{u}$ is a linear structure of *f*, then the inputs of difference $\mathbf{u}$ result in output differences of *1* or *-1* with probability *1*. In differential cryptanalysis, it is only required that inputs of difference $\Delta\mathbf{x}$ lead to a known difference $\Delta\mathbf{y}$ with high probability, or with a probability that noticeably exceeds the mean. The perfect nonlinear functions are resistant to differential cryptanalysis.

Let $F \in \mathcal{F}_{n,m}$, if $LD(F) = 0$, it means that *f* has a nontrivial linear structure. As $\mathsf{A}_n \subseteq \mathsf{LS}_n$, then $NL(F) \geq LD(F)$.

### 5.8.2 Library

The method used to obtain the linearity distance of a Vector Boolean function is the following:

```
void ld(NTL::RR& x, VBF& F)
```

The method used to the maximum linearity distance that can be achieved by a Vector Boolean function with the same number of input bits and output bits is the following:

```
NTL::RR ldmax(VBF& F)
```

### Example

The following program provides the linearity distance of a Vector Boolean function given its Truth Table together with the maximum linearity distance that can be achieved by a Vector Boolean function with the same number of input bits and output bits.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   NTL::mat_GF2 T;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> T;
   F.puttt(T);
   input.close();

   cout << "Linearity distance of the function is " << ld(F) << endl;
   cout << "The maximum linearity distance: " << ldmax(F) << endl;

   return 0;
}
```

If we use the *NibbleSub* S-box Truth Table as input, the output would be the following:

```
Linearity distance of the function is 0
```

This S-box has linear structures, and as a consequence, the distance to the set of all Boolean functions admitting nonzero linear structures is 0.

## 5.9 Propagation criterion

### 5.9.1 Description

This criterion is based on the properties of the derivatives of Boolean functions and describes the behavior of a function whenever some input bits are complemented. This concept was introduced by Preneel et al. in [PreneelLLGV90] and it is a generalization of the Strict Avalanche Criterion (SAC) defined by Webster and Tavares in [c85-Webster-Tavares].

$f \in \mathcal{F}_n$ is said to satisfy the propagation characteristics with respect to $\mathbf{u} \in \mathrm{V_n}$ if and only if $f(\mathbf{x}) + f(\mathbf{x} + \mathbf{u})$ is balanced.

A function $f \in \mathcal{F}_n$ satisfies the propagation criterion of degree $l$ $(PC(l))$ if and only if complementing any $l$ or fewer of the input bits complements exactly half of the function values.

Let $f \in \mathcal{F}_n$ and $l \in \{1, \ldots, n\}$, $f$ satisfies the propagation criterion of degree $l$ if and only if $f(\mathbf{x}) + f(\mathbf{x} + \mathbf{u})$ balanced $\forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq l$.

Let $f \in \mathcal{F}_n$ and $l \in \{1, \ldots, n\}$, $f$ satisfies the propagation criterion of degree $l$ if its Autocorrelation Matrix elements, at values of the nonzero vector indexes whose weight at most $l$, is zero: $r_f(\mathbf{u}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq l$

$F \in \mathcal{F}_{n,m}$ satisfies the propagation criterion of degree $l$ $(PC(l))$ if any component function of $F$ satisfies the $PC(l)$. This criterion can be obtained from the Autocorrelation Spectrum in the following way: $r_F(\mathbf{u}, \mathbf{v}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq l,\ \forall\, \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}$

### 5.9.2 Library

The method used to obtain this criterion is the following:

```
void PC(int& k, VBF& F)
```

### Example

The following program provides the degree of propagation criterion of a Vector Boolean function given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F;
   vec_pol p;
   int t;

   ifstream input(argv[1]);
   if(!input) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input >> p;
   F.putpol(p);
   input.close();

   cout << "The function is PC of degree " << PC(F) << endl;

   return 0;
}
```

If we use the function $f = x_1x_2 + x_3x_4$ polynomial in ANF as input, the output would be the following:

```
The function is PC of degree 4
```

The following figure represents the Autocorrelation Spectrum of $f$ and emphasizes in red the rows whose indexes are of weight 1,2,3 and 4.

| | | |
|---|---|---|
| 0000 | 16 | 16 |
| 0001 | 16 | 0 |
| 0010 | 16 | 0 |
| 0011 | 16 | 0 |
| 0100 | 16 | 0 |
| 0101 | 16 | 0 |
| 0110 | 16 | 0 |
| 0111 | 16 | 0 |
| 1000 | 16 | 0 |
| 1001 | 16 | 0 |
| 1010 | 16 | 0 |
| 1011 | 16 | 0 |
| 1100 | 16 | 0 |
| 1101 | 16 | 0 |
| 1110 | 16 | 0 |
| 1111 | 16 | 0 |

For all this rows, the Autocorrelation values are 0. As a consequence $f$ satisfies $PC(4)$.

## 5.10 Summary

Cryptographic criteria Table lists the member functions related to these criteria.

| Cryptographic criteria | |
|---|---|
| **SYNTAX** | **DESCRIPTION** |
| `void deg(int& d, VBF& F)` | $DEG(F) = d$ |
| `void nl(NTL::RR& x, VBF& F)` | $NL(F) = x$ |
| `void nlr(long& x, VBF& F, int r)` | $NL_r(F) = x$ |
| `void Bal(int& bal, VBF& F)` | If *F* is balanced returns *1*, otherwise *0* |
| `void CI(int& t, VBF& F)` | *F* is an $(n, m, t) - CI$ |
| `void AI(int& i, VBF& F)` | $AI(F) = i$ |
| `void maxAC(NTL::ZZ& x, VBF& F)` | *F* has absolute indicator *x* |
| `void sigma(NTL::ZZ& x, VBF& F)` | *F* has sum-of-squares indicator *x* |
| `void ld(NTL::RR& x, VBF& F)` | $LD(F) = x$ |
| `void PC(int& l, VBF& F)` | *F* satisfies the $PC(l)$ |

Member functions Table lists the member functions related to bounds and other properties of above criteria.

| Member functions of the cryptographic criteria | |
|---|---|
| **SYNTAX** | **DESCRIPTION** |
| `void SpectralRadius(NTL::ZZ& x, VBF& F)` | Spectral Radius |
| `NTL::RR nlmax(VBF& F)` | Maximum possible nonlinearity |
| `void typenl(int& typenl, VBF& F)` | 1=Bent, 2=Almost Bent, 3=Linear |
| `int aimax(VBF& F)` | Maximum possible algebraic immunity |
| `NTL::ZZ maxACmax(VBF& F)` | Maximum possible absolute indicator |
| `NTL::ZZ maxsigma(VBF& F)` | Maximum possible sum-of-square indicator |
| `NTL::ZZ minsigma(VBF& F)` | Minimum possible sum-of-square indicator |
| `NTL::RR ldmax(VBF& F)` | Maximum possible linearity distance |

# 6 Operations and constructions over Vector Boolean Functions

In this chapter, some basic constructions for Vector Boolean functions supported by the VBF class are described. Some of them correspond to secondary constructions, which build $(n, m)$ variable vector Boolean functions from $(n', m')$ variable ones (with $n' \leq n, m' \leq m$). The direct sum has been used to construct resilient and bent Boolean functions [Carlet:04]. The concatenation can be used to obtain resilient functions or functions with maximal nonlinearity. The concatenation of polynomials in ANF can be used to obtain functions of high nonlinearity with $n$ variables from functions with high nonlinearity with $n'$ variables $(n' < n)$. Adding coordinate functions and bricklayering are constructions used to build modern ciphers such as CAST [CAST:256], DES [DES:77] and AES [DaemenR:02]. Additionally, VBF provides operations for identification if two vector Boolean functions are equal, the sum of two vector Boolean functions, the composition of two vector Boolean functions and the inverse of a Vector Boolean function.

## 6.1 Equality testing

### 6.1.1 Description

Let $n \geq 1, m \geq 1$, $F, G \in \mathcal{F}_{n,m}$. $F$ and $G$ are *equal* if their Truth Tables are the same.

### 6.1.2 Library

We can compare two functions for equality with the following method:

```
long operator==(VBF& F, VBF& G)
long operator!=(VBF& F, VBF& G)
```

#### Example

The following program informs if two Vector Boolean functions are equal given their Truth Tables.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, X;
   NTL::mat_GF2 Tf, Tg;

   ifstream input1(argv[1]);
   if(!input1) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input1 >> Tf;
```

(continues on next page)

(continued from previous page)

```
   F.puttt(Tf);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   input2 >> Tg;
   G.puttt(Tg);
   input2.close();

   if (F == G) {
     cout << "F and G are equal" << endl;
   }  else {
     cout << "F and G are not equal" << endl;
   }

   return 0;
}
```

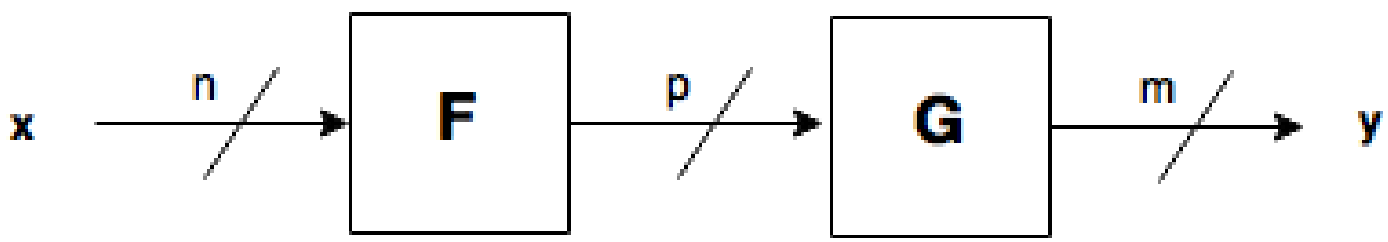
The output for the execution of the example program with the code above and the Truth Tables of $S_1$ and $S_2$ DES S-boxes as inputs would be:

```
F and G are not equal
```

## 6.2 Composition

### 6.2.1 Description

Let $F \in \mathcal{F}_{n,p}$, $G \in \mathcal{F}_{p,m}$ and the composition function $G \circ F \in \mathcal{F}_{n,m}$ where $G \circ F(\mathbf{x}) = G(F(\mathbf{x}))\ \forall\, \mathbf{x} \in \mathrm{V_n}$.




### 6.2.2 Library

It can be obtained with the following method:

```
void Comp(VBF& X, VBF& F, VBF& G)
```

#### Example 1

The following program provides the correlation immunity and balancedness of two Vector Boolean functions given their Truth Tables and calculates the same criteria for their composition.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, X;
   NTL::mat_GF2 Tf,Tg;

   ifstream input1(argv[1]);
   if(!input1)  {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input1 >> Tf;
   F.puttt(Tf);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   input2 >> Tg;
   G.puttt(Tg);
   input2.close();

   cout << "Correlation immunity of F: " << CI(F) << endl;
   if (Bal(F)) {
     cout << "F is a balanced function" << endl;
   } else {
     cout << "F is a non-balanced function" << endl;
   }
   cout << endl;

   cout << "Correlation immunity of G: " << CI(G) << endl;
   if (Bal(G)) {
     cout << "G is a balanced function" << endl;
   } else {
     cout << "G is a non-balanced function" << endl;
   }
   cout << endl;

   Comp(X,F,G);

   cout << "Correlation immunity of GoF: " << CI(X) << endl;
   if (Bal(X)) {
     cout << "GoF is a balanced function" << endl;
   } else {
     cout << "GoF is a non-balanced function" << endl;
   }

   return 0;
}
```

If we use $\mathbf{y_0}$ of CLEFIA $S_0$ cipher (see CLEFIA section in "Analysis of CRYPTEC project cryptographic algorithms")

and *NibbleSub* Truth Tables as inputs, the output would be the following:

```
Correlation immunity of F: 1
F is a balanced function

Correlation immunity of G: 0
G is a balanced function

Correlation immunity of GoF: 1
GoF is a balanced function
```

### Example 2

The following program provides the balancedness of two Vector Boolean functions given its polynomial representation in ANF and calculates the balancedness for the its composition.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, X;
   vec_pol f,g;

   ifstream input1(argv[1]);
   if(!input1)  {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input1 >> f;
   F.putpol(f);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   input2 >> g;
   G.putpol(g);
   input2.close();

   cout << "The polynomial in ANF of F is ";
   cout << endl;
   Pol(cout,F);

   if (Bal(F)) {
     cout << "F is a balanced function" << endl;
   } else {
     cout << "F is a non-balanced function" << endl;
   }
   cout << endl;
```

(continues on next page)

(continued from previous page)

```
    cout << "The polynomial in ANF of G is ";
    cout << endl;
    Pol(cout,G);

    if (Bal(G)) {
      cout << "G is a balanced function" << endl;
    } else {
      cout << "G is a non-balanced function" << endl;
    }
    cout << endl;

    Comp(X,F,G);
    cout << "The polynomial in ANF of the composition of F and G is ";
    cout << endl;
    Pol(cout,X);

    if (Bal(X)) {
      cout << "GoF is a balanced function" << endl;
    } else {
      cout << "GoF is a non-balanced function" << endl;
    }

    return 0;
}
```

If we use the Boolean functions of first example described in [GuptaS:05] as inputs, the output would be the following:

```
The polynomial in ANF of F is
x1+x2+x1x3+x1x2x3
x2+x1x2+x2x3+x1x3+x1x2x3
F is a non-balanced function

The polynomial in ANF of G is
x1+x2
G is a balanced function

The polynomial in ANF of the composition of F and G is
x2x3+x1+x1x2
GoF is a balanced function
```

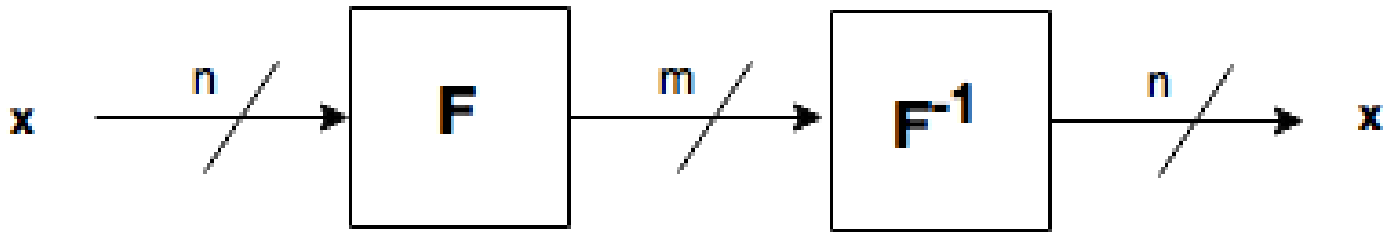
If we use the Boolean functions of second example described in [GuptaS:05] as inputs, the output would be the following:

```
The polynomial in ANF of F is
x3+x1x2+x1x2x3
x2+x3+x1x2+x2x3+x1x2x3
F is a non-balanced function

The polynomial in ANF of G is
x1x2
G is a non-balanced function

The polynomial in ANF of the composition of F and G is
x3
GoF is a balanced function
```

## 6.3 Inverse

### 6.3.1 Description

Let $n \geq 1$, $F \in \mathcal{F}_{n,n}$. $F^{-1}$ is the *functional inverse* of $F$ if the composition of both functions results in the identity function.




### 6.3.2 Library

If a Vector Boolean Function $F in funct{F}_{n,n}$ is invertible, then we can find its inverse with the following method:

```
void inv(VBF& X, VBF& F)
```

### Example

The following program provides the Truth Table of a the inverse of a Vector Boolean function given its Truth Table.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF           F, X;
   NTL::mat_GF2 Tf;

   ifstream input1(argv[1]);
   if(!input1) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input1 >> Tf;
   F.puttt(Tf);
   input1.close();

   inv(X,F);
   cout << "The Truth Table of the inverse of F is " << endl
   << TT(X) << endl;

   return 0;
}
```

The output for the execution of the example program with the code above and the Truth Table of *NibbleSub* S-box as input will be:

```
The Truth Table of the inverse of F is
[[1 1 1 0]
[0 0 1 1]
[0 1 0 0]
[1 0 0 0]
[0 0 0 1]
[1 1 0 0]
[1 0 1 0]
[1 1 1 1]
[0 1 1 1]
[1 1 0 1]
[1 0 0 1]
[0 1 1 0]
[1 0 1 1]
[0 0 1 0]
[0 0 0 0]
[0 1 0 1]
]
```

## 6.4 Sum

### 6.4.1 Description

Let $n \geq 1, m \geq 1$, $F, G \in \mathcal{F}_{n,m}$. The *Sum* of $F$ and $G$, denoted by $F + G \in \mathcal{F}_{n,m}$ is the Vector Boolean Function whose Truth Table results from the addition of the Truth Tables of $F$ and $G$: $\mathsf{T}_{F+G} = \mathsf{T}_F + \mathsf{T}_G$.

### 6.4.2 Library

It can be obtained with the following method:

```
void sum(VBF& X, VBF& F, VBF& G)
```

### Example

The following program provides the nonlinearity, absolute indicator and linearity distance of two Vector Boolean functions given its polynomial representation in ANF and its hexadecimal representation of Truth Table respectively and calculates the same criteria for the its sum.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, X;
   vec_pol      f;

   ifstream input1(argv[1]);
   if(!input1)   {
      cerr << "Error opening " << argv[1] << endl;
```

(continues on next page)

(continued from previous page)

```
      return 0;
   }
   input1 >> f;
   F.putpol(f);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2)  {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   G.putHexTT(input2);
   input2.close();

   cout << "The polynomial in ANF of F is ";
   cout << endl;
   Pol(cout,F);

   cout << "nl(F)=" << nl(F) << endl;
   cout << "ACmax(F)=" << maxAC(F) << endl;
   cout << "LD(F)=" << ld(F) << endl;
   cout << endl;

   cout << "The polynomial in ANF of G is ";
   cout << endl;
   Pol(cout,G);
   cout << endl;

   sum(X,F,G);
   cout << "The polynomial in ANF of the sum of F and G is ";
   cout << endl;
   Pol(cout,X);

   cout << "nl(F+G)=" << nl(X) << endl;
   cout << "ACmax(F+G)=" << maxAC(X) << endl;
   cout << "LD(F+G)=" << ld(X) << endl;
   cout << endl;

   return 0;
}
```

If we use the Boolean function $F$ with ANF $x_1x_2 + x_3x_4$ and function $G$ with hexadecimal representation of Truth Table *0001* as inputs, the output would be the following:

```
The polynomial in ANF of F is
x1x2+x3x4
nl(F)=6
ACmax(F)=0
LD(F)=4

The polynomial in ANF of G is
x1x2x3x4

The polynomial in ANF of the sum of F and G is
x3x4+x1x2+x1x2x3x4
nl(F+G)=5
```

(continues on next page)

(continued from previous page)

```
ACmax(F+G)=4
LD(F+G)=3
```

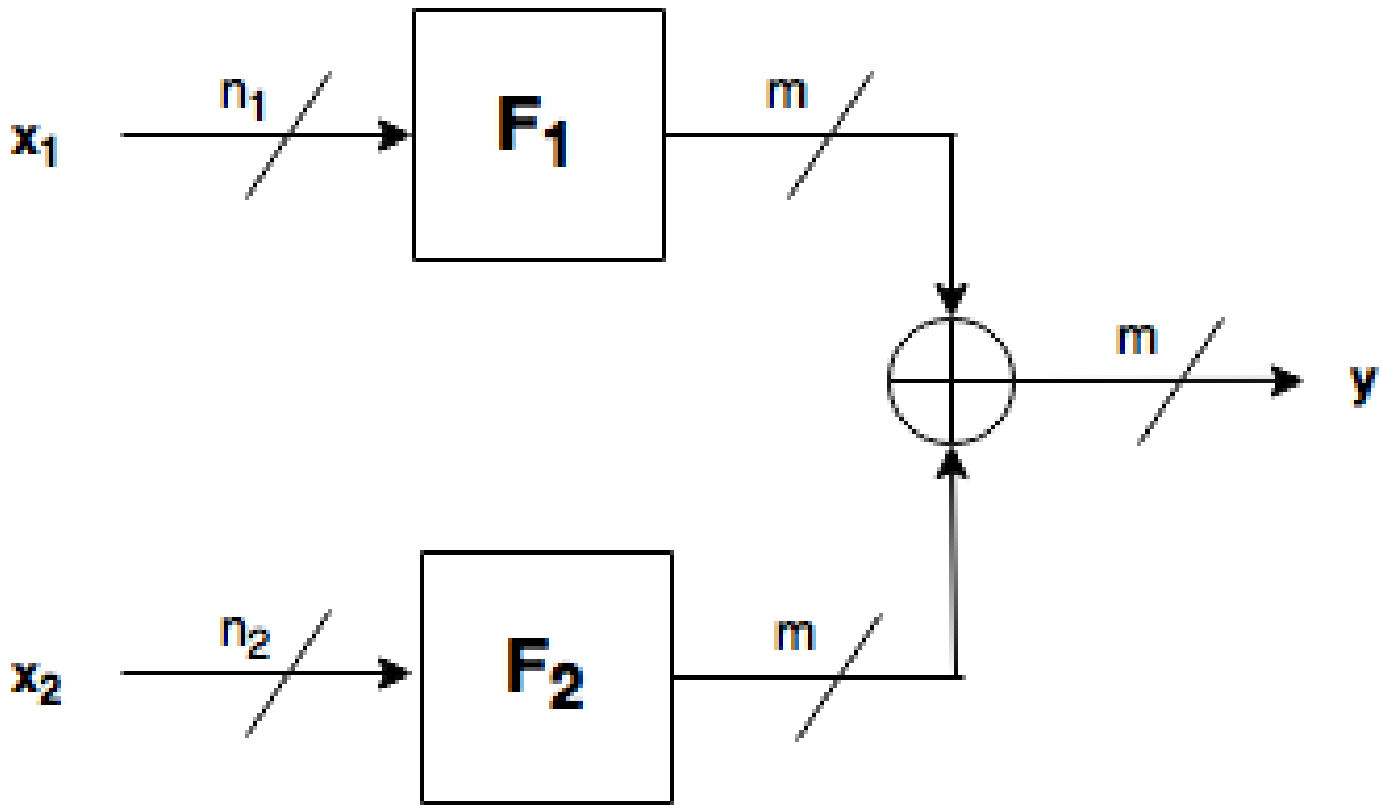
These results are congruent with the properties of changing one bit of the Truth Table:

1. $NL(F+G) = NL(F) - 1 = 6 - 1 = 5$.
2. $AC_{max}(F+G) = AC_{max}(F) + 4 = 0 + 4 = 4$.
3. $LD(F+G) = LD(F) - 1 = 4 - 1 = 3$.

## 6.5 Direct sum

### 6.5.1 Description

Let $n_1, n_2 \geq 1$, $F_1 \in \mathcal{F}_{n_1,m}, F_2 \in \mathcal{F}_{n_2,m}$ be Vector Boolean functions. Consider the Vector Boolean function $F_1 \oplus F_2 \in \mathcal{F}_{n_1+n_2,m}$, called direct sum, defined as $(F_1 \oplus F_2)((\mathbf{x_1}, \mathbf{x_2})) = F_1(\mathbf{x_1}) + F_2(\mathbf{x_2})$.




### 6.5.2 Library

The method included in VBF to perform this construction is the following:

```
void directsum(VBF& X, VBF& F, VBF& G)
```

### Example

The following program provides the weight, algebraic degree, balancedness, correlation immunity, nonlinearity and algebraic immunity of two Vector Boolean functions given its polynomial representation in ANF and calculates the same criteria for the its direct sum.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, X;

   ifstream input1(argv[1]);
   if(!input1){
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   F.putHexTT(input1);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   G.putHexTT(input2);
   input2.close();

   cout << "weight(F)=" << weight(F) << endl;
   cout << "deg(F)=" << deg(F) << endl;
   if (Bal(F)) {
     cout << "F is a balanced function" << endl;
   } else {
     cout << "F is a non-balanced function" << endl;
   }
   cout << "Degree of Correlation immunity of F=" << CI(F) << endl;
   cout << "R(F)=" << SpectralRadius(F) << endl;
   cout << "nl(F)=" << nl(F) << endl;
   cout << "ACmax(F)=" << maxAC(F) << endl;
   cout << "ld(F)=" << ld(F) << endl;
   cout << "AI(F)=" << AI(F) << endl;
   cout << "F is PC of degree " << PC(F) << endl;
   cout << endl;

   cout << "weight(G)=" << weight(G) << endl;
   cout << "deg(G)=" << deg(G) << endl;
   if (Bal(G)) {
     cout << "G is a balanced function" << endl;
   } else {
     cout << "G is a non-balanced function" << endl;
   }
   cout << "Degree of Correlation immunity of G=" << CI(G) << endl;
   cout << "R(G)=" << SpectralRadius(G) << endl;
```

(continues on next page)

(continued from previous page)

```
   cout << "nl(G)=" << nl(G) << endl;
   cout << "ACmax(G)=" << maxAC(G) << endl;
   cout << "ld(G)=" << ld(G) << endl;
   cout << "AI(G)=" << AI(G) << endl;
   cout << "G is PC of degree " << PC(G) << endl;
   cout << endl;

   directsum(X,F,G);

   cout << "weight(F directsum G)=" << weight(X) << endl;
   cout << "deg(F directsum G)=" << deg(X) << endl;
   if (Bal(X)) {
     cout << "F directsum G is a balanced function" << endl;
   } else {
     cout << "F directsum G is a non-balanced function" << endl;
   }
   cout << "Degree of Correlation immunity of F directsum G=" << CI(X) << endl;
   cout << "R(F directsum G)=" << SpectralRadius(X) << endl;
   cout << "nl(F directsum G)=" << nl(X) << endl;
   cout << "ACmax(F directsum G)=" << maxAC(X) << endl;
   cout << "ld(F directsum G)=" << ld(G) << endl;
   cout << "AI(F directsum G)=" << AI(X) << endl;
   cout << "F directsum G is PC of degree " << PC(X) << endl;

   return 0;
}
```

If we use the Boolean functions with the following Truth Tables (in hexadecimal representation) as inputs:

```
6cb405778ea9bd30
```

```
5c721bcaac27b1c5
```

The output would be the following:

```
weight(F)=32
deg(F)=3
F is a balanced function
Degree of Correlation immunity of F=1
R(F)=16
nl(F)=24
ACmax(F)=32
ld(F)=8
AI(F)=3
F is PC of degree 2

weight(G)=32
deg(G)=3
G is a balanced function
Degree of Correlation immunity of G=2
R(G)=32
nl(G)=16
ACmax(G)=64
ld(G)=0
AI(G)=2
G is PC of degree 1
```

(continues on next page)

(continued from previous page)

```
weight(F directsum G)=2048
deg(F directsum G)=3
F directsum G is a balanced function
Degree of Correlation immunity of F directsum G=4
R(F directsum G)=512
nl(F directsum G)=1792
ACmax(F directsum G)=4096
ld(F directsum G)=0
AI(F directsum G)=3
F directsum G is PC of degree 1
```

These results are congruent with the properties derived in [SarkarMaitra:00] and others derived by Jose Antonio Alvarez:

1. $wt(F \oplus G) = 2^6 \cdot 32 + 2^6 \cdot 32 - 2 \cdot 32 \cdot 32 = 2048$.
2. $deg(F \oplus G) = \max\{3, 3\} = 3$.
3. $F$ is *1*-resilient, $G$ is *2*-resilient, and $F \oplus G$ is $(1 + 2 + 1)$-resilient.
4. $R(F \oplus G) = 16 \cdot 32 = 512$ because $F$ and $G$ are Boolean functions.
5. $NL(F \oplus G) = 2^{12-1} - \frac{1}{2} \cdot 512 = 1792$.
6. $AC_{max}(F \oplus G) = \max\{32 \cdot 64, 64 \cdot 64\} = 4096$.
7. $LD(F \oplus G) = 2^{12-2} - \frac{1}{4} \cdot 4096 = 0$.
8. $\max\{3, 2\} \leq AI(F \oplus G) = 3 \leq \min\{\max\{3, 3\}, 3 + 2\}$.

## 6.6 Concatenation

### 6.6.1 Description

Let $n_1, n_2 \geq 1$, $F_1 \in \mathcal{F}_{n,m}$, $F_2 \in \mathcal{F}_{n,m}$ be Vector Boolean functions. Consider the Vector Boolean function $F_1|_c F_2 \in \mathcal{F}_{n+1,m}$ defined as $(\mathbf{x}, x_{n+1}) \rightarrow (x_{n+1} + 1)\, F_1(\mathbf{x}) + x_{n+1} F_2(\mathbf{x})$ where $\mathbf{x} \in \mathrm{V_n}$.

### 6.6.2 Library

The method included in VBF to perform this construction is the following:

```
void concat(VBF& X, VBF& F, VBF& G)
```

#### Example

The following program provides the weight, algebraic degree, balancedness, correlation immunity, nonlinearity and algebraic immunity of two Vector Boolean functions given its polynomial representation in ANF and calculates the same criteria for its concatenation.

```
#include <iostream>
#include <fstream>
#include "VBF.h"
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, X;
   vec_pol f,g;

   ifstream input1(argv[1]);
   if(!input1) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   input1 >> f;
   F.putpol(f);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   input2 >> g;
   G.putpol(g);
   input2.close();

   cout << "weight(F)=" << weight(F) << endl;
   cout << "deg(F)=" << deg(F) << endl;
   if (Bal(F))  {
     cout << "F is a balanced function" << endl;
   } else {
     cout << "F is a non-balanced function" << endl;
   }
   cout << "Degree of Correlation immunity of F=" << CI(F) << endl;
   cout << "nl(F)=" << nl(F) << endl;
   cout << "AI(F)=" << AI(F) << endl;
   cout << endl;

   cout << "weight(G)=" << weight(G) << endl;
   cout << "deg(G)=" << deg(G) << endl;
   if (Bal(G)) {
     cout << "G is a balanced function" << endl;
   } else {
     cout << "G is a non-balanced function" << endl;
   }
   cout << "Degree of Correlation immunity of G=" << CI(G) << endl;
   cout << "nl(G)=" << nl(G) << endl;
   cout << "AI(G)=" << AI(G) << endl;
   cout << endl;

   concat(X,F,G);
   cout << "The polynomial in ANF of the concatenation of F and G is ";
   cout << endl;
   Pol(cout,X);

   cout << "weight(F concat G)=" << weight(X) << endl;
   cout << "deg(F concat G)=" << deg(X) << endl;
   if (Bal(X)) {
```

(continues on next page)

(continued from previous page)

```
      cout << "F concat G is a balanced function" << endl;
    } else {
      cout << "F concat G is a non-balanced function" << endl;
    }
    cout << "Degree of Correlation immunity of F concat G="
    << CI(X) << endl;
    cout << "nl(F concat G)=" << nl(X) << endl;
    cout << "AI(F concat G)=" << AI(X) << endl;

    return 0;
}
```

If we use the Boolean functions $1 + x_3x_4 + x_2 + x_2x_4 + x_1 + x_1x_3 + x_1x_3x_4$ and $x_3 + x_2x_4 + x_1 + x_1x_4 + x_1x_3x_4$ as inputs, the output would be the following:

```
weight(F)=8
deg(F)=3
F is a balanced function
Degree of Correlation immunity of F=0
nl(F)=4
AI(F)=2

weight(G)=8
deg(G)=3
G is a balanced function
Degree of Correlation immunity of G=0
nl(G)=4
AI(G)=2

The polynomial in ANF of the concatenation of F and G is
1+x4x5+x3+x3x5+x2+x2x4+x2x4x5
weight(F concat G)=16
deg(F concat G)=3
F concat G is a balanced function
Degree of Correlation immunity of F concat G=0
nl(F concat G)=8
AI(F concat G)=2
```

These results are congruent with the properties of this construction:

1. $wt(F|_cG) = 8 + 8 = 16$.
2. $deg(F|_cG) = 3 \leq 1 + \max\{3, 3\} = 1 + 3 = 4$.
3. $F$ is *0*-resilient, $G$ is *0*-resilient, and $F|_cG$ is *0*-resilient.
4. $NL(F|_cG) = 8 \geq 4 + 4 = 8$.
5. If $AI(F) = AI(G) = 2$, then $AI(F|_cG) = 2 \leq 2 + 1$.

## 6.7 Concatenation of polynomials in ANF

### 6.7.1 Description

Let $n_1, n_2 \geq 1$, $F_1 \in \mathcal{F}_{n_1,m}, F_2 \in \mathcal{F}_{n_2,m}$ be Vector Boolean functions. Consider the Vector Boolean function $F_1|_pF_2 \in \mathcal{F}_{n_1+n_2,m}$ defined as $(x_1, \ldots, x_{n_1}, x_{n_1+1}, \ldots, x_{n_1+n_2}) \rightarrow F_1(x_1, \ldots, x_{n_1}) + F_2(x_{n_1+1}, \ldots, x_{n_1+n_2})$ where $\mathbf{x} \in \mathrm{V}_{\mathrm{n}_1+\mathrm{n}_2}$.

### 6.7.2 Library

The method included in VBF to perform this construction is the following:

```
void concatpol(VBF& X, VBF& F, VBF& G)
```

### Example

The following program provides the ANF of the concatenation of polynomials in ANF of two Vector Boolean functions given its polynomial representation.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F,G,H;
   vec_pol    f,g;
   NTL::mat_GF2 T;

   ifstream inputf(argv[1]);
   if(!inputf) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   inputf >> f;
   F.putpol(f);
   inputf.close();

   ifstream inputg(argv[2]);
   if(!inputg) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   inputg >> g;
   G.putpol(g);
   inputg.close();

   concatpol(H,F,G);
   cout << "The ANF of the concatenation of polynomials
   in ANF of F and G is ";
   cout << endl;
   Pol(cout,H);

   return 0;
}
```

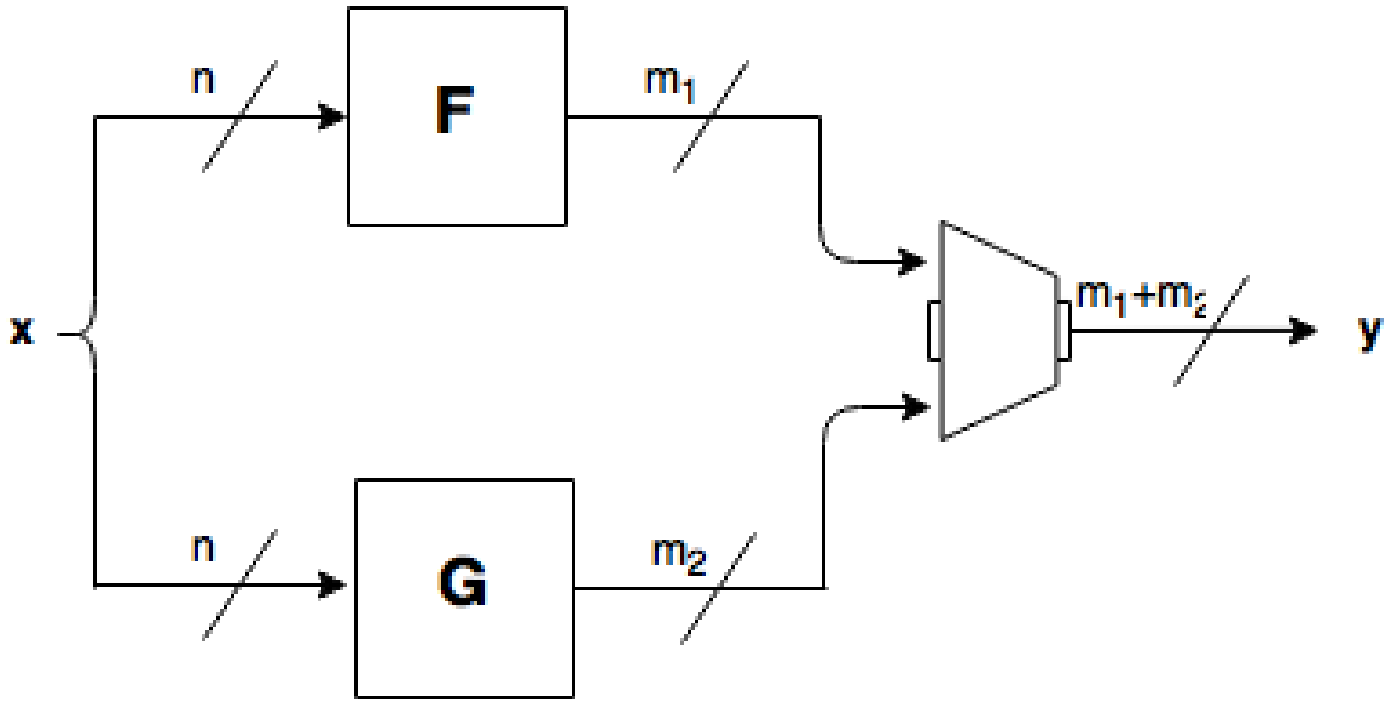
If we use the Boolean functions $x_1x_2 + x_3x_4$ and $x_1 + 1$ as inputs, the output would be the following:

```
The ANF of the concatenation of polynomials in ANF of F and G is
x1x2+x3x4+x5+1
```

## 6.8 Addition of coordinate functions

### 6.8.1 Description

Let $F = (f_1, \ldots, f_{m_1}) \in \mathcal{F}_{n,m_1}$, $G = (g_1, \ldots, g_{m_2}) \in \mathcal{F}_{n,m_2}$ and the function conformed by adding the coordinate functions $(F, G) = (f_1, \ldots, f_{m_1}, g_1, \ldots, g_{m_2}) \in \mathcal{F}_{n,m_1+m_2}$. Let $\mathbf{v} \in \mathrm{V}_{\mathrm{m_1+m_2}}$, $\mathbf{v_F} \in \mathrm{V}_{\mathrm{m_1}}$ and $\mathbf{v_G} \in \mathrm{V}_{\mathrm{m_2}}$ so that $\mathbf{v} = (\mathbf{v_F}, \mathbf{v_G})$.




### 6.8.2 Library

This construction can be obtained with the following method:

```
void addimage(VBF& X, VBF& F, VBF& G)
```

### Example

The following program provides the Truth Tables of the different intermediate constructions that allow to obtain CLEFIA $S_0$ $8 \times 8$ S-box from the Truth Tables of the four 4-bit S-boxes $SS_0, SS_1, SS_2$ and $SS_3$ in which it is constructed and the Truth Table of the multiplication operation in *0x2* performed in $\mathrm{GF}(2^4)$ defined by the primitive polynomial $x^4 + x + 1$.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;
```

(continues on next page)

(continued from previous page)

```
VBF F,G,T20,T21,U0,U1,Y0,Y1,Y;
NTL::mat_GF2 TSS0, TSS1, TSS2, TSS3, Tmul2;
NTL::mat_GF2 T2t0, T2t1, Tu0, Tu1, Ty0, Ty1, Ty;

ifstream inputSS0("SS0.tt");
if(!inputSS0) {
   cerr << "Error opening " << "SS0.tt" << endl;
   return 0;
}
inputSS0 >> TSS0;
inputSS0.close();

ifstream inputSS1("SS1.tt");
if(!inputSS1) {
   cerr << "Error opening " << "SS1.tt" << endl;
   return 0;
}
inputSS1 >> TSS1;
inputSS1.close();

ifstream inputSS2("SS2.tt");
if(!inputSS2) {
   cerr << "Error opening " << "SS2.tt" << endl;
   return 0;
}
inputSS2 >> TSS2;
inputSS2.close();

ifstream inputSS3("SS3.tt");
if(!inputSS3) {
   cerr << "Error opening " << "SS3.tt" << endl;
   return 0;
}
inputSS3 >> TSS3;
inputSS3.close();

ifstream inputmul2("Mul2.tt");
if(!inputmul2) {
   cerr << "Error opening " << "Mul2.tt" << endl;
   return 0;
}
inputmul2 >> Tmul2;
inputmul2.close();

cout << "t0=" << endl;
cout << TSS0 << endl << endl;
cout << "t1=" << endl;
cout << TSS1 << endl << endl;
F.puttt(TSS1);
G.puttt(Tmul2);
Comp(T21,F,G);
T2t1 = TT(T21);
cout << "0x2.t1=" << endl;
cout << T2t1 << endl;
F.kill();
G.kill();
F.puttt(TSS0);
```

(continues on next page)

(continued from previous page)

```
    G.puttt(Tmul2);
    Comp(T20,F,G);
    T2t0 = TT(T20);
    cout << "0x2.t0=" << endl;
    cout << T2t0 << endl;
    cout << "u0=t0+0x2.t1=" << endl;
    F.kill();
    F.puttt(TSS0);
    directsum(U0,F,T21);
    Tu0 = TT(U0);
    cout << Tu0 << endl;
    G.kill();
    cout << "u1=0x2.t0+t1=" << endl;
    G.puttt(TSS1);
    directsum(U1,T20,G);
    Tu1 = TT(U1);
    cout << Tu1 << endl;
    G.kill();
    cout << "y0=SS2(u0)=" << endl;
    G.puttt(TSS2);
    Comp(Y0,U0,G);
    Ty0 = TT(Y0);
    cout << Ty0 << endl;
    G.kill();
    cout << "y1=SS3(u1)=" << endl;
    G.puttt(TSS3);
    Comp(Y1,U1,G);
    Ty1 = TT(Y1);
    cout << Ty1 << endl;
    addimage(Y,Y0,Y1);
    Ty = TT(Y);
    cout << "y=(y0,y1)=" << endl;
    cout << Ty << endl;

    return 0;
}
```

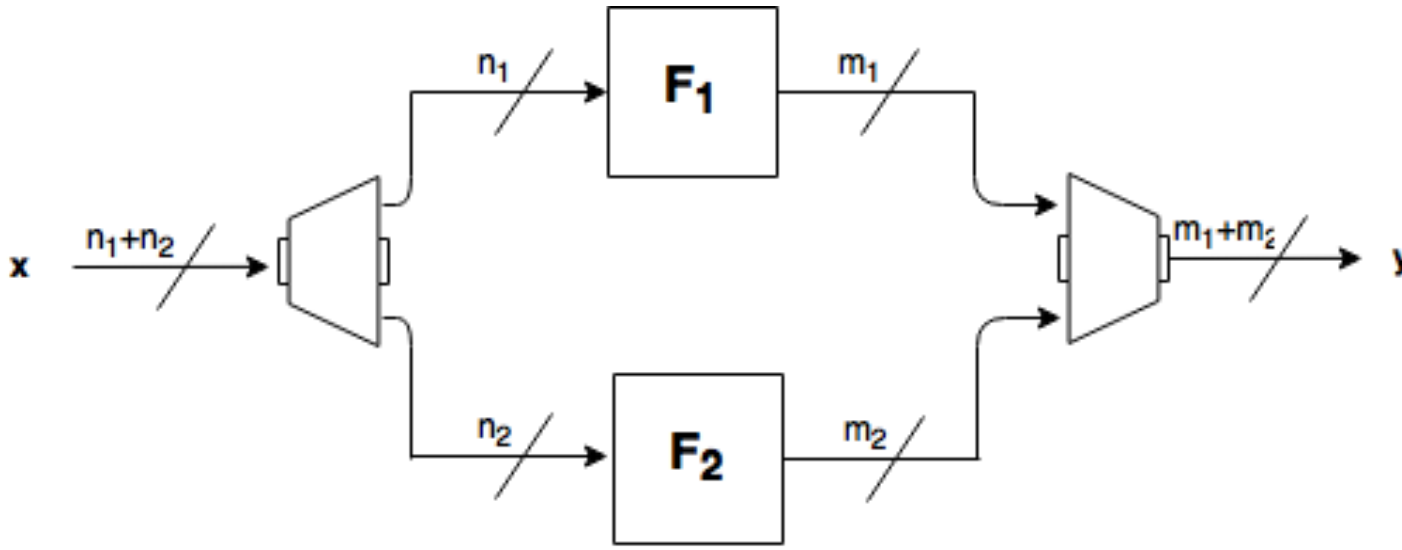
The output of this program is described in CLEFIA section in "Analysis of CRYPTEC project cryptographic algorithms".

Note that the output of $S_0$ S-box $\mathbf{y} \in \mathcal{F}_{8,8}$ is defined by the addition of coordinate functions of both $\mathbf{y_0} \in \mathcal{F}_{8,4}$ and $\mathbf{y_1} \in \mathcal{F}_{8,4}$.

## 6.9 Bricklayer

### 6.9.1 Description

Let $n_1, n_2, m_1, m_2 \geq 1$ and $F_1 \in \mathcal{F}_{n_1,m_1}$ , $F_2 \in \mathcal{F}_{n_2,m_2}$ and the Bricklayer function $F_1|F_2 \in \mathcal{F}_{n_1+n_2,m_1+m_2}$. Let $\mathbf{u_1} \in \mathrm{V}_{\mathrm{n}_1}$, $\mathbf{u_2} \in \mathrm{V}_{\mathrm{n}_2}$ and $\mathbf{u} = (\mathbf{u_1}, \mathbf{u_2})$, $\mathbf{v_1} \in \mathrm{V}_{\mathrm{m}_1}$, $\mathbf{v_2} \in \mathrm{V}_{\mathrm{m}_2}$ and $\mathbf{v} = (\mathbf{v_1}, \mathbf{v_2})$.



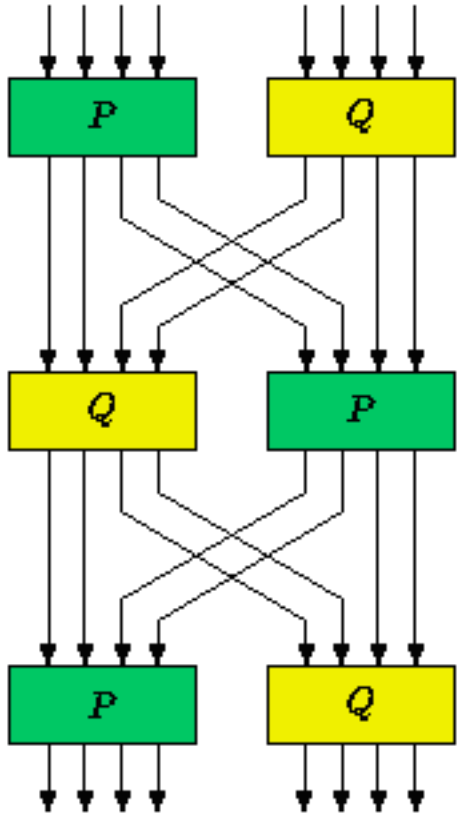

### 6.9.2 Library

It can be obtained with the following method:

```
void bricklayer(VBF& X, VBF& F, VBF& G)
```

### Example 1

KHAZAD is a block cipher designed by Paulo S. L. M. Barreto together with Vincent Rijmen, which was presented at the first NESSIE workshop in 2000, and, after some small changes, was selected as a finalist in the project. This cipher uses a $8 \times 8$ S-box composed of smaller pseudo-randomly generated $4 \times 4$ mini S-boxes (the P-box and the Q-box) as represented in the figure:




The following program provides the Truth Tables of the different intermediate constructions that allow to obtain KHAZAD S-box from $P$ and $Q$ mini S-boxes and the permutation that apply between them.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          P, Q, PQ, R, QP, S, T, U, A;
   NTL::mat_GF2 Tp, Tq;
   NTL::vec_ZZ  r;

   ifstream inputp("P.tt");
   if(!inputp) {
      cerr << "Error opening " << "P.tt" << endl;
      return 0;
   }
   inputp >> Tp;
   P.puttt(Tp);
   inputp.close();
```

(continues on next page)

(continued from previous page)

```
   ifstream inputq("Q.tt");
   if(!inputq) {
      cerr << "Error opening " << "Q.tt" << endl;
      return 0;
   }
   inputq >> Tq;
   Q.puttt(Tq);
   inputq.close();

   ifstream input("R.per");
   if(!input)  {
      cerr << "Error opening " << "R.per" << endl;
      return 0;
   }
   input >> r;
   R.putper(r);
   input.close();

   bricklayer(PQ,P,Q);
   cout << "Bricklayer of P and Q=" << endl;
   cout << TT(PQ) << endl;

   Comp(S,PQ,R);
   cout << "Composition of 1st bricklayer
   with permutation=" << endl;
   cout << TT(S) << endl;

   bricklayer(QP,Q,P);
   cout << "Bricklayer of Q and P=" << endl;
   cout << TT(QP) << endl;

   Comp(T,S,QP);
   cout << "Composition of previous result
   with 2nd bricklayer=" << endl;
   cout << TT(T) << endl;

   Comp(U,T,R);
   cout << "Composition of previous result
   with permutation=" << endl;
   cout << TT(U) << endl;

   Comp(A,U,PQ);
   cout << "Composition of previous result
   with 1st bricklayer=" << endl;
   cout << TT(A) << endl;

   return 0;
}
```

If we use the Truth Tables of $P$ and $Q$ and the representation of the permutation between them, the output are the Truth Tables described KHAZAD section in “Analysis of NESSIE project cryptographic algorithms”. Spectral radius, nonlinearity, linear potential, differential potential and linearity distance for bricklayer of $P$ and $Q$ mini S-boxes:

| S-box | $r$ | $NL$ | $lp$ | $dp$ | $AC_{max}$ | $L$ |
|---|---|---|---|---|---|---|
| $P$ | 8 | 4 | 0.25 | 0.25 | 8 | 2 |
| $Q$ | 8 | 4 | 0.25 | 0.25 | 8 | 2 |
| $P\|Q$ | 128 | 64 | 0.25 | 0.25 | 256 | 0 |
| $Q\|P$ | 128 | 64 | 0.25 | 0.25 | 256 | 0 |
| $R \circ (P\|Q)$ | 128 | 64 | 0.25 | 0.25 | 256 | 0 |
| $(Q\|P) \circ ((R \circ (P\|Q)))$ | 96 | 80 | 0.140625 | 0.125 | 160 | 24 |
| $R \circ ((Q\|P) \circ ((R \circ (P\|Q))))$ | 96 | 80 | 0.140625 | 0.125 | 160 | 24 |
| $S = (P\|Q) \circ (R \circ ((Q\|P) \circ ((R \circ (P\|Q)))))$ | 64 | 96 | 0.0625 | 0.03125 | 104 | 38 |

### Example 2

The following program provides the balancedness and correlation immunity (resiliency) of two Vector Boolean functions given its Truth Table in hexadecimal representation and calculates the same criteria for the bricklayering of $F$ and $G$ taking as inputs their Truth Tables in hexadecimal representation.

```
#include <iostream>
#include <fstream>
#include "VBF.h"

int main(int argc, char *argv[])
{
   using namespace VBFNS;

   VBF          F, G, H;

   ifstream input1(argv[1]);
   if(!input1) {
      cerr << "Error opening " << argv[1] << endl;
      return 0;
   }
   F.putHexTT(input1);
   input1.close();

   ifstream input2(argv[2]);
   if(!input2) {
      cerr << "Error opening " << argv[2] << endl;
      return 0;
   }
   G.putHexTT(input2);
   input2.close();

   cout << "Correlation immunity of F: " << CI(F) << endl;
   if (Bal(F)) {
     cout << "F is a balanced function" << endl;
   } else {
     cout << "F is a non-balanced function" << endl;
   }
```

(continues on next page)

(continued from previous page)

```
  cout << "Correlation immunity of G: " << CI(G) << endl;
  if (Bal(G)) {
    cout << "G is a balanced function" << endl;
  } else {
    cout << "G is a non-balanced function" << endl;
  }

  bricklayer(H,F,G);

  cout << "Correlation immunity of F bricklayer G: " << CI(H) << endl;
  if (Bal(H)) {
    cout << "F bricklayer G is a balanced function" << endl;
  } else {
    cout << "F bricklayer G is a non-balanced function" << endl;
  }

  return 0;
}
```

If we use the Boolean functions with the following Truth Tables (in hexadecimal representation) as inputs:

```
6cb405778ea9bd30
```

```
5c721bcaac27b1c5
```

The output would be the following:

```
Correlation immunity of F: 1
F is a balanced function
Correlation immunity of G: 2
G is a balanced function
Correlation immunity of F bricklayer G: 1
F bricklayer G is a balanced function
```

## 6.10 Summary

| Operations and constructions over VBF | |
|---|---|
| **SYNTAX** | **DESCRIPTION** |
| `long operator==(VBF& F, VBF& G)` | Returns *1* if $F$ and $G$ are equal |
| `void Comp(VBF& X, VBF& F, VBF& G)` | $X = G \circ F$ |
| `void inv(VBF& X, VBF& A)` | $X = F^{-1}$ |
| `void sum(VBF& X, VBF& F, VBF& G)` | $X = F + G$ |
| `void directsum(VBF& X, VBF& F, VBF& G)` | $X = F \oplus G$ |
| `void concat(VBF& X, VBF& F, VBF& G)` | $X(\mathbf{x}, x_{n+1}) = (x_{n+1} + 1)\, F(\mathbf{x}) + x_{n+1} G(\mathbf{x})$ |
| `void concatpol(VBF& X, VBF& F, VBF& G)` | $X(x_1, \ldots, x_{n_1}, x_{n_1+1}, \ldots, x_{n_1+n_2}) = F(x_1, \ldots, x_{n_1}) + G(x_{n_1+1}, \ldots, x_{n_1+n_2})$ |
| `void addimage(VBF& X, VBF& F, VBF& F)` | $X = (F, G)$ |
| `void bricklayer(VBF& X, VBF& F, VBF& G)` | $X = F \vert G$ |

# 7 Analysis of AES competition cryptographic algorithms

In January 1997, the US National Institute of Standards and Technology (NIST) announced the start of an initiative to develop a new encryption standard: the AES. The AES selection process was open in which 15 candidates were accepted for the first evaluation round and 5 finalists were announced in the second round. On October 2, 2000, NIST officially announced that Rijndael would become the AES. In this chapter, a number of cryptographic algorithms from the AES (Advanced Encryption Standard) candidates accepted for the first evaluation round process are analysed.

Below you can find a legend describing the cryptographic criteria used in this chapter:

| | |
|---|---|
| **NL** | Nonlinearity |
| **NL2** | 2-nd order nonlinearity |
| **LD** | Linearity distance |
| **DEG** | Algebraic degree |
| **AI** | Algebraic immunity |
| **MAXAC** | Absolute indicator |
| $\sigma$ | Sum-of-squares indicator |
| **LP** | Linear potential |
| **DP** | Differential Potential |

**Hyperlinks to representations**

Open the hyperlinks to representations below in a new browser window or in a new tab.

## 7.1 CAST-256

### 7.1.1 Description

*CAST-256* is a symmetric-key block cipher published in June 1998. It was submitted as a candidate for the Advanced Encryption Standard (AES); however, it was not among the five AES finalists. It is an extension of an earlier cipher, CAST-128; both were designed according to the "CAST" design methodology invented by Carlisle Adams and Stafford Tavares. Howard Heys and Michael Wiener also contributed to the design. It has four 8x32 S-boxes: S1,S2,S3 and S4.

### 7.1.2 S1

#### Representations

Polynomial representation in ANF

Truth Table

ANF Table

### 7.1.3 S2

#### Representations

Polynomial representation in ANF

Truth Table

ANF Table

### 7.1.4 S3

#### Representations

Polynomial representation in ANF

Truth Table

ANF Table

### 7.1.5 S4

#### Representations

Polynomial representation in ANF

Truth Table

ANF Table

## 7.2 Crypton

### 7.2.1 Description

*CRYPTON* is a symmetric block cipher designed by Chae Hoon Lim of Future Systems Inc. In this section, we study both the AES proposal (v0.5) and the revised version (v1.0). In v0.5 the authors used two 8x8 S-boxes constructed from 4-bit permutations using a 3-round Feistel Cipher (Tabular representation of S-boxes). In v1.0 the authors used four variants of one S-box, instead of independent four S-boxes to allow greater flexibility in memory requirements. The four 8x8 S-boxes are $S_i (0 \leq i \leq 3)$, such that $S2 = {S_0}^{-1}$ and $S3 = {S_1}^{-1}$

### 7.2.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| S0 (v0.5) | 96 | 32 | 5 | 3 | 128 | 581632 | 0.0625 | 0.03125 |
| S1 (v0.5) | 96 | 28 | 5 | 3 | 144 | 581632 | 0.0625 | 0.03125 |
| S0 (v1.0) | 96 | 40 | 6 | 4 | 96 | 280192 | 0.0625 | 0.0390625 |
| S1 (v1.0) | 96 | 40 | 6 | 4 | 96 | 280192 | 0.0625 | 0.0390625 |
| S2 (v1.0) | 96 | 40 | 6 | 4 | 96 | 280192 | 0.0625 | 0.0390625 |
| S3 (v1.0) | 96 | 40 | 6 | 4 | 96 | 280192 | 0.0625 | 0.0390625 |

### 7.2.3 S0 (v0.5)

#### Representations

Polynomial representation in ANF

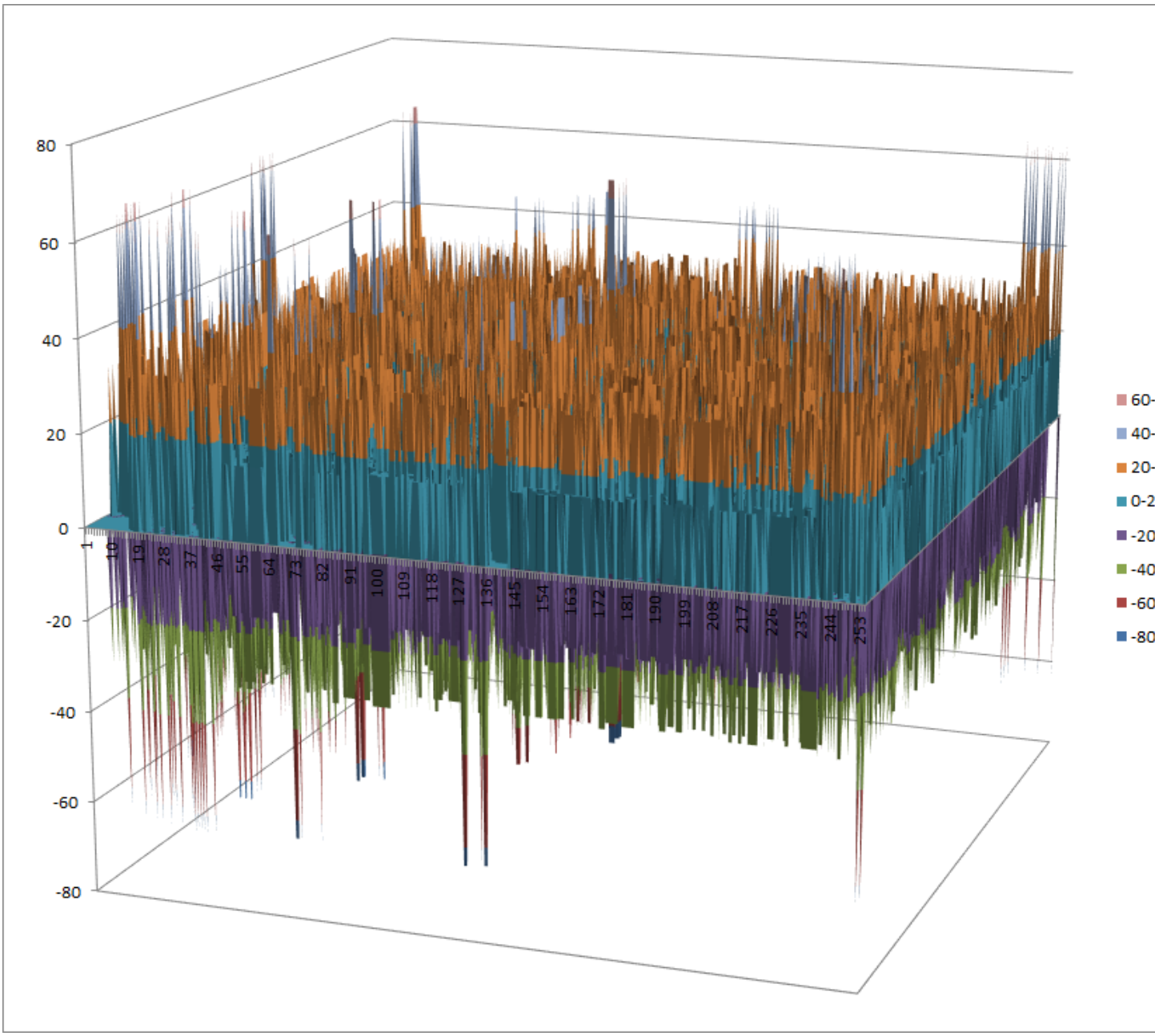
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 3 | 1 |
| 7 | 3 |
| 10 | 2 |
| 11 | 2 |
| 15 | 1 |
| 16 | 2 |
| 17 | 1 |
| 22 | 1 |
| 42 | 1 |
| 61 | 1 |

There are no linear structures

It has 1 fixed point: (0,0,0,0,1,1,1,1)

It has 1 negated fixed point: (1,1,1,0,1,1,1,1)

### 7.2.4 S1 (v0.5)

#### Representations

Polynomial representation in ANF

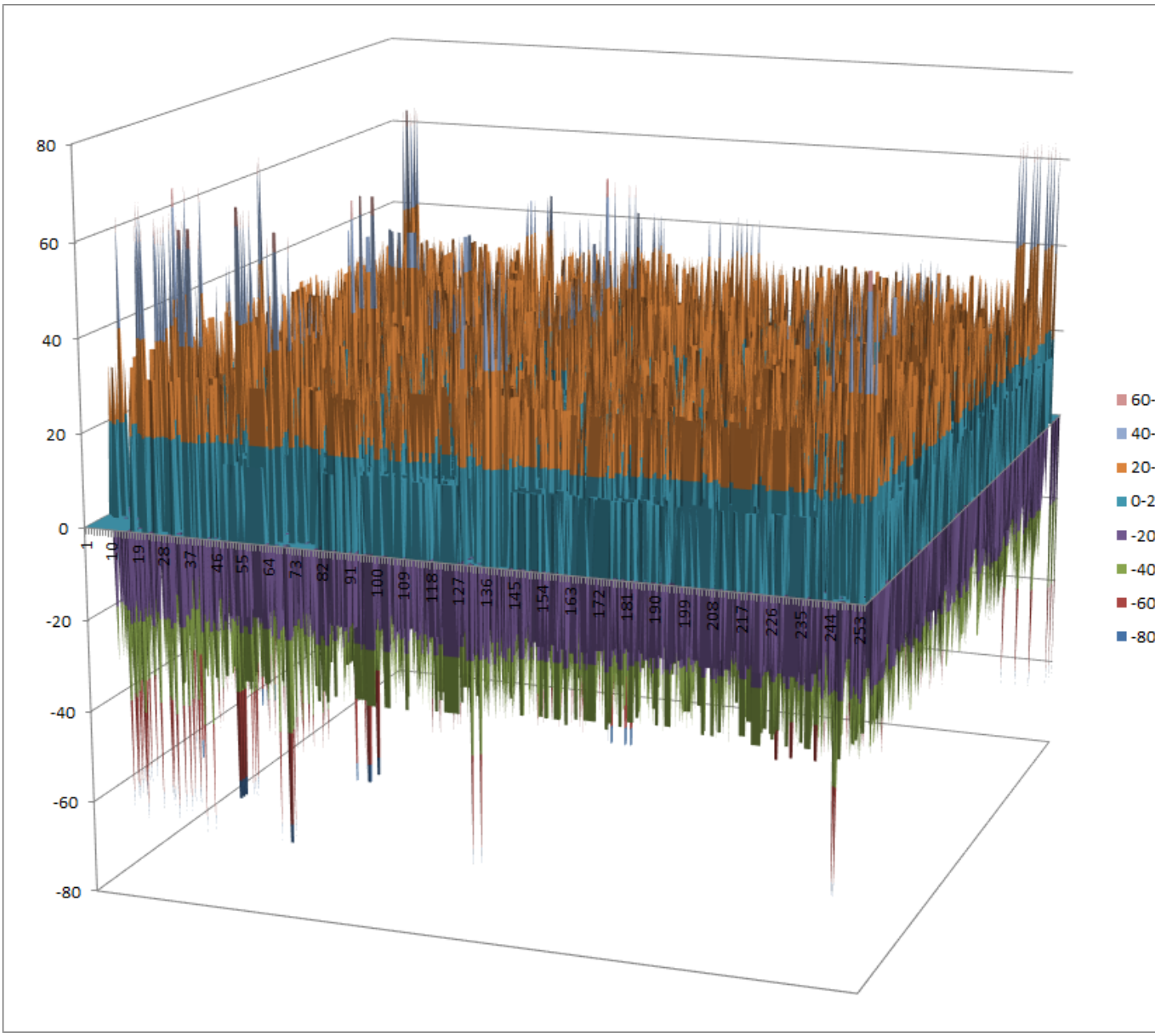
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 3 | 1 |
| 7 | 3 |
| 10 | 2 |
| 11 | 2 |
| 15 | 1 |
| 16 | 2 |
| 17 | 1 |
| 22 | 1 |
| 42 | 1 |
| 61 | 1 |

There are no linear structures

It has 1 fixed point: (0,0,0,0,1,1,1,1)

It has 1 negated fixed point: (0,0,0,1,0,0,0,0)

### 7.2.5 S0 (v1.0)

#### Representations

Polynomial representation in ANF

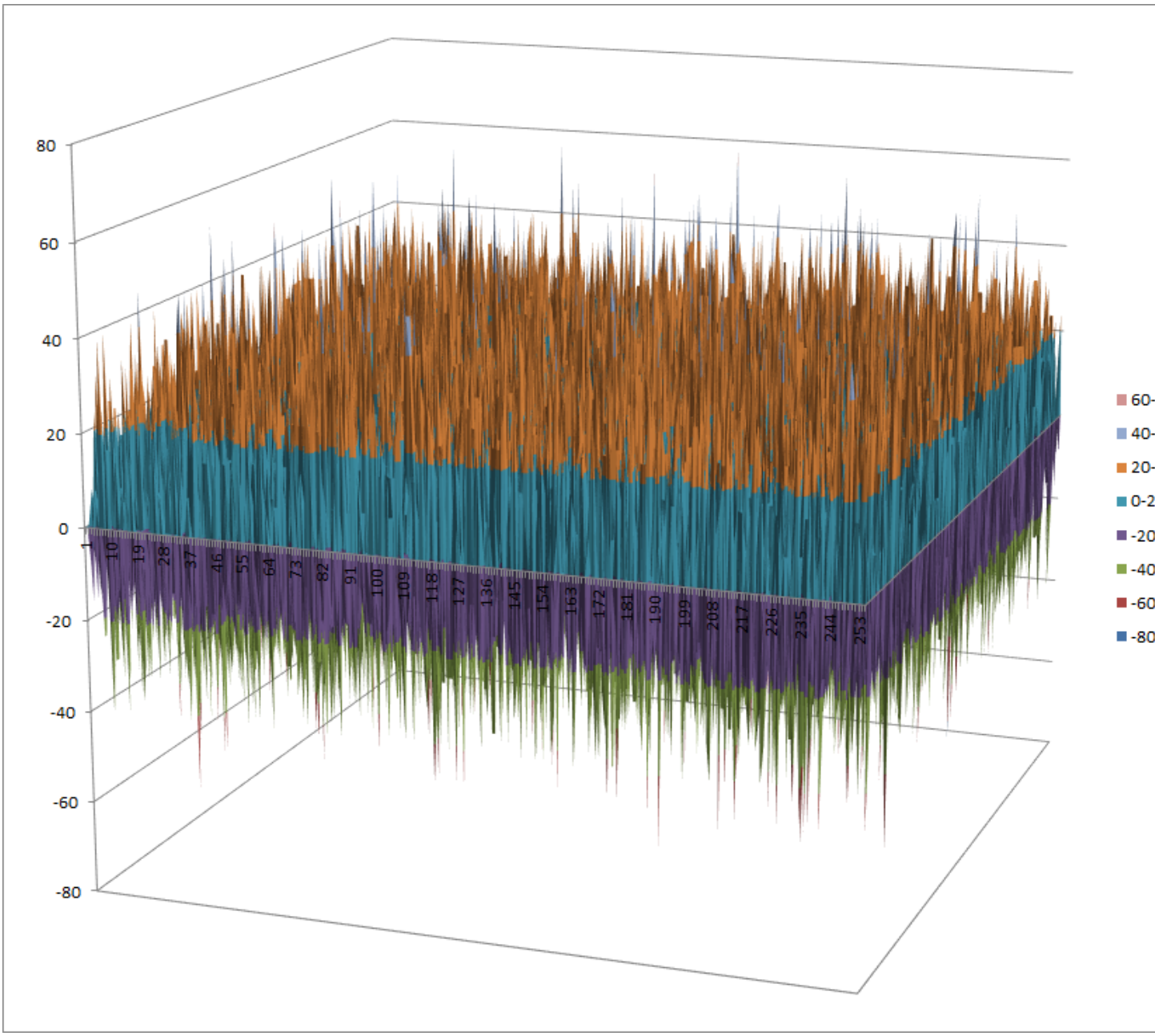
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 5 | 1 |
| 248 | 1 |

There are no linear structures

It has 1 fixed point: (0,1,1,1,0,1,0,1)

It has no negated fixed points

### 7.2.6 S1 (v1.0)

#### Representations

Polynomial representation in ANF

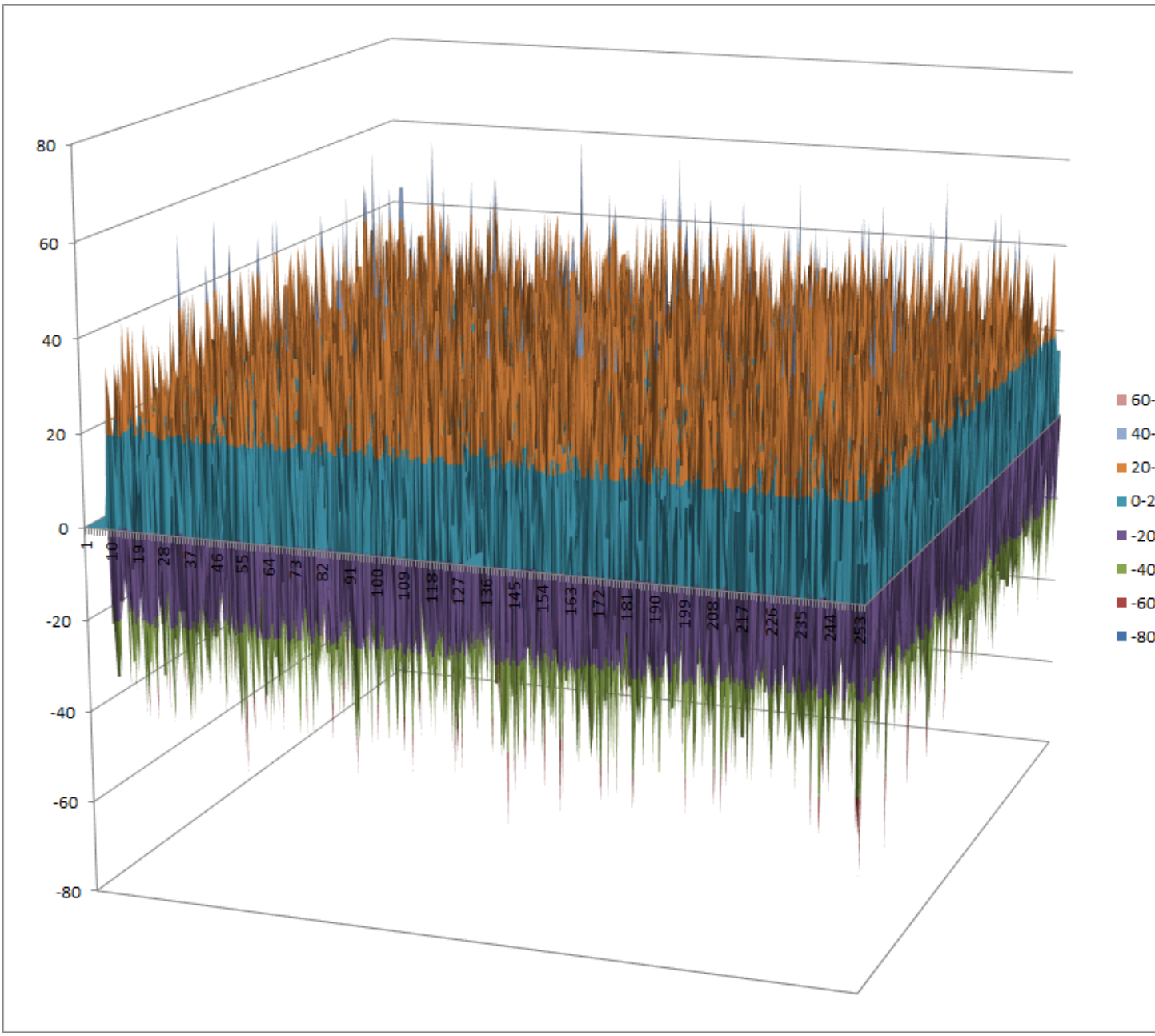
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 3 | 1 |
| 5 | 1 |
| 12 | 1 |
| 22 | 1 |
| 28 | 1 |
| 48 | 1 |
| 50 | 1 |
| 88 | 1 |

There are no linear structures

It has no fixed points

It has 1 negated fixed point: (1,0,1,0,1,1,1,0)

### 7.2.7 S2 (v1.0)

#### Representations

Polynomial representation in ANF

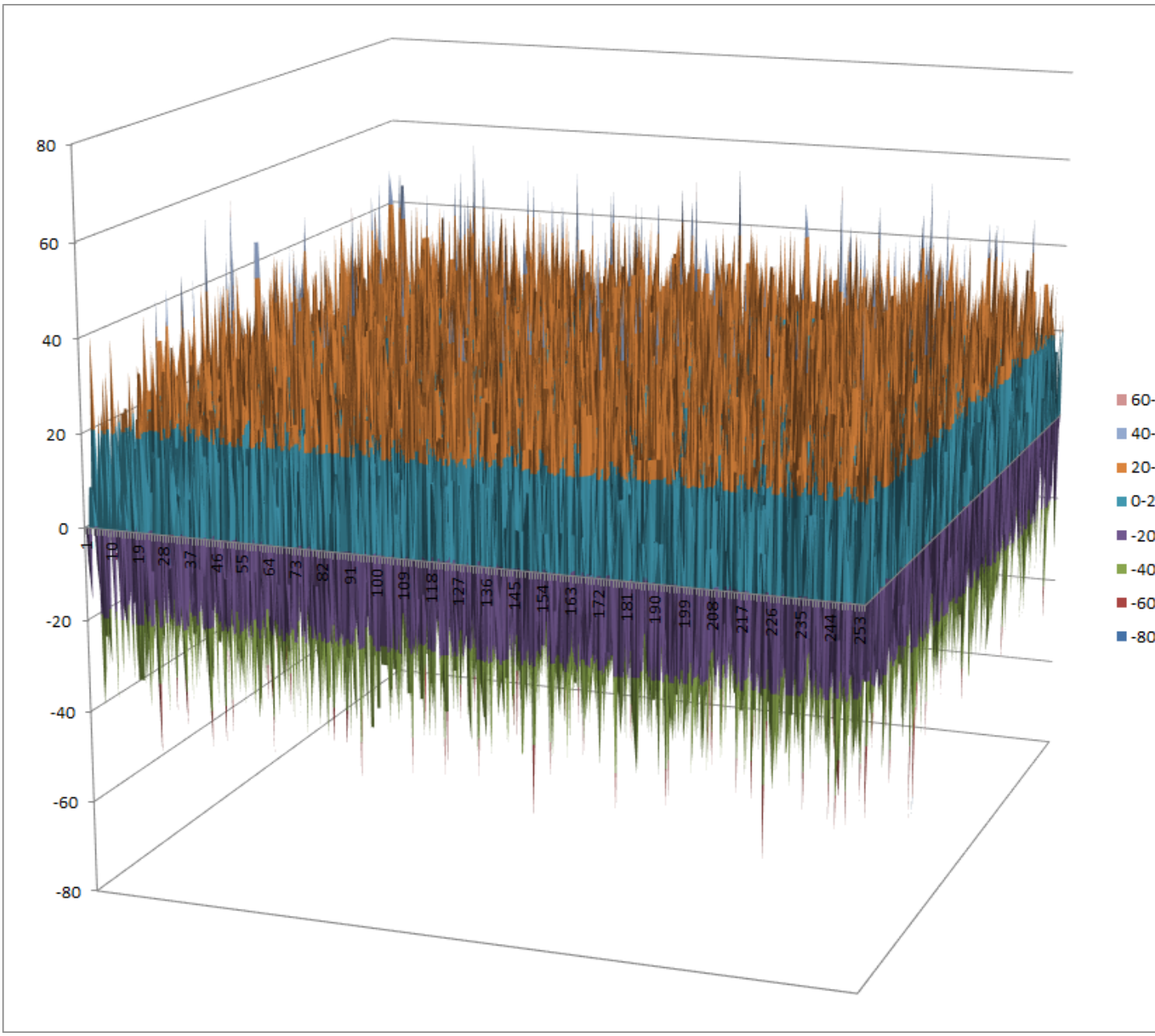
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 5 | 1 |
| 248 | 1 |

There are no linear structures

It has 1 fixed point: (0,1,1,1,0,1,0,1)

It has no negated fixed points

### 7.2.8 S3 (v1.0)

#### Representations

Polynomial representation in ANF

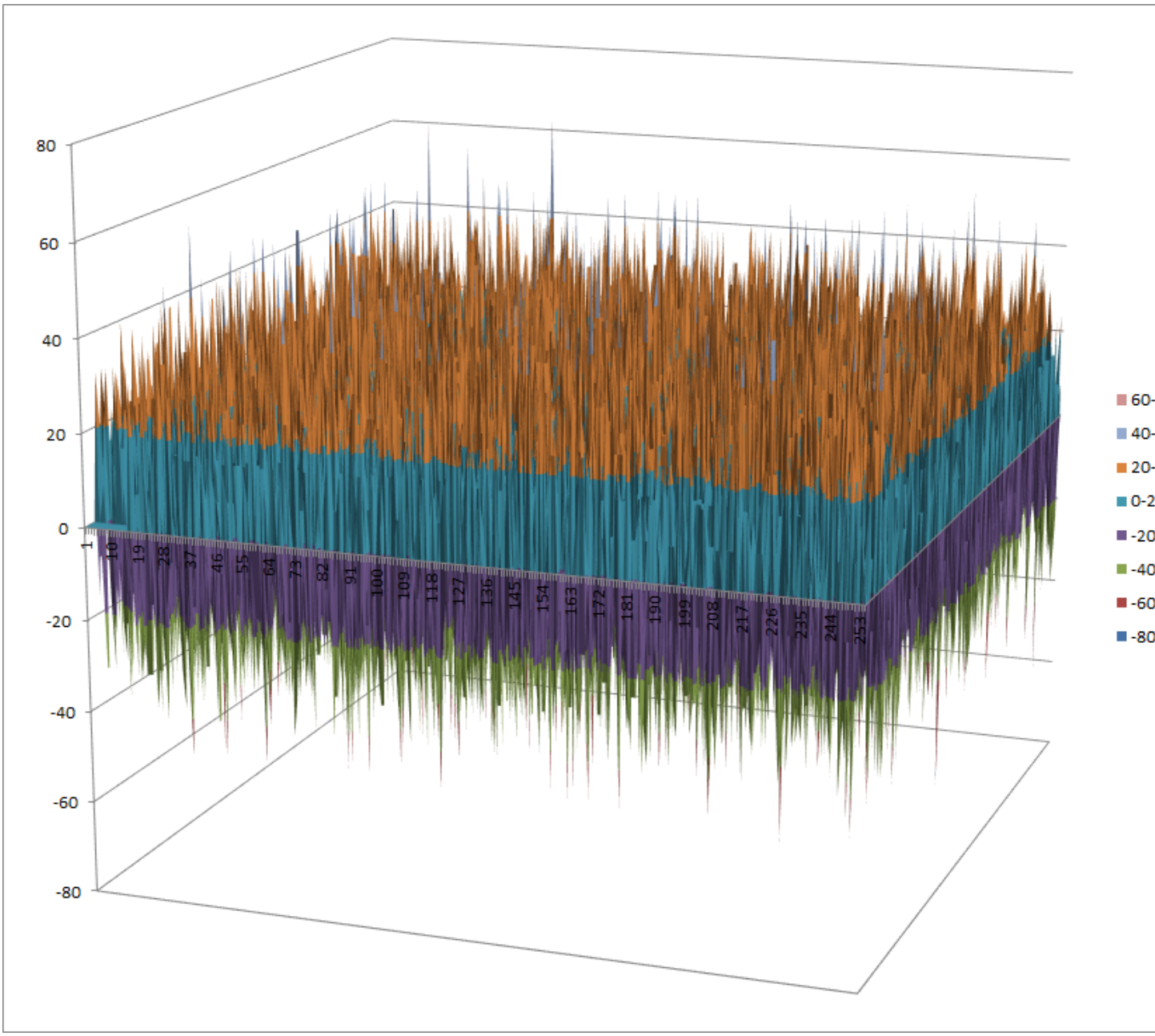
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 3 | 1 |
| 5 | 1 |
| 12 | 1 |
| 22 | 1 |
| 28 | 1 |
| 48 | 1 |
| 50 | 1 |
| 88 | 1 |

There are no linear structures

It has no fixed points

It has 1 negated fixed point: (0,1,0,1,0,0,0,1)

## 7.3 DEAL

### 7.3.1 Description

*DEAL* (Data Encryption Algorithm with Larger blocks) is a symmetric block cipher derived from the Data Encryption Standard (DES). The design was proposed in a report by Lars Knudsen in 1998, and was submitted to the AES competition by Richard Outerbridge (who notes that Knudsen had presented the design at the SAC conference in 1997).

DEAL has eight 6x4 S-boxes: S1, S2, S3, S4, S5, S6, S7, S8. They are the same as DES S-boxes and you can see an analysis in DES.

## 7.4 E2

### 7.4.1 Description

*E2* is a symmetric block cipher which was created in 1998 by NTT and submitted to the AES competition. E2 has an input transformation and output transformation that both use modular multiplication, but the round function itself consists only of XORs and S-box lookups. The single 8×8-bit S-box is constructed from the composition of an affine transformation with the discrete exponentiation $x^{127}$ over the finite field $GF(2^8)$. NTT adopted many of E2's special characteristics in Camellia, which has essentially replaced E2.

### 7.4.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| S | 100 | 38 | 6 | 4 | 104 | 236800 | 0.0478515625 | 0.0390625 |

### 7.4.3 S

#### Representations

Polynomial representation in ANF

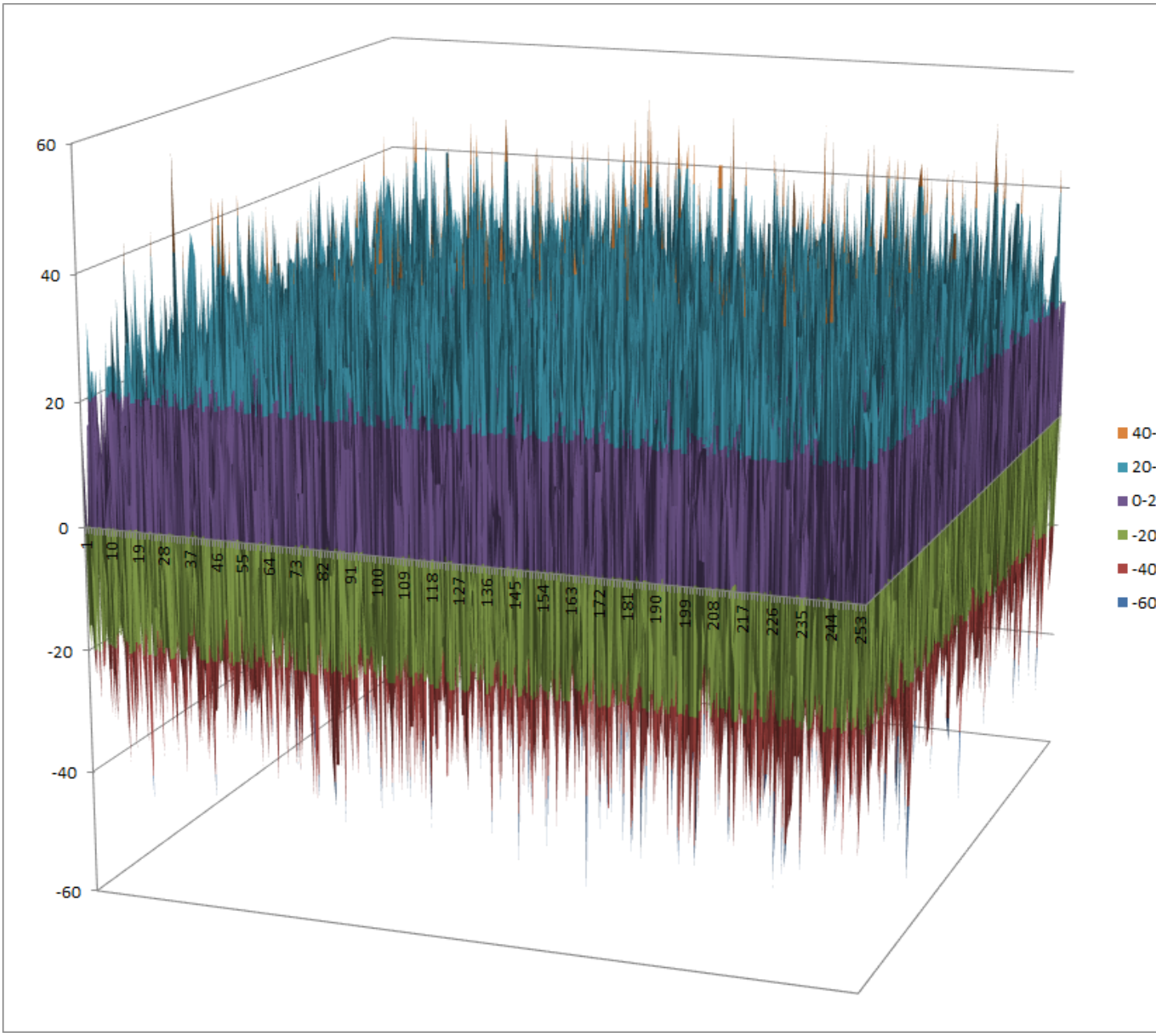
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 3 | 1 |
| 7 | 1 |
| 24 | 1 |
| 64 | 1 |
| 156 | 1 |

There are no linear structures

It has no fixed points.

It has 2 negated fixed points: (0,0,1,0,1,0,0,1), (1,0,0,0,1,0,1,1)

## 7.5 LOKI97

### 7.5.1 Description

*LOKI97* is a block cipher which was a candidate in the Advanced Encryption Standard competition. It is a member of the LOKI family of ciphers, earlier instances being LOKI89 and LOKI91. LOKI97 was designed by Lawrie Brown, assisted by Jennifer Seberry and Josef Pieprzyk. The LOKI97 round function uses two columns each with multiple copies of two basic S-boxes. The S-boxes chosen for LOKI97 use cubing in a galois field $GF(2^n)$ with n odd. In order to use odd sized inputs, S1 uses 13 input bits, and S2 uses 11. The S-box functions are:

S1[x]=((x XOR 1FFF)^3 mod 2911) & FF $\in GF(2^{13})$

S2[x]=((x XOR 7FF)^3 mod AA7) & FF $\in GF(2^{11})$

where all constant above are written in hex and all computations are done as polynomials in $GF(2^n)$.

### 7.5.2 Summary

| S-box | size | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S1 | 13x8 | 4032 | 0 | 2 | 2 | 8192 | 134217728 | 0.000244140625 | 0.0078125 |
| S2 | 11x8 | 992 | 0 | 2 | 2 | 2048 | 8388608 | 0.0009765625 | 0.0078125 |

### 7.5.3 S1

#### Representations

Polynomial representation in ANF

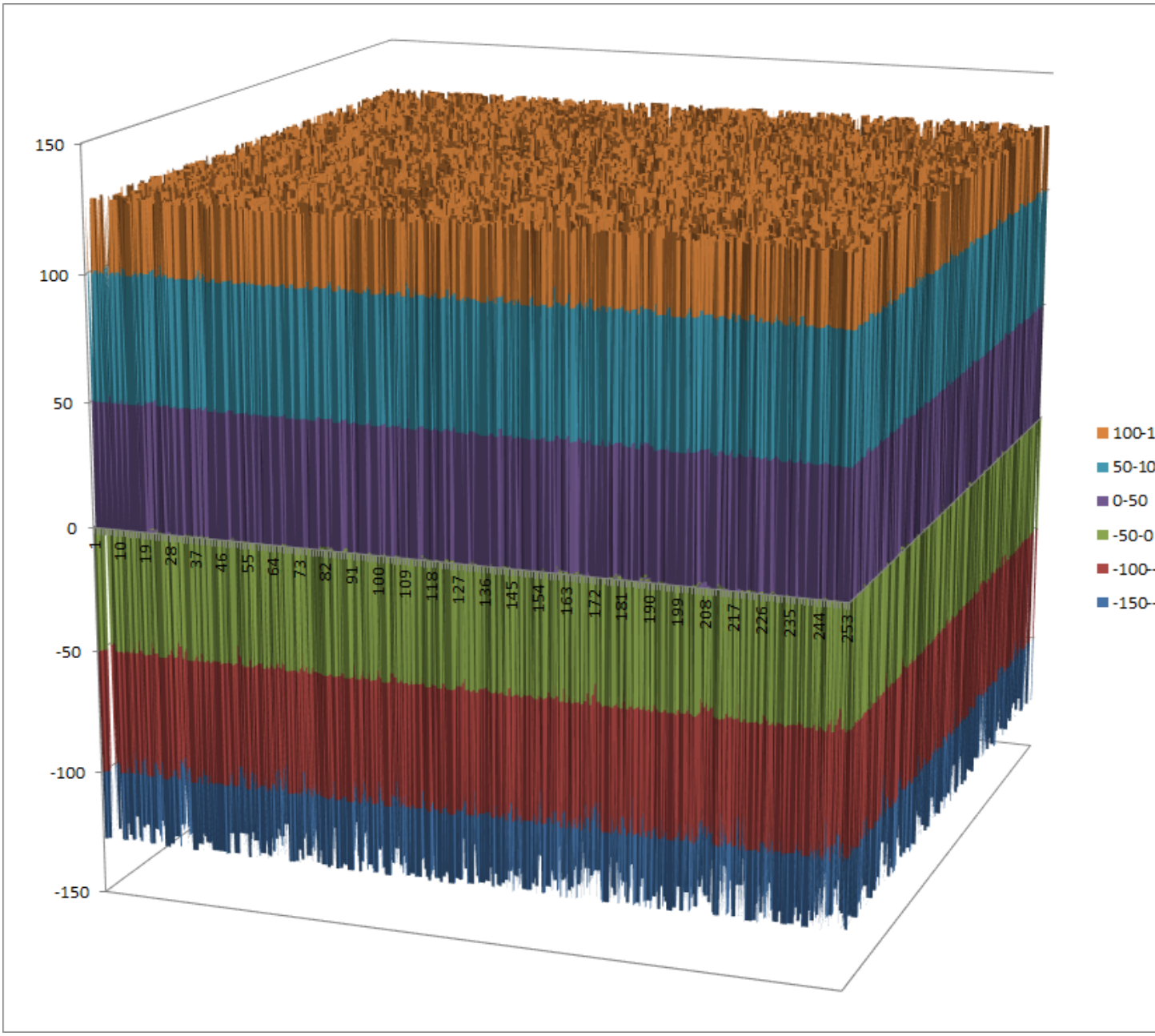
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (256x256 first values except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are 255 linear structures:

Linear structures

### 7.5.4 S2

#### Representations

Polynomial representation in ANF

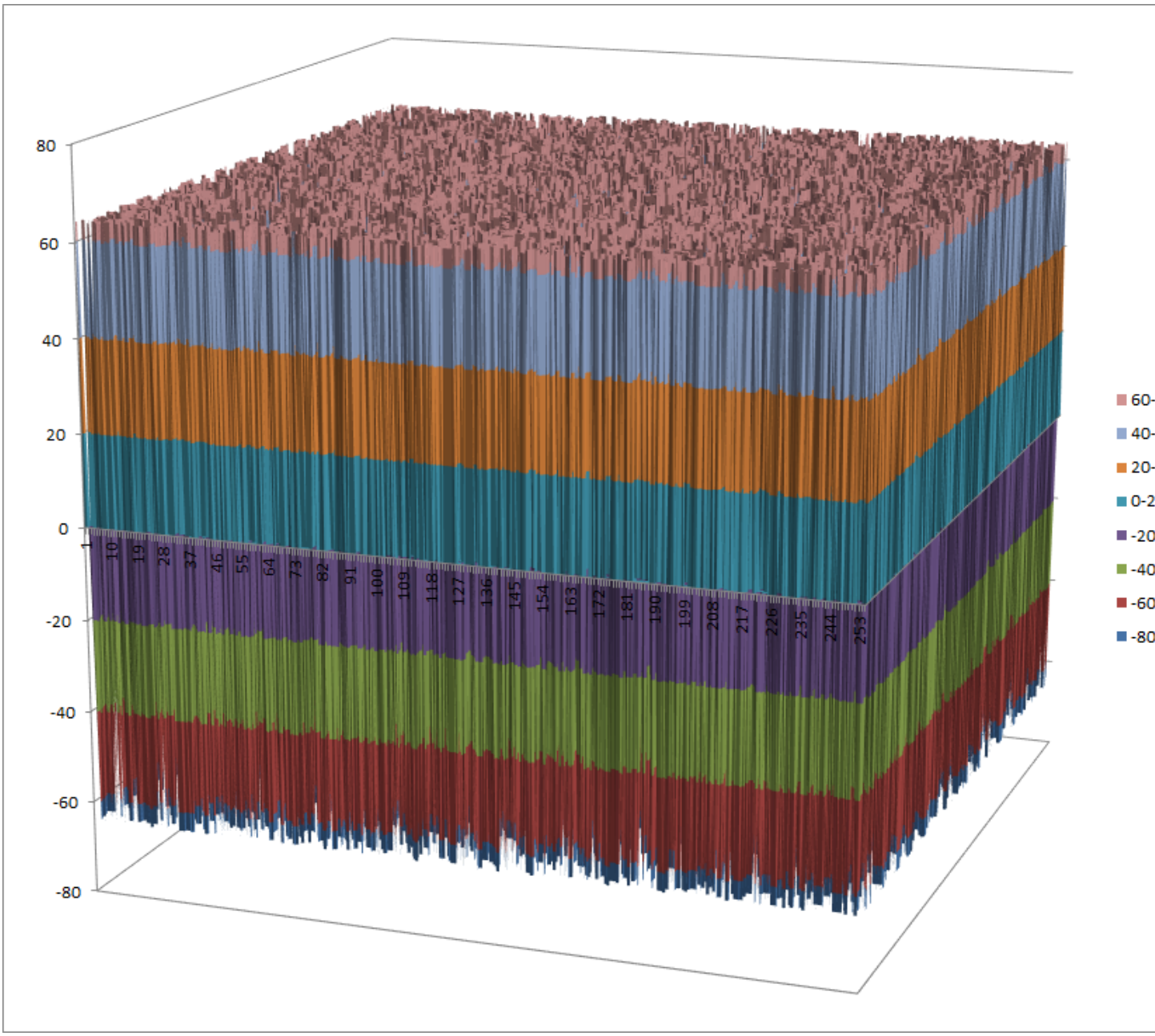
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (256x256 first values except first row and column):



Linear Profile

Differential Profile

Autocorrelation Spectrum

#### Other useful information in cryptanalysis

There are 256 linear structures:

Linear structures

## 7.6 Magenta

### 7.6.1 Description

*Magenta* is a symmetric key block cipher developed by Michael Jacobson Jr. and Klaus Huber for Deutsche Telekom. The cipher was submitted to the Advanced Encryption Standard process, but did not advance beyond the first round; cryptographic weaknesses were discovered and it was found to be one of the slower ciphers submitted. It has one 8x8 S-box.

### 7.6.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| S | 102 | 44 | 7 | 4 | 80 | 217600 | 0.04125976563 | 0.03125 |

### 7.6.3 S

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^2 + 1$: Trace representation

Polynomial representation in ANF

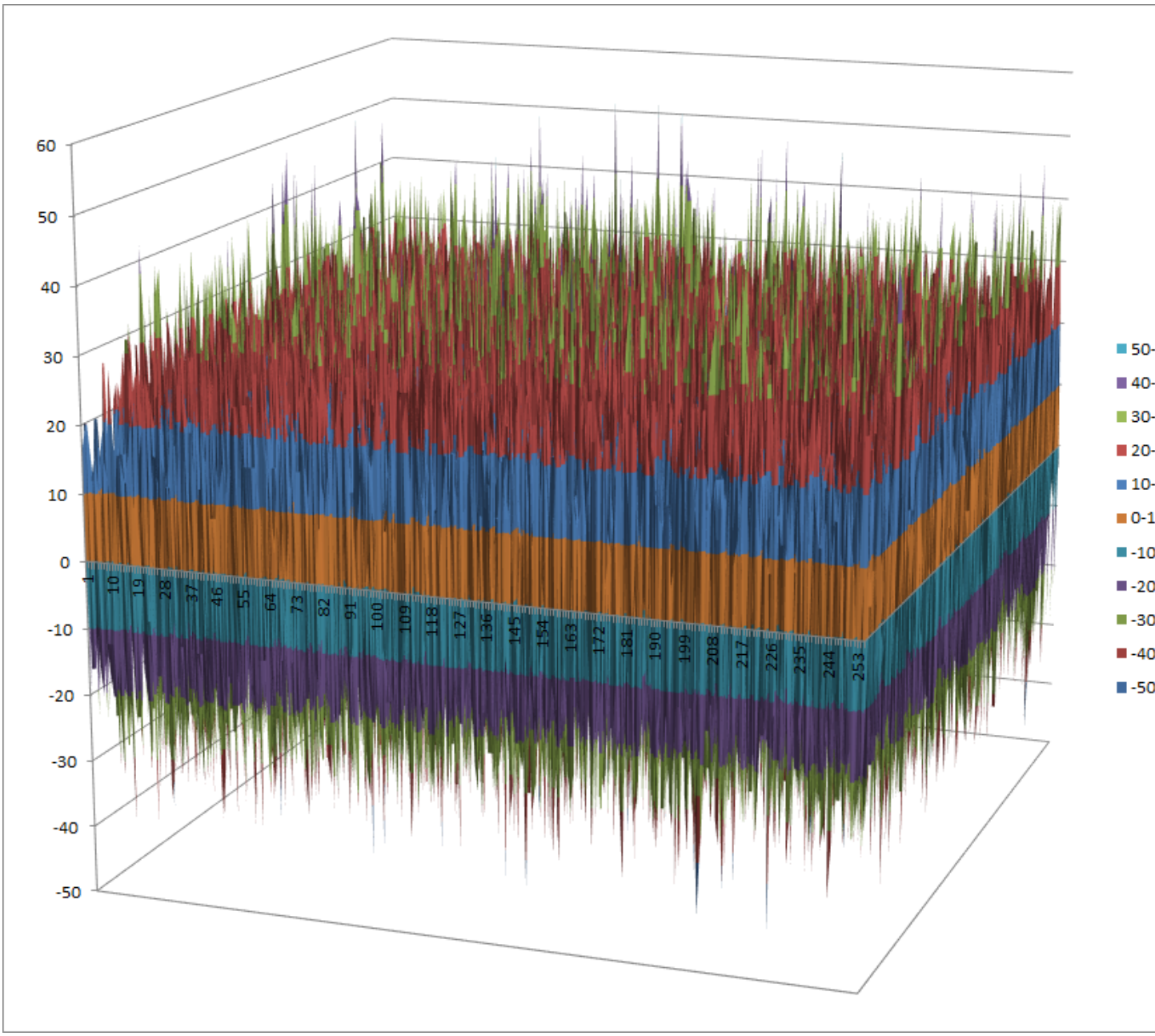
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 5 | 2 |
| 9 | 1 |
| 38 | 1 |
| 198 | 1 |

There are no linear structures

It has 1 fixed point: (1,0,1,0,0,0,0,1)

It has 2 negated fixed points: (0,0,0,1,1,1,0,0), (1,1,1,1,1,1,1,1)

## 7.7 Mars

### 7.7.1 Description

*Mars* is a block cipher that was IBM's submission to the Advanced Encryption Standard process. MARS was selected as an AES finalist in August 1999, after the AES2 conference in March 1999, where it was voted as the fifth and last finalist algorithm. It has two 8x32 S-boxes: S0 and S1

### 7.7.2 S0

#### Representations

Polynomial representation in ANF

Truth Table

ANF Table

### 7.7.3 S1

#### Representations

Polynomial representation in ANF

Truth Table

ANF Table

## 7.8 Rijndael

### 7.8.1 Description

*Rijndael*, later called *AES*, is a block cipher used for securing sensitive but unclassified material by U.S. Government agencies since December 6, 2001 and has become the de facto encryption standard for commercial transactions in the private sector.

$S_{RD}$ is AES 8x8 S-box and it is generated by determining the multiplicative inverse for a given number in $GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$, Rijndael's finite field. ${S_{RD}}^{-1}$ is the inverse S-box and it is simply the S-box $S_{RD}$ run in reverse.

The tabular representation of $S_{RD}$ S-box represented in hexadecimal notation is the following:

Tabular representation in hexadecimal notation

Here the column is determined by the least significant nibble (four-bit aggregation), and the row is determined by the most significant nibble. For example, the value 0x9a is converted into 0xb8 by Rijndael's S-box. Note that the multiplicative inverse of 0x00 is defined as itself.

The tabular representation of ${S_{RD}}^{-1}$ S-box represented in hexadecimal notation is the following:

Tabular representation in hexadecimal notation

For hardware implementations, it might be useful to use the following decomposition of $S_{RD}$:

$S_{RD}[a] = f(g(a))$

where $g(a)$ is the mapping

$a \to a^{-1}$ in $GF(2^8)$, and $f(a)$ is an affine mapping. Since $g(a)$ is self-inverse, we have:

${S_{RD}}^{-1}[a] = g^{-1}(f^{-1}(a)) = g(f^{-1}(a))$

The tabular representation of $g$ mapping is represented in hexadecimal notation as follows:

Tabular representation in hexadecimal notation

The tabular representation of $f$ mapping is represented in hexadecimal notation as follows:

Tabular representation in hexadecimal notation

The tabular representation of $f^{-1}$ mapping is represented in hexadecimal notation as follows:

Tabular representation in hexadecimal notation

In the algorithm of Rijndael there are multiplications of a variable with a constant. A mapping called xtime is implemented in the algorithm in order to multiply by 02. Since all elements of $GF(2^8)$ can be written as a sum of powers of 02, multiplication by any constant can be implemented by a repeated use of xtime. The tabular representation of xtime mapping is represented in hexadecimal notation as follows:

Tabular representation in hexadecimal notation

### 7.8.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| $S_{RD}$ | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| ${S_{RD}}^{-1}$ | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| $g$ | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| $f$ | 0 | 0 | 1 | 1 | 256 | 16777216 | 1 | 1 |
| $f^{-1}$ | 0 | 0 | 1 | 1 | 256 | 16777216 | 1 | 1 |
| xtime | 0 | 1 | 1 | 1 | 256 | 16777216 | 1 | 0.9921875 |

### 7.8.3 $S_{RD}$

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$: Trace representation

Polynomial representation in ANF

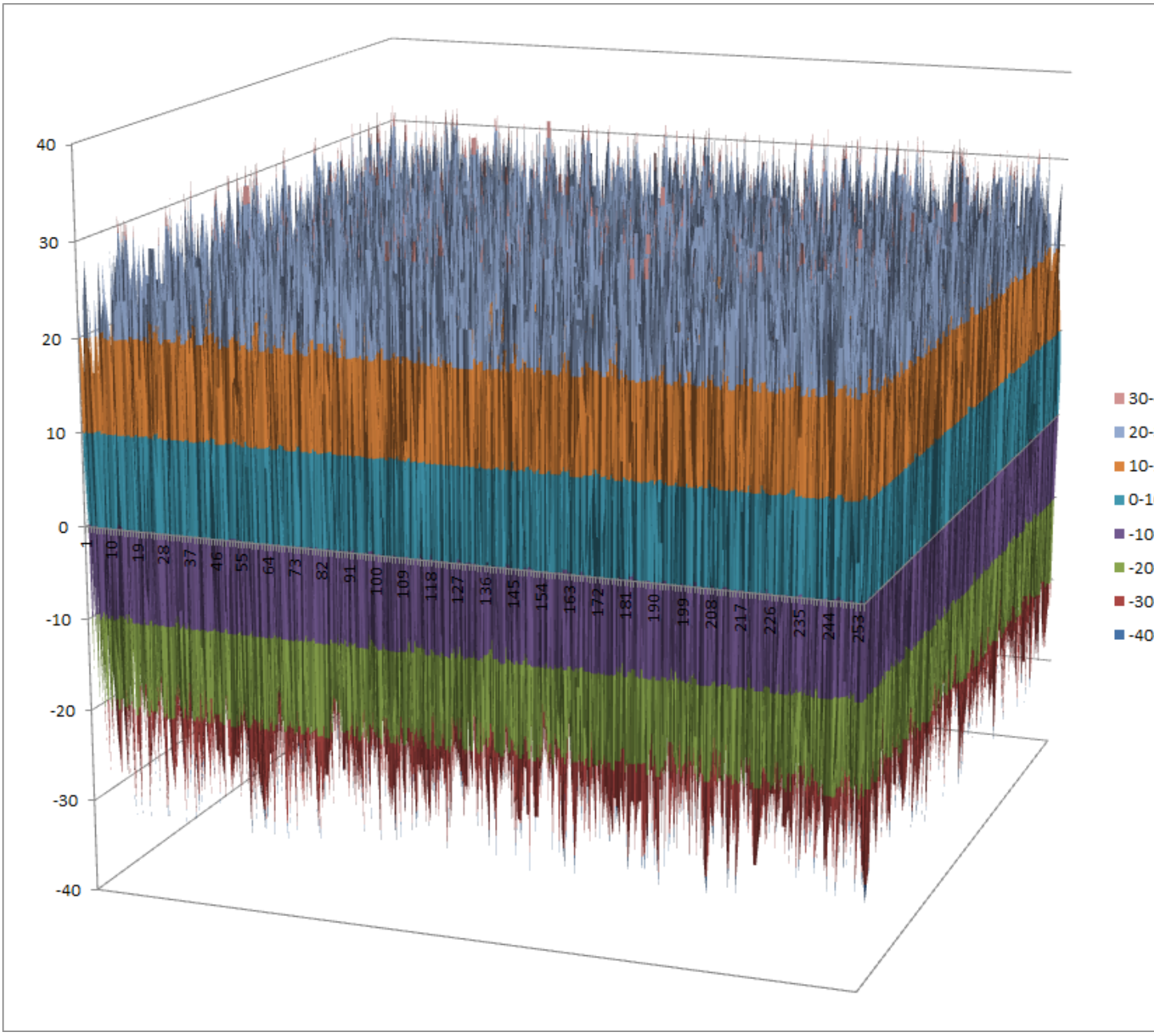
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 27 | 1 |
| 59 | 1 |
| 81 | 1 |
| 87 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points

### 7.8.4 ${S_{RD}}^{-1}$

### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$: Trace representation

Polynomial representation in ANF

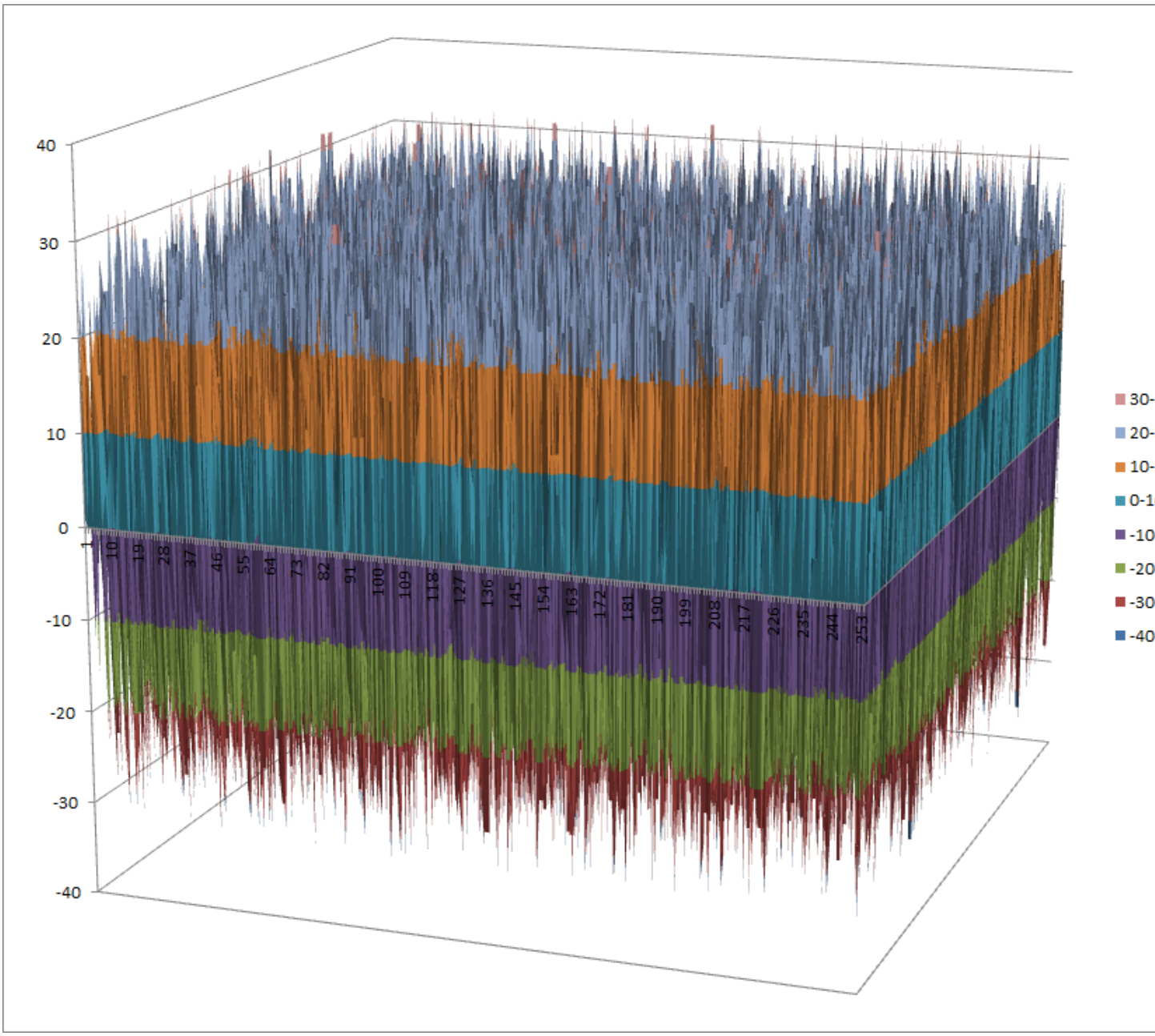
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 27 | 1 |
| 59 | 1 |
| 81 | 1 |
| 87 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points

### 7.8.5 $g$

#### Representations

Polynomial representation in ANF

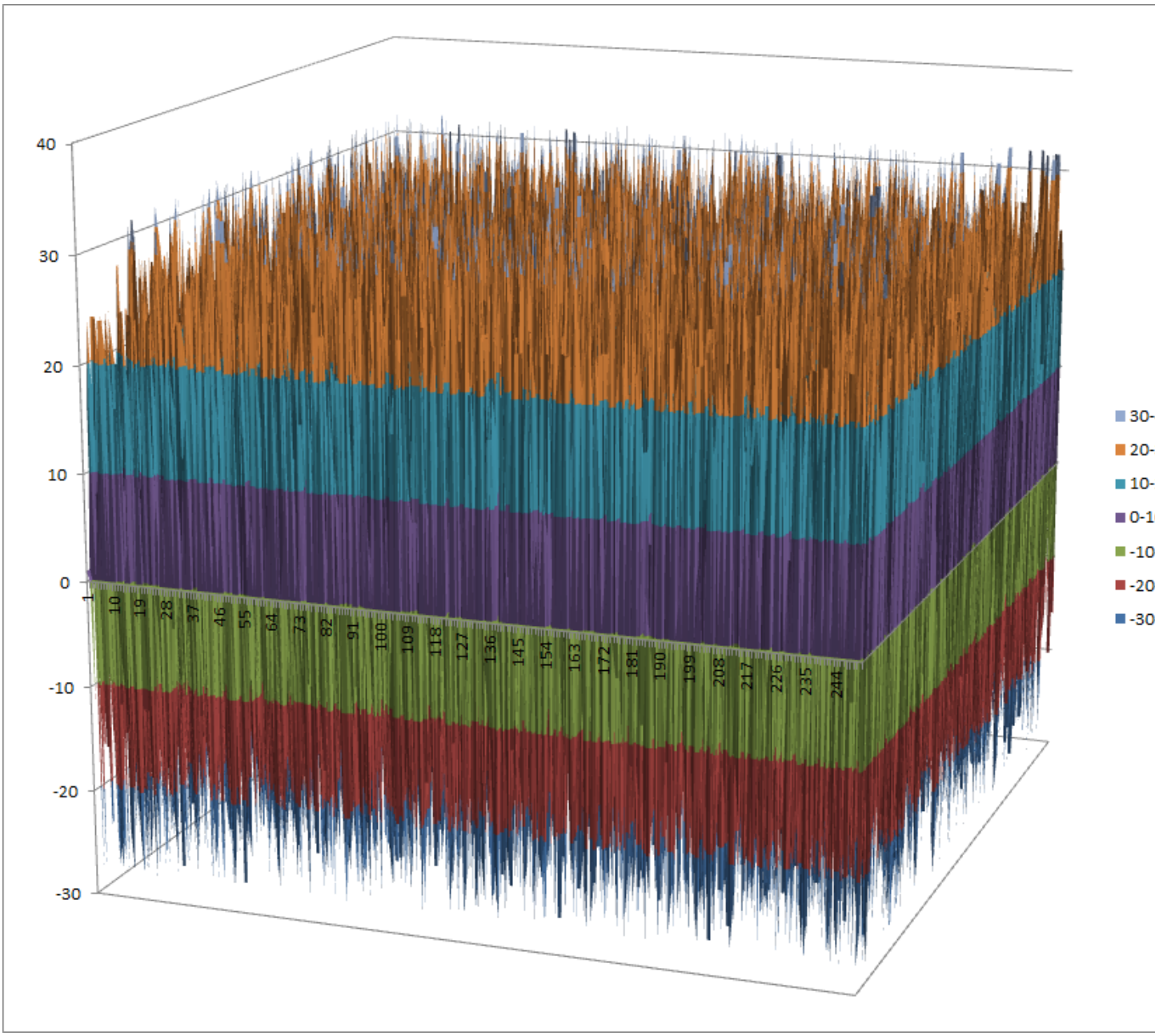
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 2 | 127 |

There are no linear structures

It has 2 fixed points: (0,0,0,0,0,0,0,0), (0,0,0,0,0,0,0,1)

It has no negated fixed points: (0,1,1,1,1,1,1,0), (1,0,0,0,0,0,0,1)

### 7.8.6 $f$

#### Representations

Polynomial representation in ANF

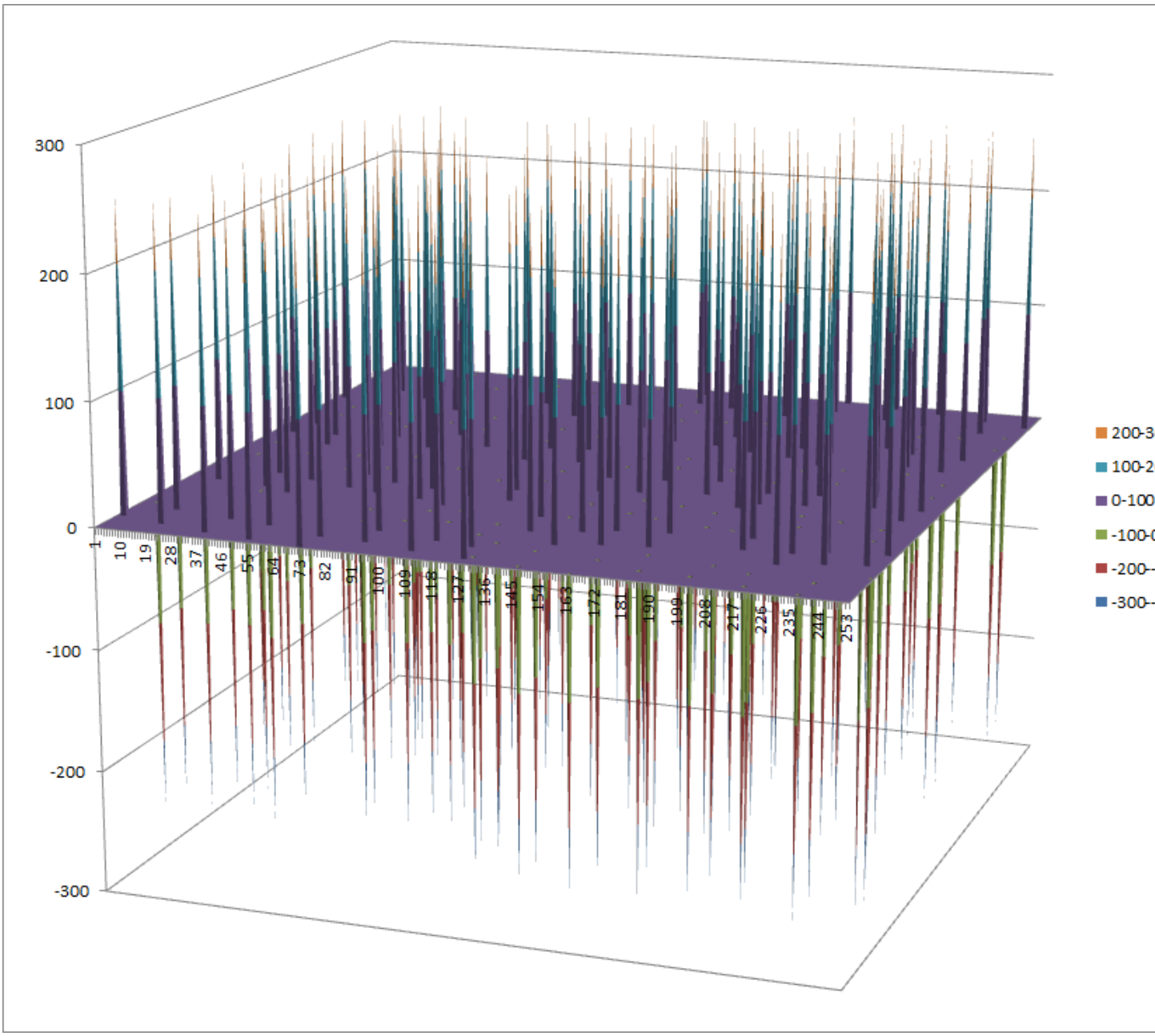
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 4 | 64 |

There are 65025 linear structures

It has no fixed points. It has no negated fixed points

### 7.8.7 $f^{-1}$

#### Representations

Polynomial representation in ANF

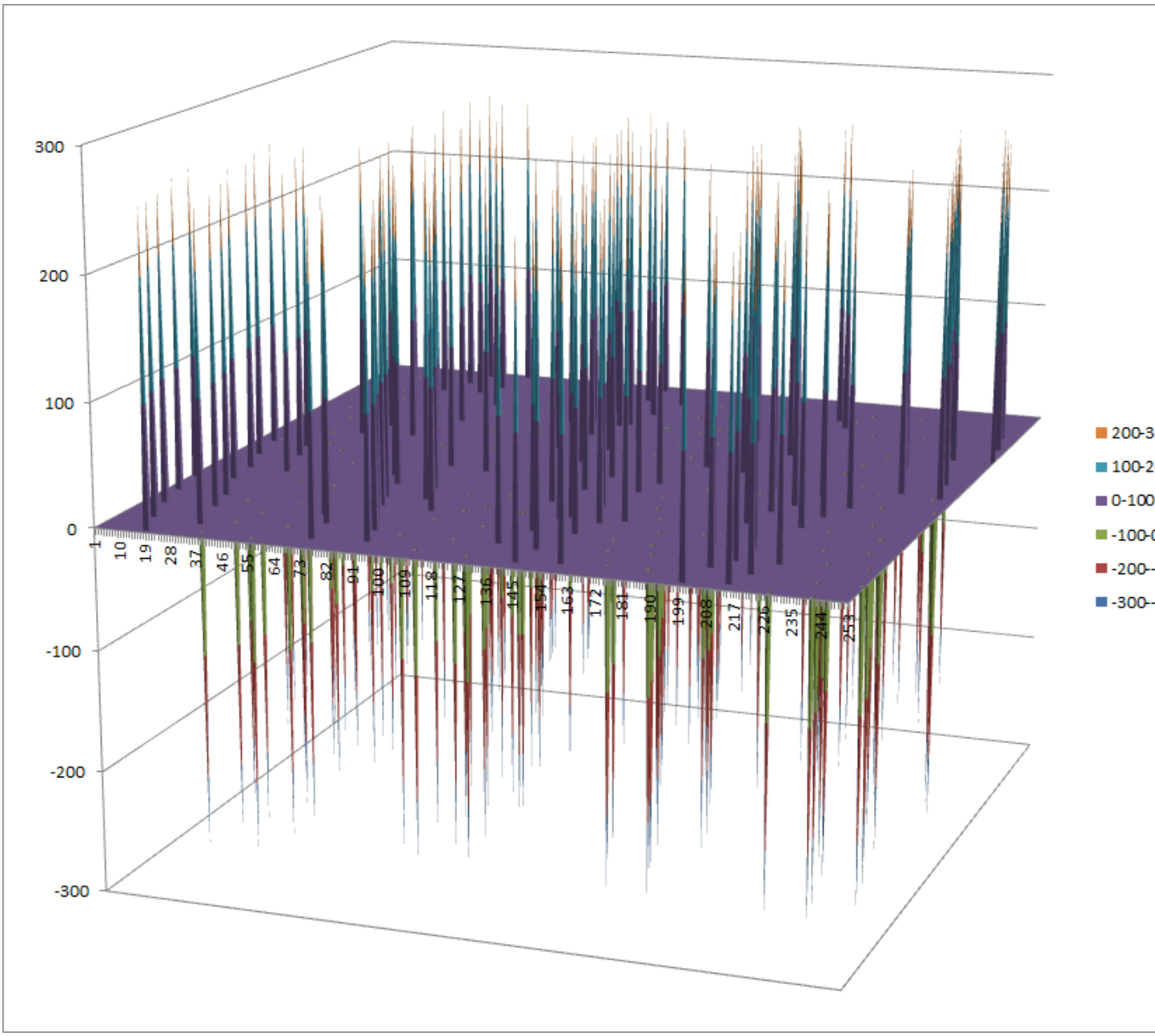
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 4 | 64 |

There are 65025 linear structures

It has no fixed points. It has no negated fixed points

### 7.8.8 xtime

#### Representations

Polynomial representation in ANF

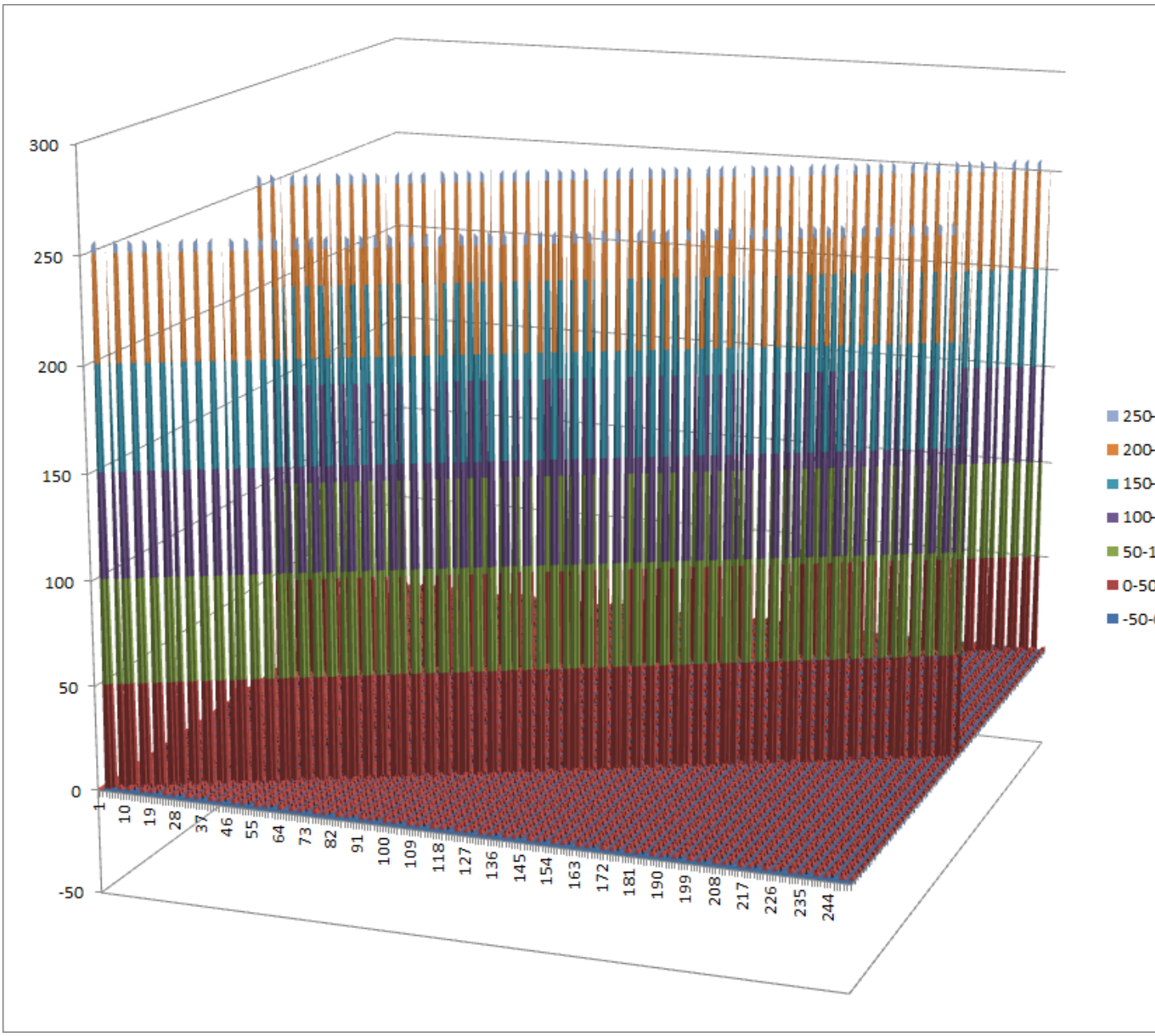
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 51 | 1 |
| 79 | 1 |
| 85 | 1 |
| 92 | 1 |
| 100 | 1 |

There are no linear structures

It has 1 fixed point: (0,0,0,0,0,0,0,0)

It has 1 negated fixed point: (0,1,0,1,0,1,0,1)

## 7.9 Safer+

### 7.9.1 Description

*Safer+* (Massey et al., 1998) was submitted as a candidate for the Advanced Encryption Standard and has a block size of 128 bits. The cipher was not selected as a finalist. Bluetooth uses custom algorithms based on SAFER+ for key derivation (called E21 and E22) and authentication as message authentication codes (called E1). Encryption in Bluetooth does not use SAFER+. It has one 8x8 S-box called expf and other 9x8 S-box called logf.

### 7.9.2 Summary

| S-box | size | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| expf | 8x8 | 82 | 0 | 6 | 3 | 256 | 711424 | 0.1291503906 | 0.5 |
| logf | 9x8 | 164 | 0 | 6 | 3 | 512 | 4520960 | 0.1291503906 | 0.5 |

### 7.9.3 expf

#### Representations

Polynomial representation in ANF

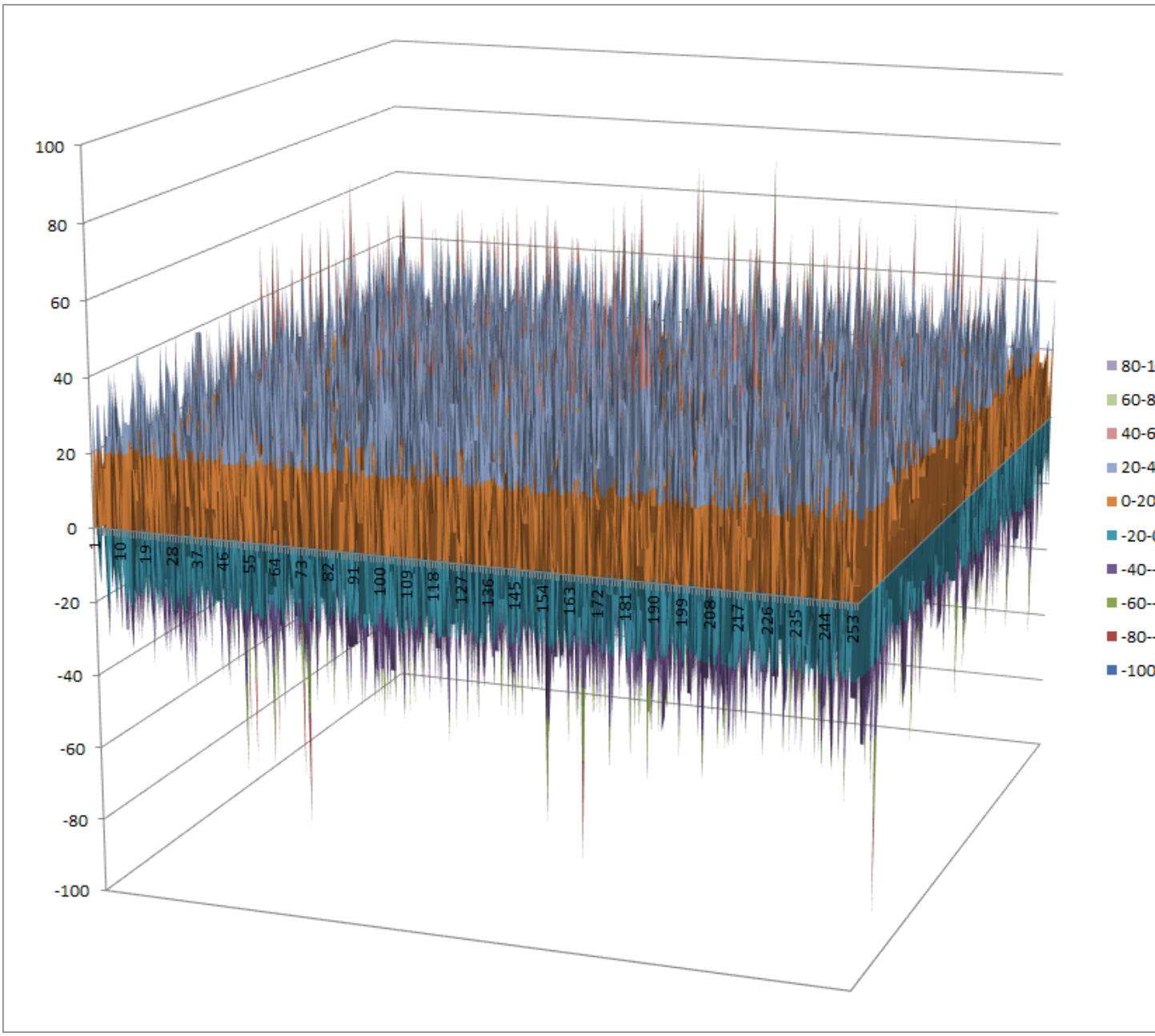
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 3 |
| 2 | 1 |
| 41 | 1 |
| 42 | 1 |
| 168 | 1 |

There are 3 linear structures:

```
([1 0 0 0 0 0 0 0],[0 0 0 0 0 0 0 1])
([1 0 0 0 0 0 0 0],[0 0 0 0 0 0 1 0])
([1 0 0 0 0 0 0 0],[0 0 0 0 0 0 1 1])
```

It has 3 fixed points: (0,0,0,1,1,0,1,1), (0,1,0,1,0,1,1,1), (0,1,0,1,1,1,0,0)

It has 4 negated fixed points: (0,0,0,1,0,1,0,0), (0,0,1,0,1,1,0,0), (0,1,0,1,1,1,1,1), (1,0,1,0,1,1,1,1)

### 7.9.4 logf

#### Representations

Polynomial representation in ANF

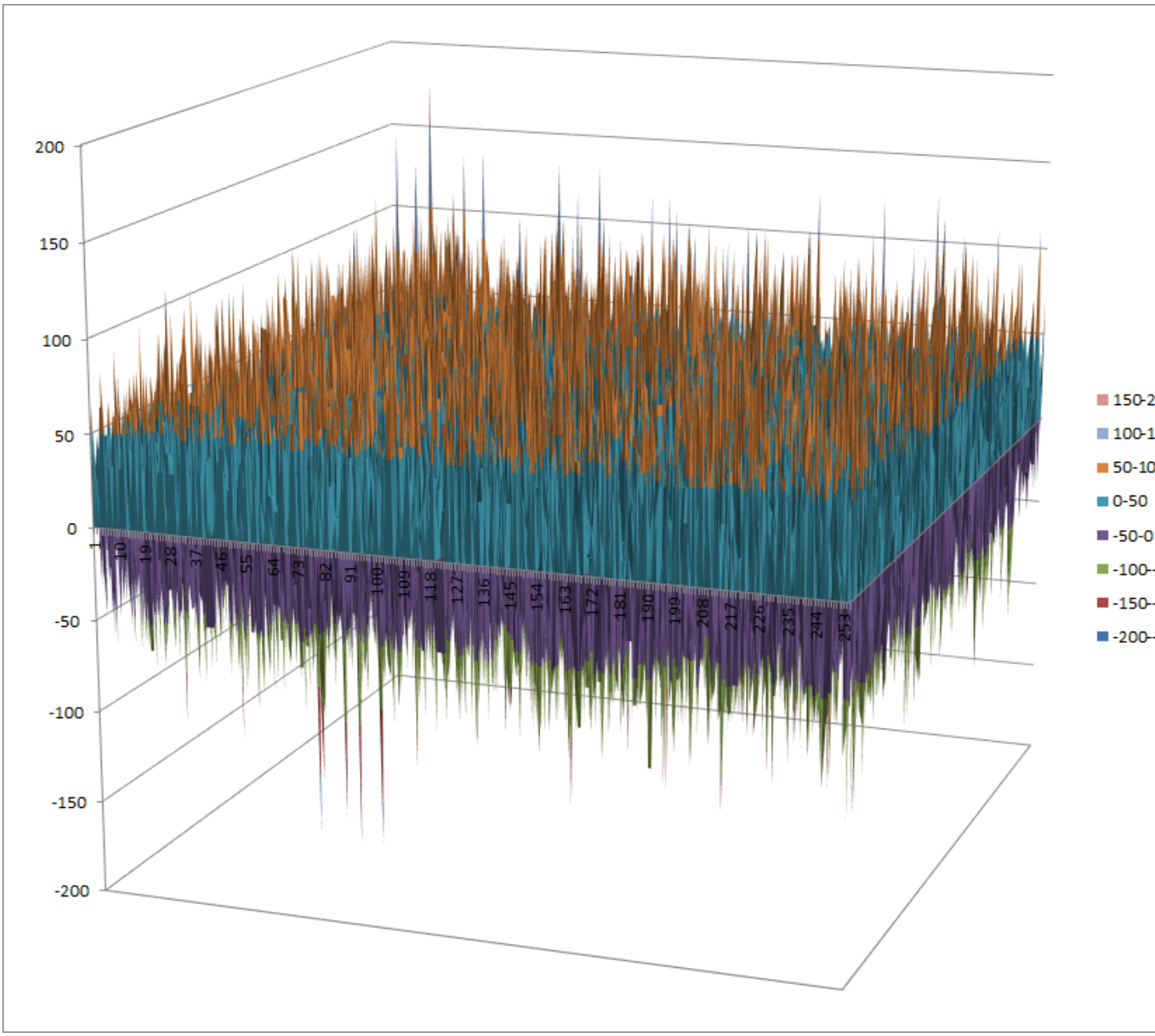
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (first 256x256 values except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There is 255 linear structures:

Linear structures

## 7.10 Serpent

### 7.10.1 Description

*Serpent* is a symmetric key block cipher that was a finalist in the Advanced Encryption Standard (AES) contest, where it was ranked second to Rijndael. Serpent was designed by Ross Anderson, Eli Biham, and Lars Knudsen. It has eight 4x4 S-boxes: S0, S1, S2, S3, S4, S5, S6, S7.

### 7.10.2 Summary

| S-box | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S0 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S1 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S2 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S3 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S4 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S5 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S6 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S7 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |

### 7.10.3 S0

#### Representations

Polynomial representation in ANF

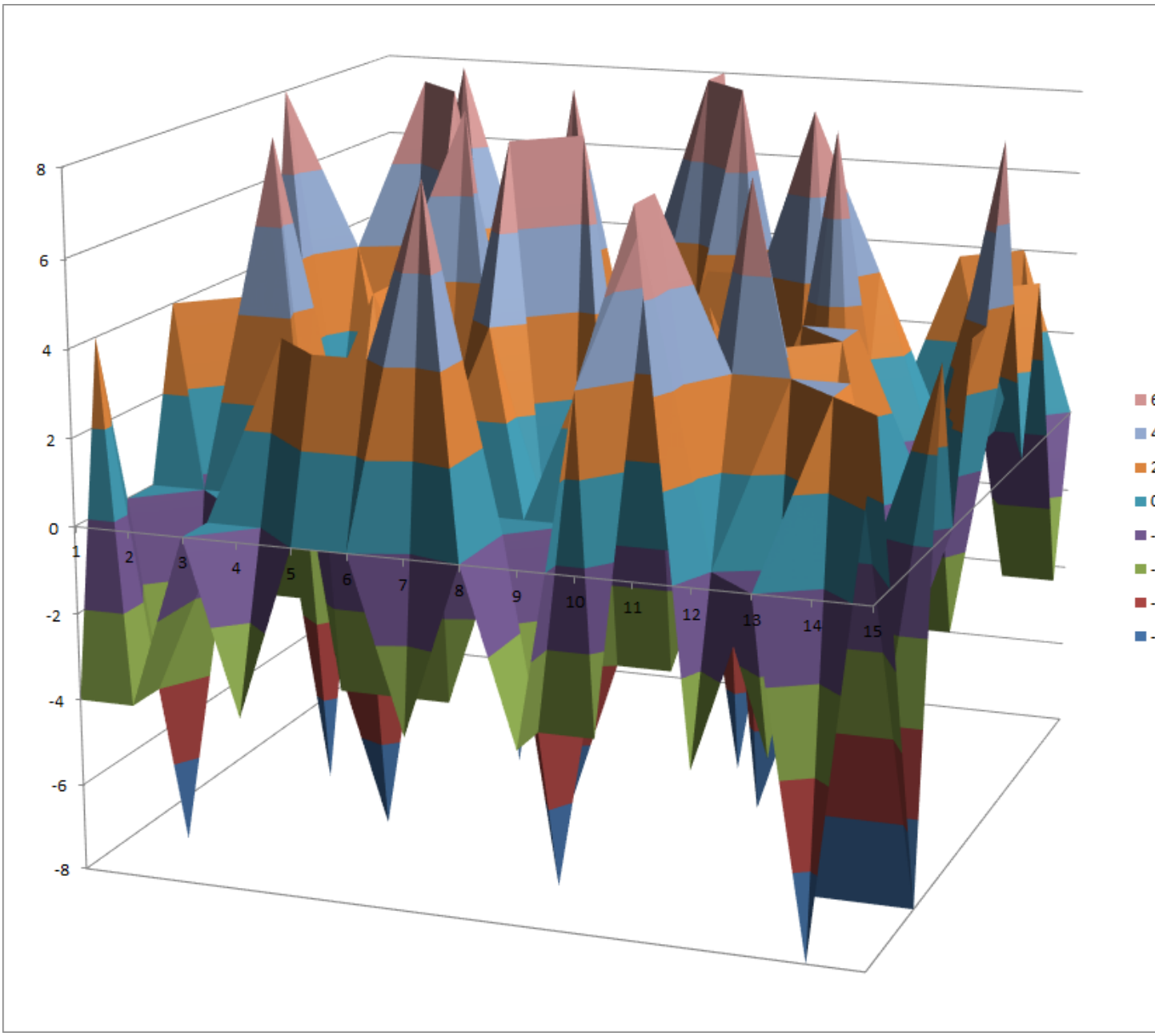
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 2 |
| 5 | 1 |
| 7 | 1 |

There are 9 linear structures:

```
([0 0 1 0],[1 0 0 0])
([0 1 0 0],[0 0 1 1])
([0 1 0 0],[1 0 0 0])
([0 1 0 0],[1 0 1 1])
([0 1 1 0],[1 0 0 0])
([1 0 0 1],[0 0 1 1])
([1 0 1 1],[1 0 1 1])
([1 1 0 1],[0 0 1 1])
([1 1 1 1],[1 0 1 1])
```

It has no fixed points. It has no negated fixed points

### 7.10.4 S1

#### Representations

Polynomial representation in ANF

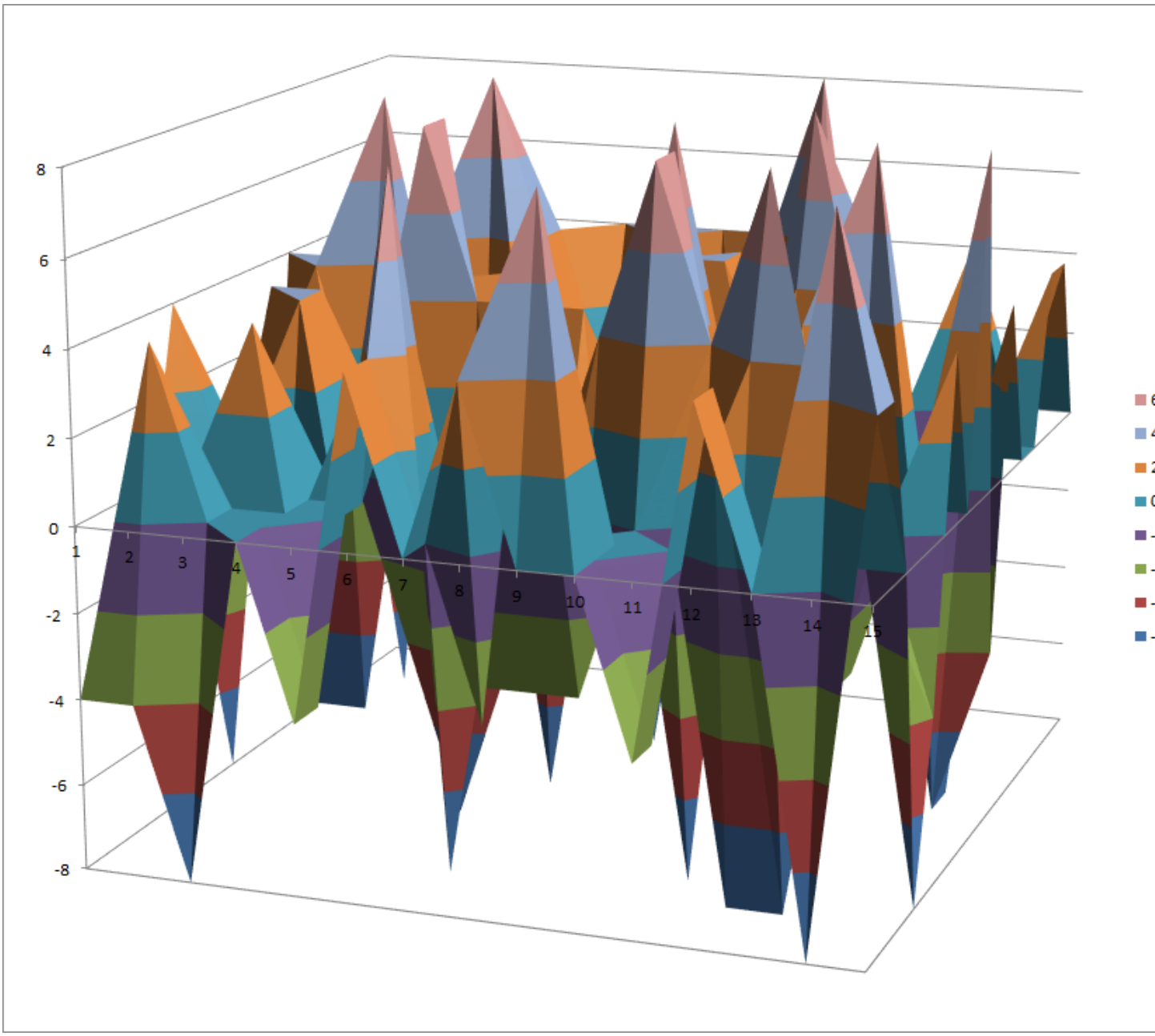
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 4 | 1 |
| 10 | 1 |

There are 9 linear structures:

```
([0 0 1 1],[1 1 1 0])
([0 1 0 0],[0 1 0 0])
([0 1 0 0],[1 0 1 0])
([0 1 0 0],[1 1 1 0])
([0 1 1 1],[1 1 1 0])
([1 0 0 0],[0 1 0 0])
([1 0 1 1],[1 0 1 0])
([1 1 0 0],[0 1 0 0])
([1 1 1 1],[1 0 1 0])
```

It has 1 fixed point: (0,0,1,0)

It has 1 negated fixed point: (0,0,0,0)

### 7.10.5 S2

#### Representations

Polynomial representation in ANF

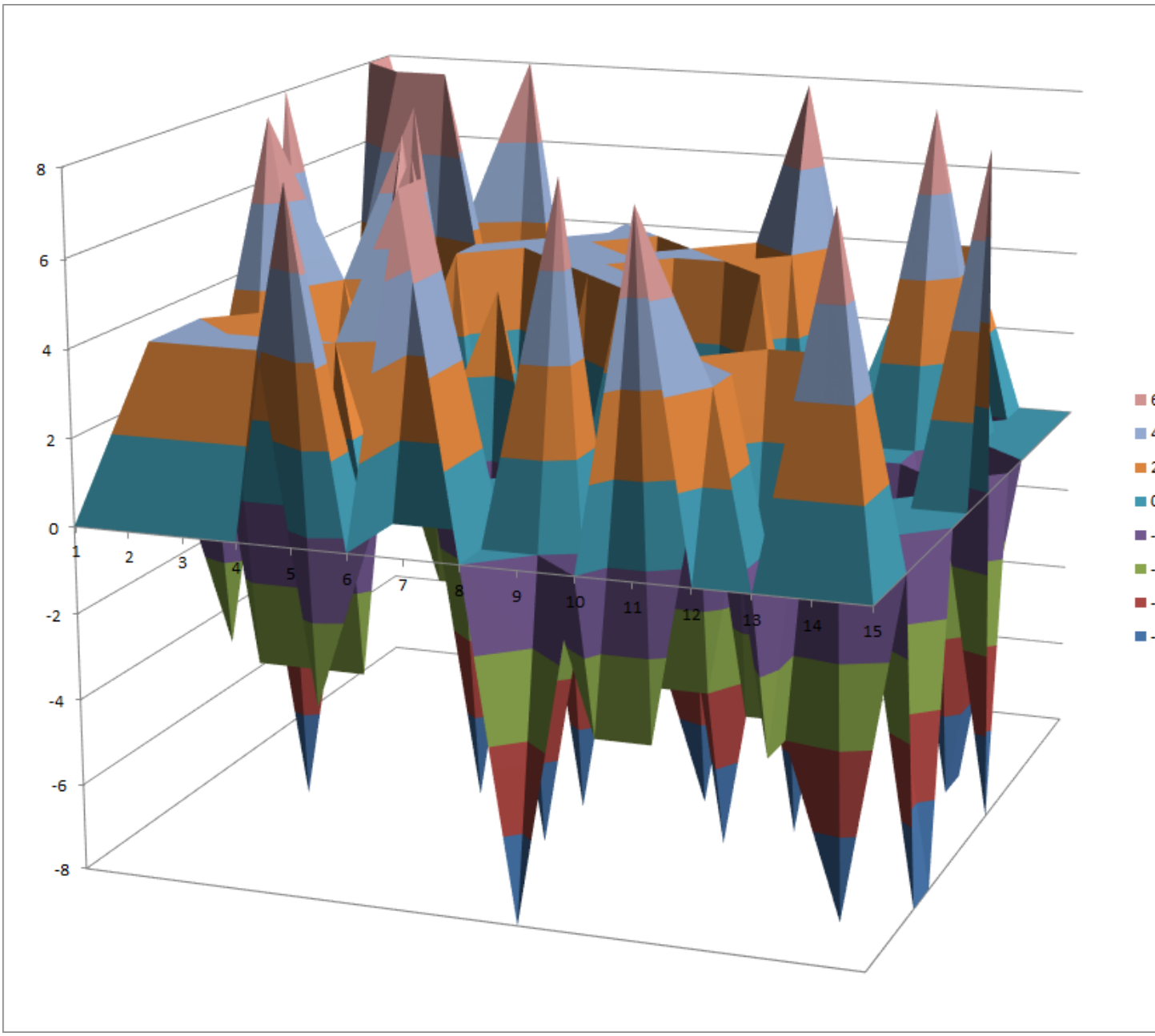
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 3 | 1 |
| 13 | 1 |

There are 9 linear structures:

```
([0 0 1 0],[0 0 0 1])
([0 0 1 0],[1 1 1 0])
([0 0 1 0],[1 1 1 1])
([0 1 0 0],[1 1 1 0])
([0 1 1 0],[1 1 1 0])
([1 0 0 0],[0 0 0 1])
([1 0 1 0],[0 0 0 1])
([1 1 0 0],[1 1 1 1])
([1 1 1 0],[1 1 1 1])
```

It has no fixed points

It has 1 negated fixed point: (1,0,1,1)

### 7.10.6 S3

#### Representations

Polynomial representation in ANF

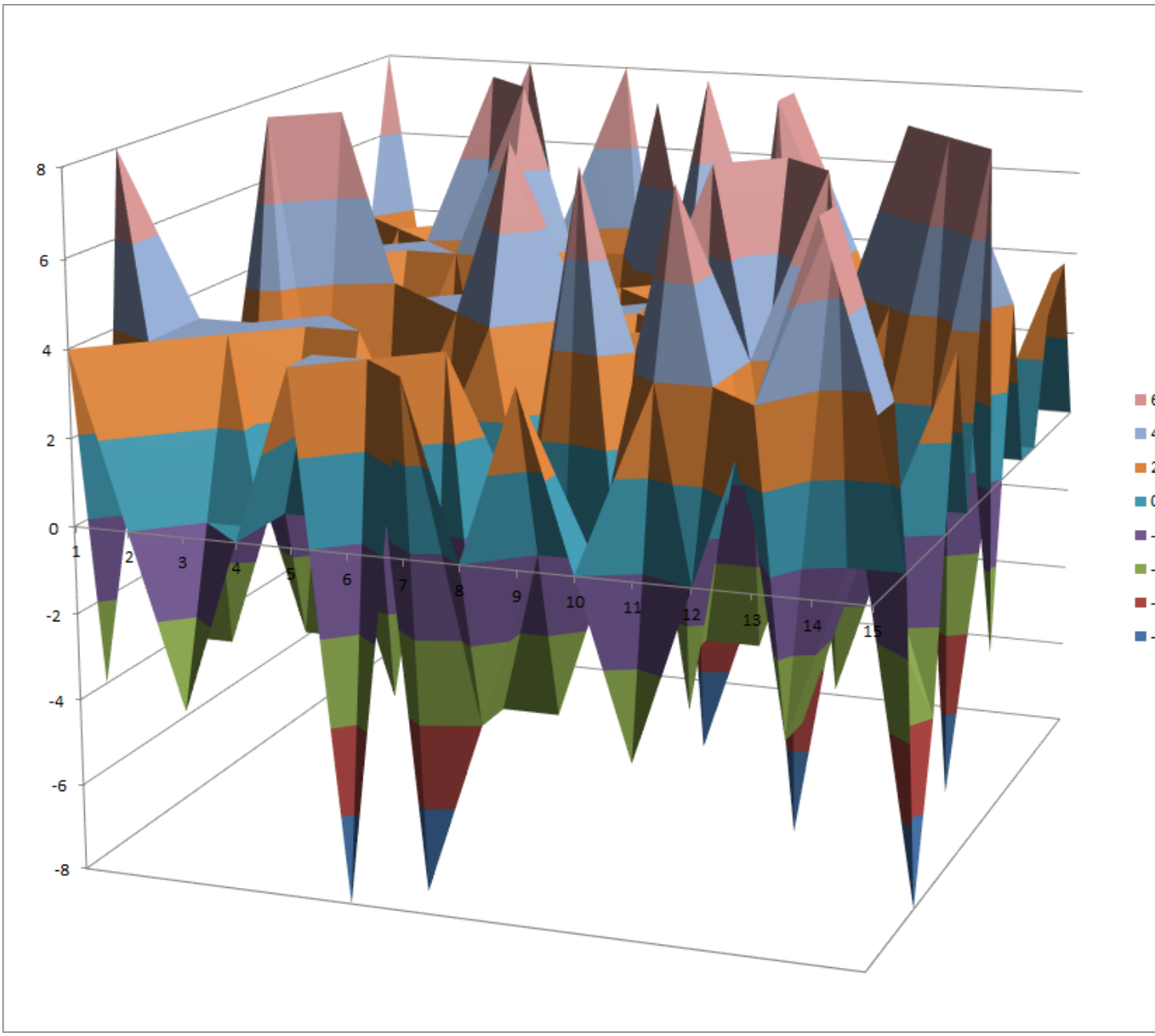
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 4 | 1 |
| 5 | 2 |

There are 3 linear structures:

```
([0 0 1 1],[1 1 1 0])
([0 1 0 0],[1 1 1 0])
([0 1 1 1],[1 1 1 0])
```

It has 1 fixed point: (0,0,0,0)

It has 1 negated fixed point: (1,0,1,1)

### 7.10.7 S4

#### Representations

Polynomial representation in ANF

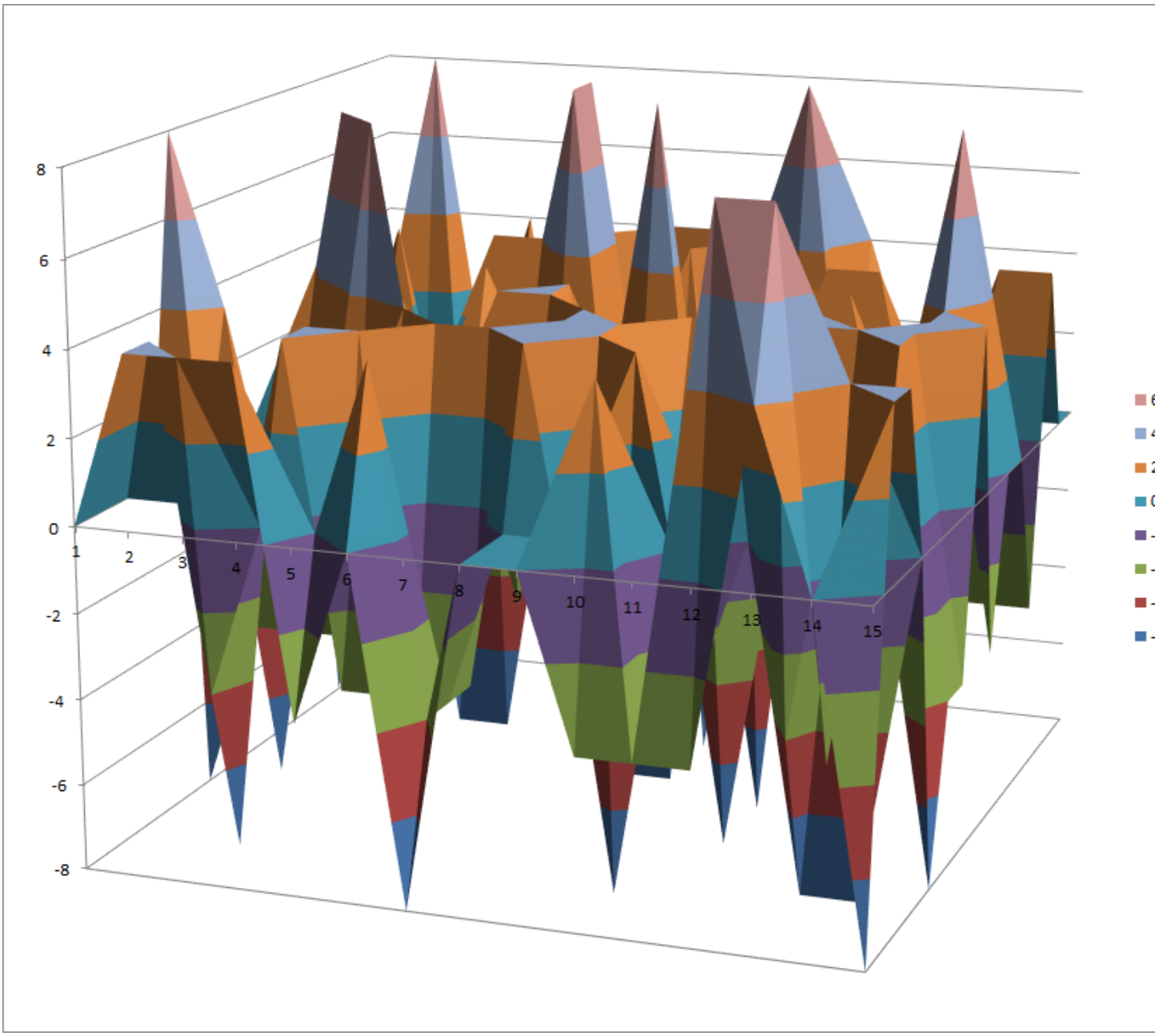
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 13 | 1 |

There are 3 linear structures:

```
([0 1 0 0],[0 0 0 1])
([1 0 1 1],[0 0 0 1])
([1 1 1 1],[0 0 0 1])
```

It has 1 fixed point: (0,0,1,1)

It has no negated fixed points

### 7.10.8 S5

#### Representations

Polynomial representation in ANF

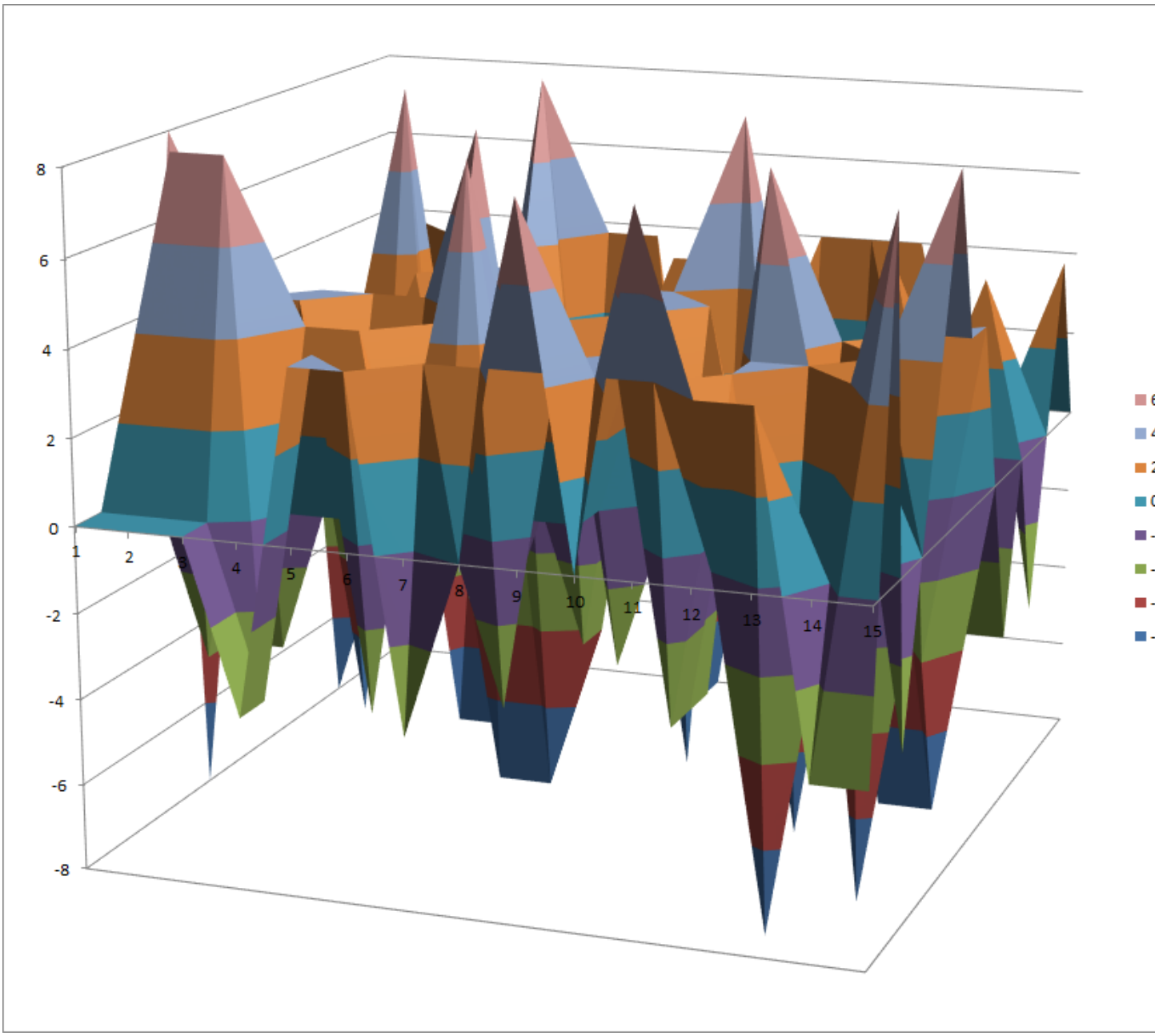
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 14 | 1 |

There are 3 linear structures:

```
([0 1 0 0],[0 0 0 1])
([1 0 1 1],[0 0 0 1])
([1 1 1 1],[0 0 0 1])
```

It has 1 fixed point: (0,0,1,0)

It has 3 negated fixed points: (0,0,0,0), (0,1,0,1), (0,1,1,0)

### 7.10.9 S6

#### Representations

Polynomial representation in ANF

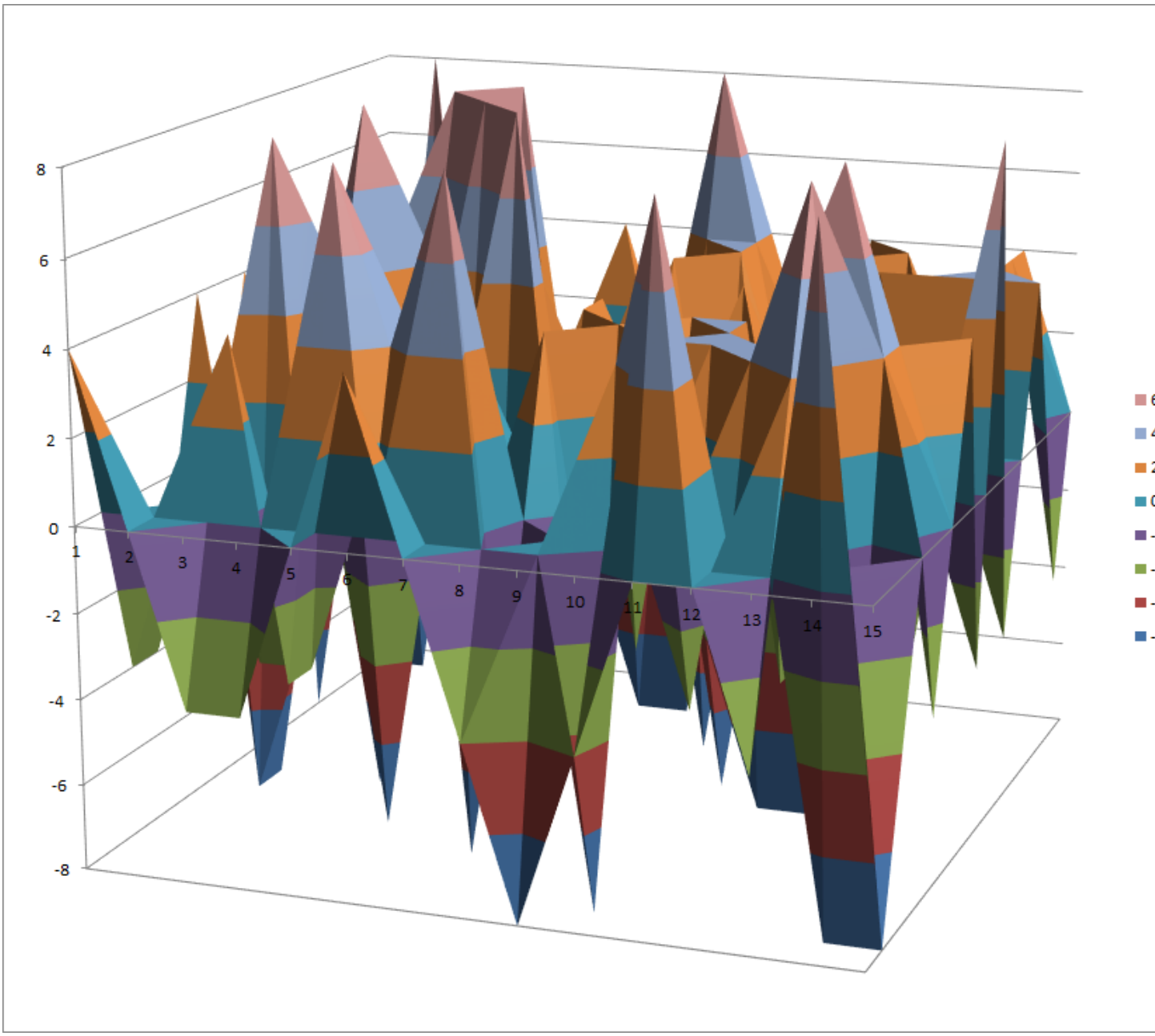
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 4 | 1 |
| 10 | 1 |

There are 9 linear structures:

```
([0 0 1 0],[0 0 1 0])
([0 1 0 0],[0 0 1 0])
([0 1 1 0],[0 0 1 0])
([0 1 1 0],[0 1 0 1])
([0 1 1 0],[0 1 1 1])
([1 0 0 1],[0 1 0 1])
([1 0 1 1],[0 1 1 1])
([1 1 0 1],[0 1 1 1])
([1 1 1 1],[0 1 0 1])
```

It has 1 fixed point: (0,1,1,0)

It has 1 negated fixed point: (1,1,1,1)

### 7.10.10 S7

#### Representations

Polynomial representation in ANF

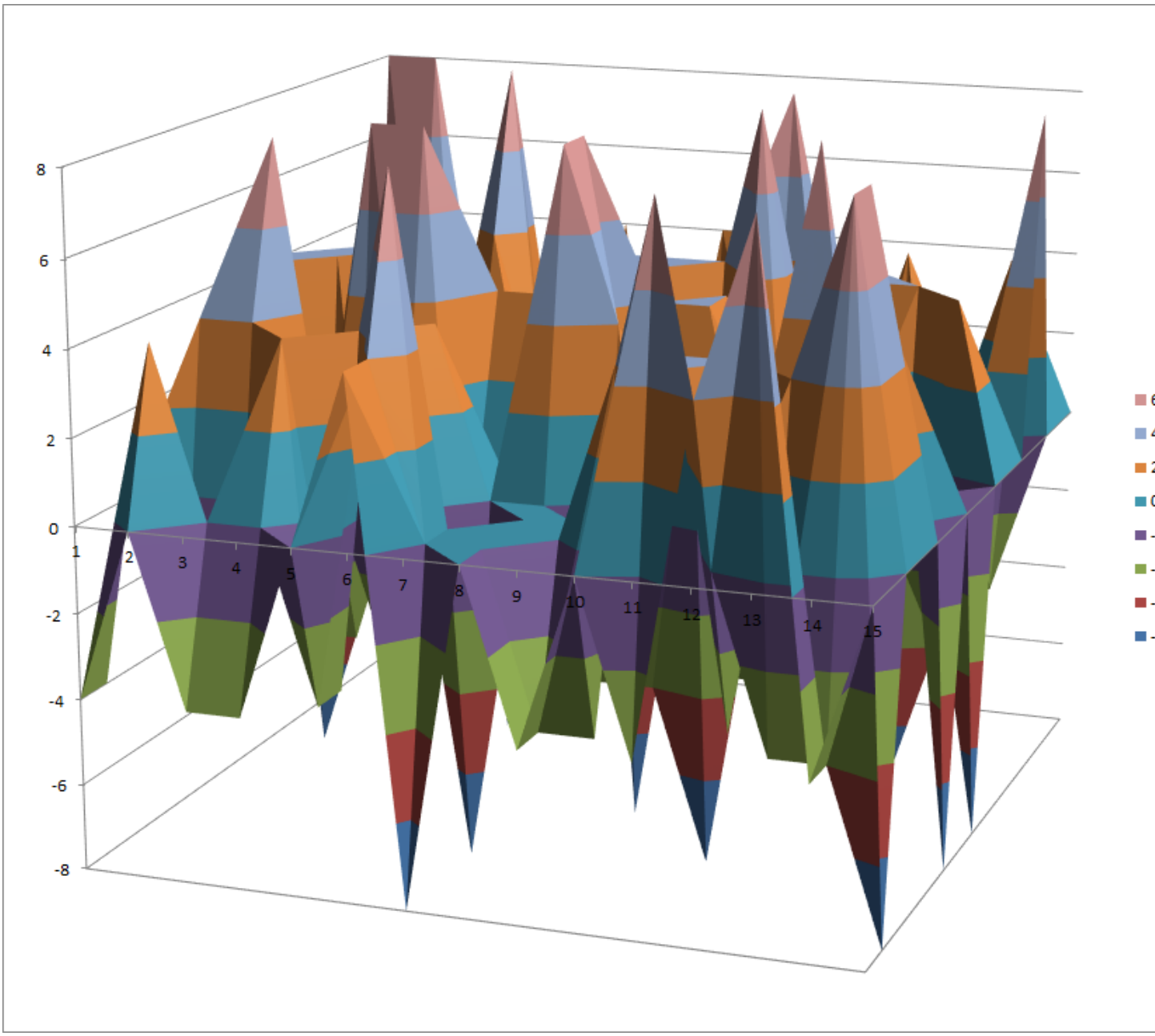
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 3 | 1 |
| 4 | 1 |
| 9 | 1 |

There are 3 linear structures:

```
([0 0 0 1],[1 1 1 1])
([1 0 1 0],[1 1 1 1])
([1 0 1 1],[1 1 1 1])
```

It has no fixed points

It has 1 negated fixed point: (1,0,0,0)

# 8 Analysis of CRYPTEC project cryptographic algorithms

CRYPTREC is the Cryptography Research and Evaluation Committees set up by the Japanese Government to evaluate and recommend cryptographic techniques for government and industrial use. It is comparable in many respects to the European Union's NESSIE project and to the Advanced Encryption Standard process run by NIST in the U.S.. In this chapter, certain cryptographic algorithms from CRYPTEC project candidates are analysed.

Below you can find a legend describing the cryptographic criteria used in this chapter:

| | |
|---|---|
| **NL** | Nonlinearity |
| **NL2** | 2-nd order nonlinearity |
| **LD** | Linearity distance |
| **DEG** | Algebraic degree |
| **AI** | Algebraic immunity |
| **MAXAC** | Absolute indicator |
| $\sigma$ | Sum-of-squares indicator |
| **LP** | Linear potential |
| **DP** | Differential Potential |

**Hyperlinks to representations**

Open the hyperlinks to representations below in a new browser window or in a new tab.

* *Other useful information in cryptanalysis*

## 8.1 CIPHERUNICORN-E

### 8.1.1 Description

*CIPHERUNICORN-E* is a block cipher created by NEC in 1998. It was among the cryptographic techniques recommended for Japanese government use by CRYPTREC in 2003, however, has been dropped to "candidate" by CRYPTREC revision in 2013. It has four 8x8 S-boxes: S0,S1,S2,S3

### 8.1.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| S0 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| S1 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| S2 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| S3 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |

### 8.1.3 S0

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$: Trace representation

Polynomial representation in ANF

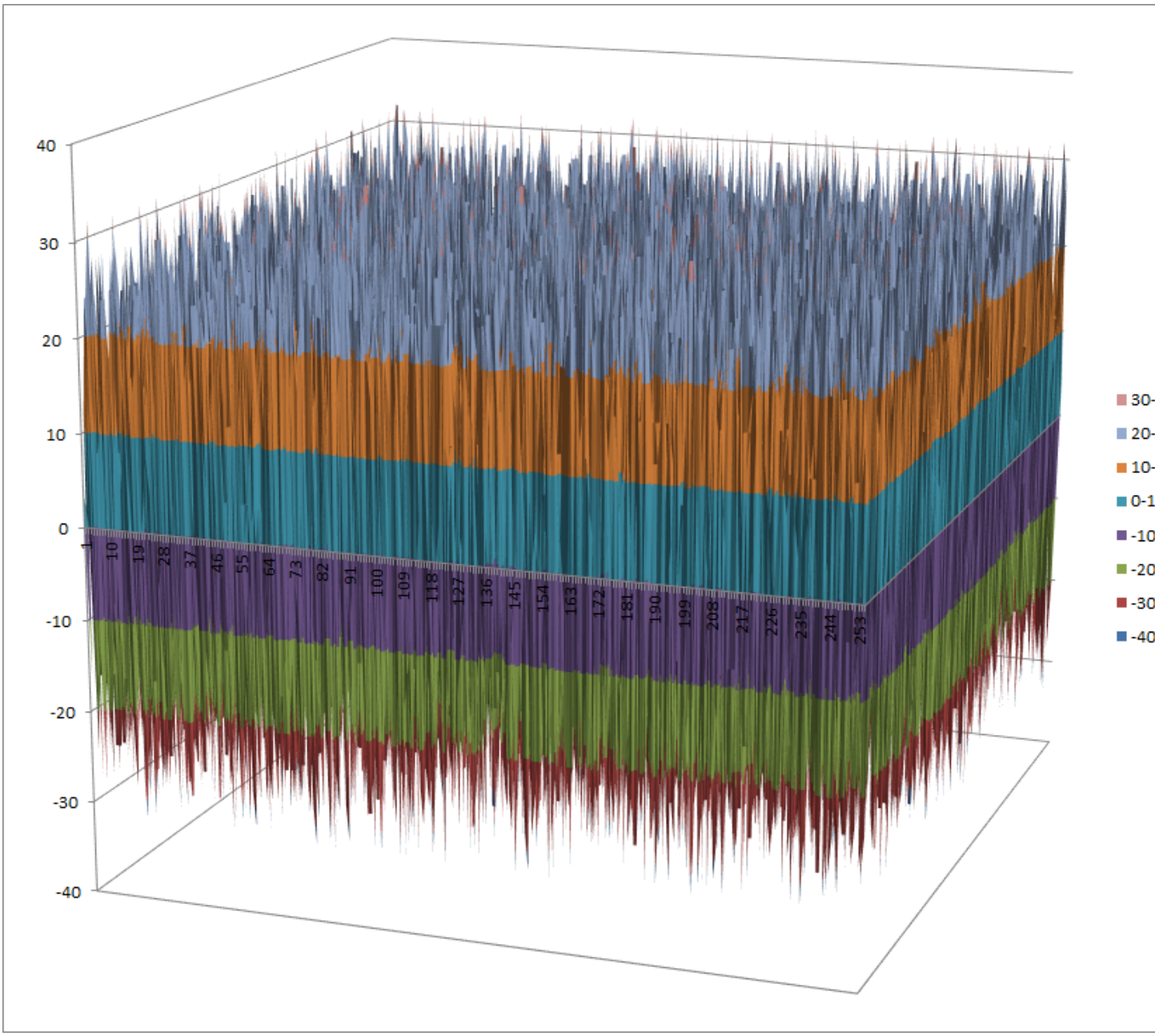
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 45 | 1 |
| 209 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points.

### 8.1.4 S1

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^2 + 1$: Trace representation

Polynomial representation in ANF

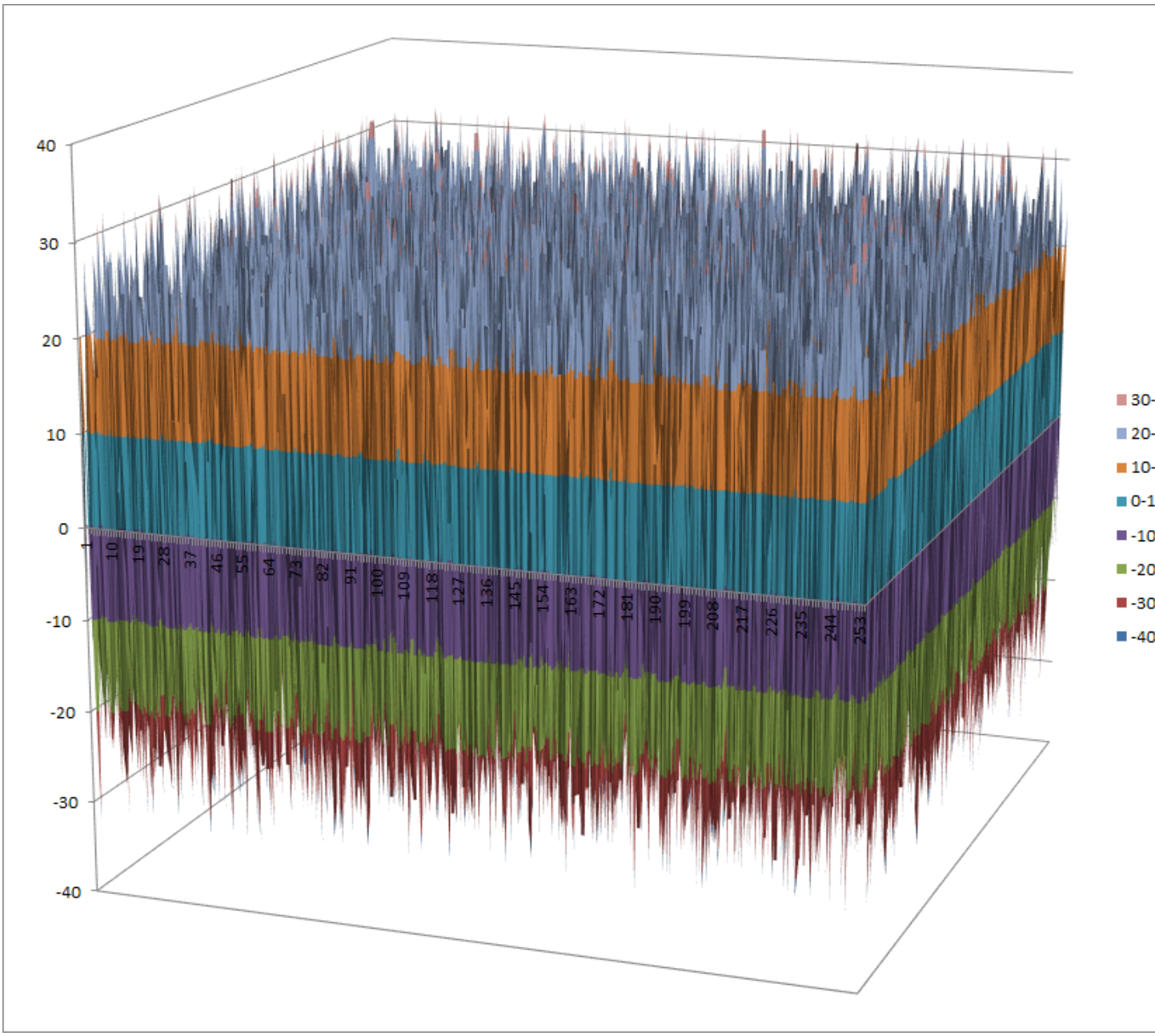
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 49 | 1 |
| 205 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points.

### 8.1.5 S2

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^3 + x^2 + 1$: Trace representation

Polynomial representation in ANF

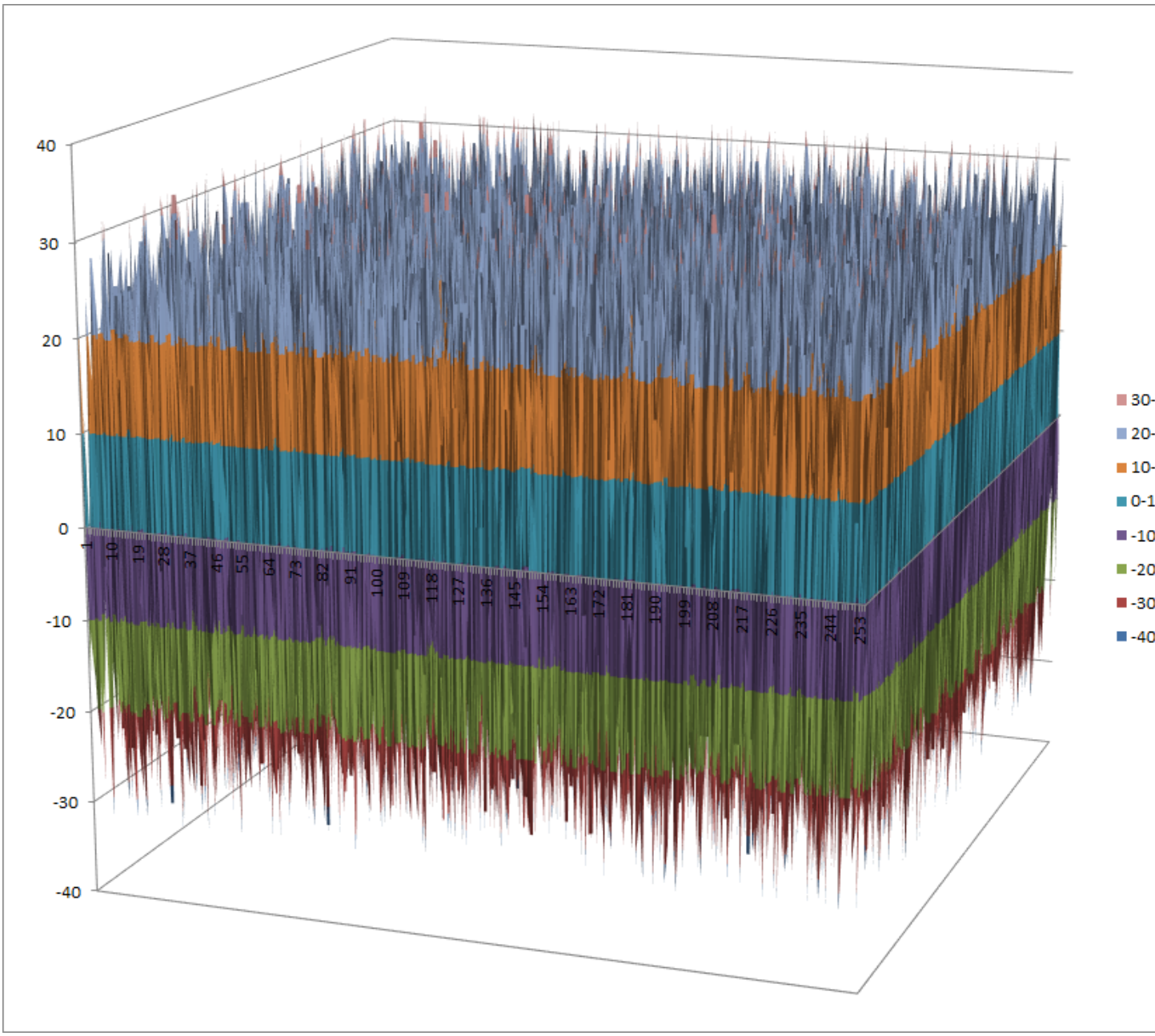
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 4 | 1 |
| 6 | 1 |
| 33 | 1 |
| 73 | 1 |
| 140 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points.

### 8.1.6 S3

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^4 + 1$: Trace representation

Polynomial representation in ANF

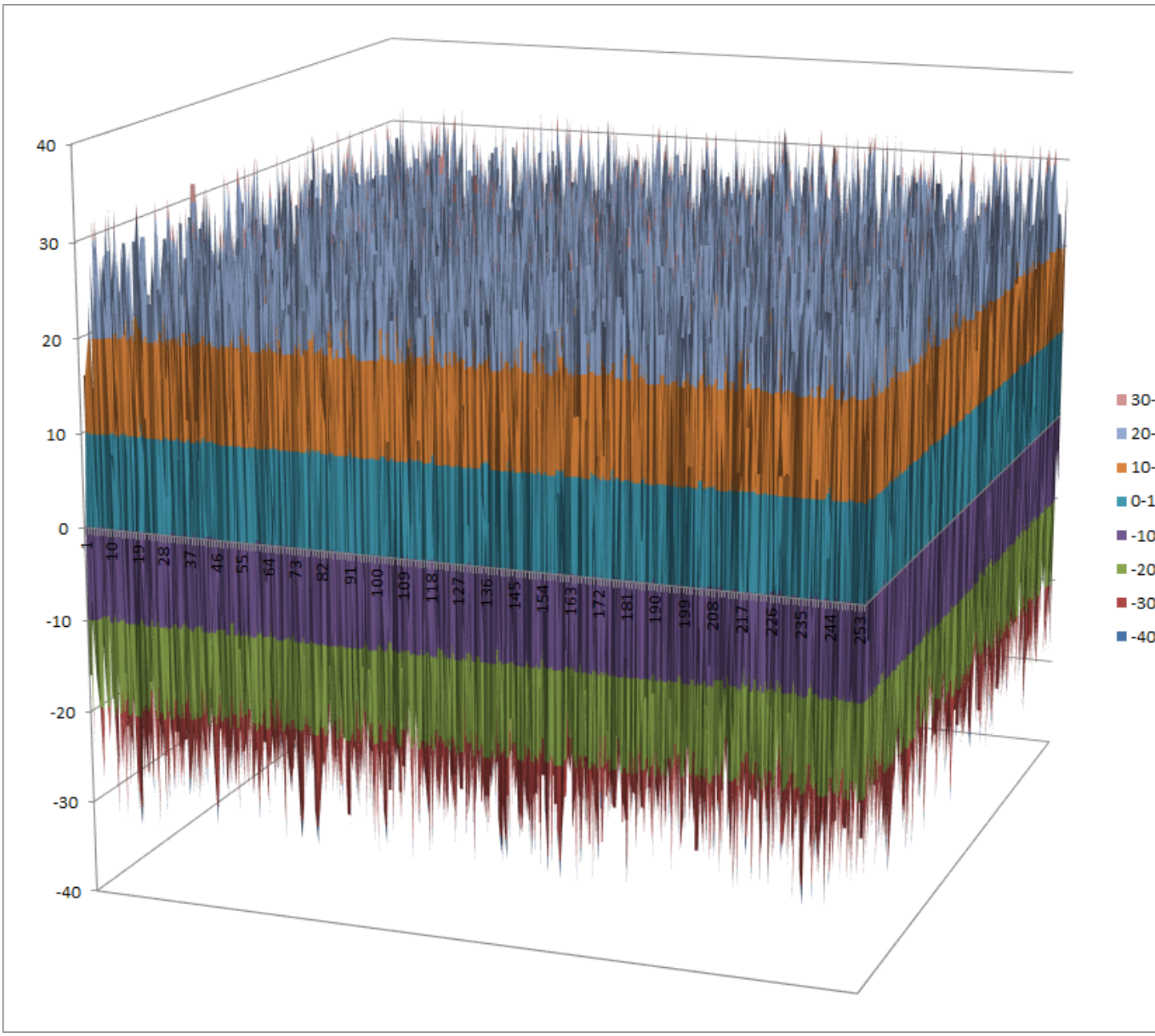
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 3 | 2 |
| 8 | 1 |
| 21 | 1 |
| 221 | 1 |

There are no linear structures

It has no fixed points.

It has 2 negated fixed points: (0,0,1,0,0,0,1,1), (0,1,1,1,1,1,1,1)

## 8.2 CLEFIA

### 8.2.1 Description

*CLEFIA* is a proprietary block cipher algorithm, developed by Sony. It is intended to be used in DRM systems. It is among the cryptographic techniques recommended candidate for Japanese government use by CRYPTREC revision in 2013. It has two $8 \times 8$ S-boxes: $S_0, S_1$

### 8.2.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| S0 | 100 | 40 | 6 | 4 | 96 | 269056 | 0.0478515625 | 0.0390625 |
| S1 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |

### 8.2.3 S0

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$: Trace representation

Polynomial representation in ANF

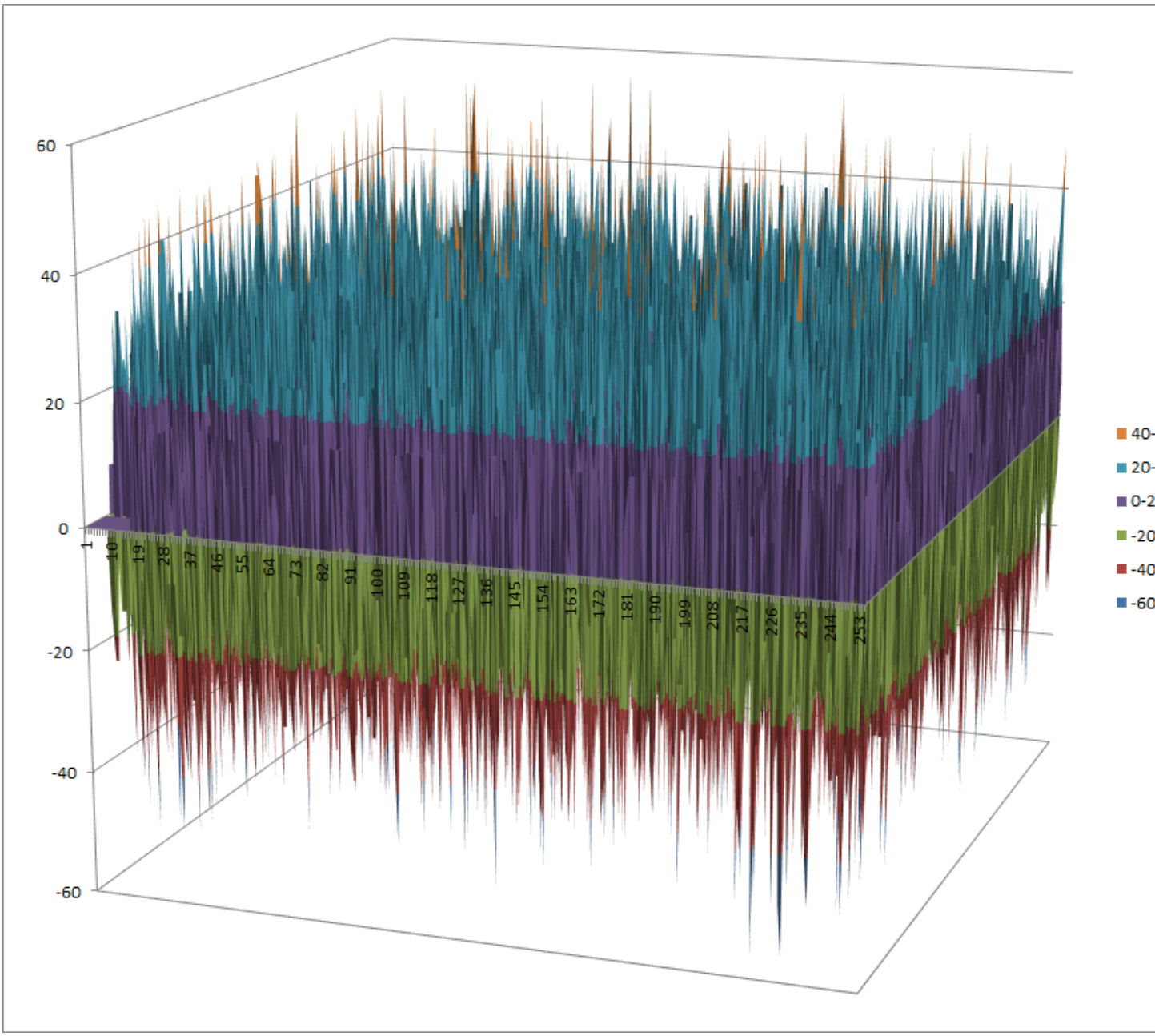
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 4 | 1 |
| 5 | 2 |
| 17 | 1 |
| 109 | 1 |
| 116 | 1 |

There are no linear structures

It has no fixed points.

It has 2 negated fixed points: (0,1,0,0,0,0,0,1), (1,1,1,1,1,1,0,1)

## Construction

$S_0$ is generated by combining four 4-bit S-boxes $SS_0, SS_1, SS_2$ and $SS_3$ in the following way:

Step 1. $\mathbf{t_0} = SS_0(\mathbf{x_0}),\ \mathbf{t_1} = SS_1(\mathbf{x_1})$ where $\mathbf{x} = \mathbf{x_0}|\mathbf{x_1}, \mathbf{x_i} \in \mathrm{V}_4$

Step 2. $\mathbf{u_0} = \mathbf{t_0} + 0x2 \cdot \mathbf{t_1},\ \mathbf{u_1} = 0x2 \cdot \mathbf{t_0} + \mathbf{t_1}$

Step 3. $\mathbf{y_0} = SS_2(\mathbf{u_0}),\ \mathbf{y_1} = SS_3(\mathbf{u_1})$ where $\mathbf{y} = \mathbf{y_0}|\mathbf{y_1}, \mathbf{y_i} \in \mathrm{V}_4$

Tables of CLEFIA S-boxes $SS_i (0 \leq i \leq 3)$:

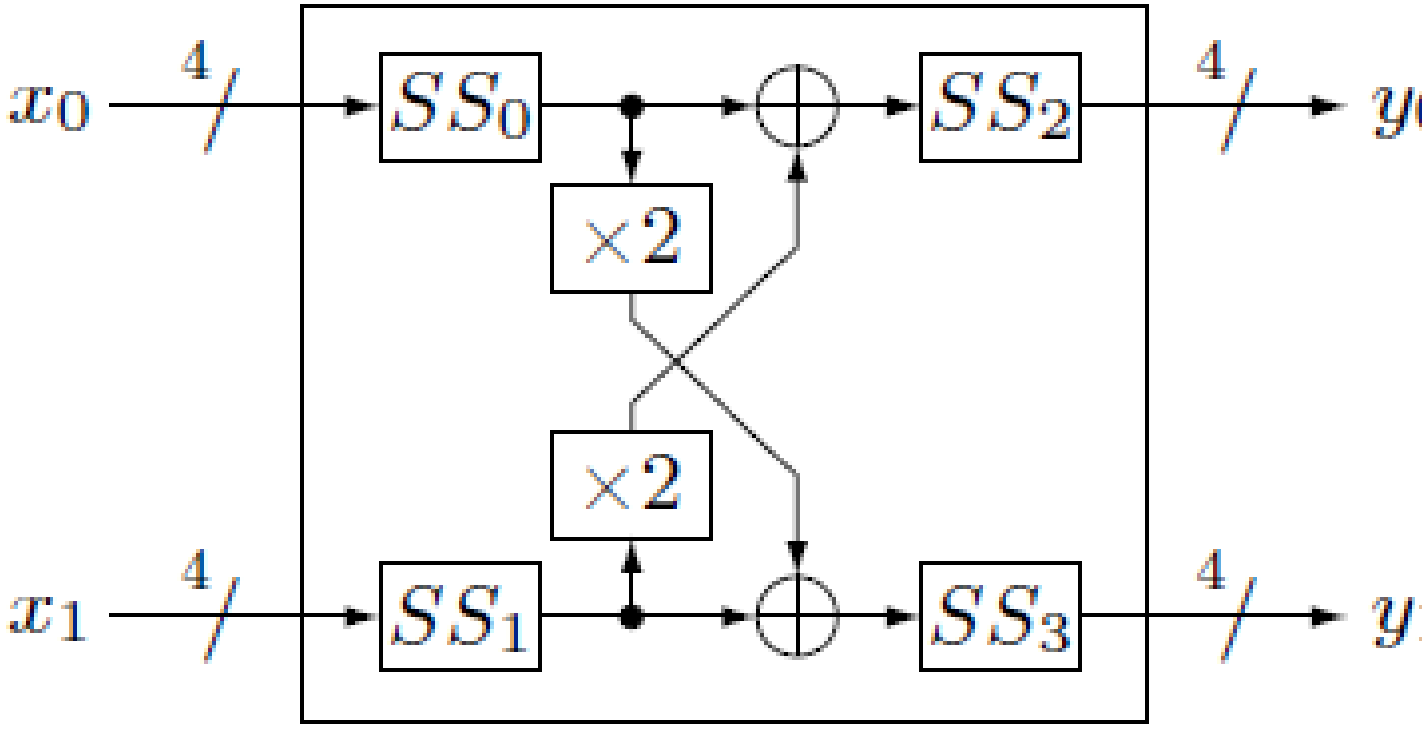
| x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SS0(x) | e | 6 | c | a | 8 | 7 | 2 | f | b | 1 | 4 | 0 | 5 | 9 | d | 3 |
| SS1(x) | 6 | 4 | 0 | d | 2 | b | a | 3 | 9 | c | e | f | 8 | 7 | 5 | 1 |
| SS2(x) | b | 8 | 5 | e | a | 6 | 4 | c | f | 7 | 2 | 3 | 1 | 0 | d | 9 |
| SS3(x) | a | 2 | 6 | d | 3 | 4 | 5 | e | 0 | 7 | 8 | 9 | b | f | c | 1 |

The multiplication in $0x2 \cdot \mathbf{t_i}$ is performed in $\mathrm{GF}(2^4)$ defined by the lexicographically first primitive polynomial $x^4 + x + 1$. Here we provide the table of multiplication of $0x2$ with an element modulo $x^4 + x + 1$. The entries in the Table are represented in hexadecimal notation for compactness. The column indices represent the element to be multplied by $0x2$ modulo $x^4 + x + 1$, and the product is the corresponding entry in the column.

Table of the multiplication $0x2 \cdot \mathbf{x}$:

| $\mathbf{x}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $0x2 \cdot \mathbf{x}$ | 0 | 2 | 4 | 6 | 8 | a | c | e | 3 | 1 | 7 | 5 | b | 9 | f | d |

Next figure shows the construction of $S_0$:




Hence, CLEFIA S0 can be denoted by:

$S_0(\mathbf{x_0}, \mathbf{x_1}) = (SS_2\,(SS_0(\mathbf{x_0}) \oplus Mul2\,(SS_1(\mathbf{x_1}))), SS_3\,(Mul2\,(SS_0(\mathbf{x_0})) \oplus SS_1(\mathbf{x_1})))$

Note that the symbol $\circ$ refers to the composition of functions, $\oplus$ refers to the direct sum of functions and $Mul2(\mathbf{x}) = 0x2 \cdot \mathbf{x}$.

The criteria of several constructions in $S_0$ are summarized in the following tables:

| S-box | $r$ | $NL$ | $lp$ | $dp$ | $AC_{max}$ | $LI$ |
|---|---|---|---|---|---|---|
| $t_0 = SS_0$ | 8 | 4 | 0.25 | 0.25 | 16 | 0 |
| $t_1 = SS_1$ | 8 | 4 | 0.25 | 0.25 | 16 | 0 |
| $SS_2$ | 8 | 4 | 0.25 | 0.25 | 16 | 0 |
| $SS_3$ | 8 | 4 | 0.25 | 0.25 | 16 | 0 |
| $Mul2(\mathbf{x}) = 0x2 \cdot \mathbf{x}$ | 16 | 0 | 1 | 1 | 16 | 0 |
| $0x2 \cdot \mathbf{t_0} = Mul2 \circ SS_0$ | 8 | 4 | 0.25 | 0.25 | 16 | 0 |
| $0x2 \cdot \mathbf{t_1} = Mul2 \circ SS_1$ | 8 | 4 | 0.25 | 0.25 | 16 | 0 |
| $\mathbf{u_0} = SS_0 \oplus (Mul2 \circ SS_1)$ | 64 | 96 | 0.0625 | 0.25 | 256 | 0 |
| $\mathbf{u_1} = (Mul2 \circ SS_0) \oplus SS_1$ | 64 | 96 | 0.0625 | 0.25 | 256 | 0 |
| $\mathbf{y_0} = SS2 \circ \mathbf{u_0}$ | 56 | 100 | 0.0478515625 | 0.15625 | 96 | 40 |
| $\mathbf{y_1} = SS3 \circ \mathbf{u_1}$ | 56 | 100 | 0.0478515625 | 0.15625 | 88 | 42 |
| $S_0 = \mathbf{y} = (\mathbf{y_0}, \mathbf{y_1})$ | 56 | 100 | 0.0478515625 | 0.0390625 | 96 | 40 |

| S-box | $deg$ | $AI$ | $\sigma$ | $CI$ |
|---|---|---|---|---|
| $t_0 = SS_0$ | 2 | 2 | 1024 | 0 |
| $t_1 = SS_1$ | 2 | 2 | 1024 | 0 |
| $SS_2$ | 2 | 2 | 1024 | 0 |
| $SS_3$ | 2 | 2 | 1024 | 0 |
| $Mul2(\mathbf{x}) = 0x2 \cdot \mathbf{x}$ | 1 | 1 | 4096 | 0 |
| $0x2 \cdot \mathbf{t_0} = Mul2 \circ SS_0$ | 2 | 2 | 1024 | 0 |
| $0x2 \cdot \mathbf{t_1} = Mul2 \circ SS_1$ | 2 | 2 | 1024 | 0 |
| $\mathbf{u_0} = SS_0 \oplus (Mul2 \circ SS_1)$ | 3 | 3 | 256 | 1 |
| $\mathbf{u_1} = (Mul2 \circ SS_0) \oplus SS_1$ | 3 | 3 | 256 | 1 |
| $\mathbf{y_0} = Mul2 \circ \mathbf{u_0}$ | 6 | 4 | 269056 | 1 |
| $\mathbf{y_1} = Mul2 \circ \mathbf{u_1}$ | 6 | 4 | 246784 | 1 |
| $S_0 = \mathbf{y} = (\mathbf{y_0}, \mathbf{y_1})$ | 6 | 4 | 269056 | 0 |

You can find a program which calculates the Truth Tables of these constructions in chapter “Operations and constructions over Vector Boolean Functions”, section “Addition of coordinate functions”.

### Mul2

Let $Mul2(\mathbf{x}) = 0x2 \cdot \mathbf{x}$ the multiplication in $\mathrm{GF}(2^4)$ defined by the primitive polynomial $x^4 + x + 1$ as in CLEFIA cipher.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 15 | 1 |

There are 225 linear structures

Linear Structures

It has 1 fixed point: (0,0,0,0)

It has 1 negated fixed point: (0,1,0,1)

$0x2 \cdot \mathbf{t_1}$

The operation $0x2 \cdot \mathbf{t_1}$ in Step 2 can be interpreted as the composition of $Mul2$ and $SS_1$.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 10 | 1 |

There are 3 linear structures:

```
([0 0 1 0],[0 1 0 1])
([0 1 0 0],[0 1 0 1])
([0 1 1 0],[0 1 0 1])
```

It has 2 fixed points: (0,1,0,0), (0,1,0,1)

It has 1 negated fixed point: (1,1,0,0)

### $0x2 \cdot \mathbf{t_0}$

The operation $0x2 \cdot \mathbf{t_0}$ in Step 2 can be interpreted as the composition of $Mul2$ and $SS_0$.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 16 | 1 |

There are 3 linear structures:

```
([0 0 1 1],[1 0 0 1])
([1 0 0 1],[1 0 0 1])
([1 0 1 0],[1 0 0 1])
```

It has no fixed points.

It has 1 negated fixed point: (0,0,0,0)

### $\mathbf{u_0}$

The operation $\mathbf{u_0} = \mathbf{t_0} \oplus 0x2 \cdot \mathbf{t_1}$ in Step 2 can be interpreted as the direct sum of $SS_0$ and $Mul2 \circ SS_1$.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

There are 6 linear structures:

```
([0 0 0 0 0 0 1 0],[0 1 0 1])
([0 0 0 0 0 1 0 0],[0 1 0 1])
([0 0 0 0 0 1 1 0],[0 1 0 1])
([0 0 1 1 0 0 0 0],[1 1 0 0])
```

(continues on next page)

(continued from previous page)

```
([1 0 0 1 0 0 0 0],[1 1 0 0])
([1 0 1 0 0 0 0 0],[1 1 0 0])
```

#### $\mathbf{u_1}$

The operation $\mathbf{u_1} = 0x2 \cdot \mathbf{t_0} \oplus \mathbf{t_1}$ in Step 2 can be interpreted as the direct sum of $Mul2 \circ SS_0$ and $SS_1$.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

There are 6 linear structures:

```
([0 0 0 0 0 0 1 0],[1 0 1 0])
([0 0 0 0 0 1 0 0],[1 0 1 0])
([0 0 0 0 0 1 1 0],[1 0 1 0])
([0 0 1 1 0 0 0 0],[1 0 0 1])
([1 0 0 1 0 0 0 0],[1 0 0 1])
([1 0 1 0 0 0 0 0],[1 0 0 1])
```

#### $\mathbf{y_0}$

In the Step 3, $\mathbf{y_0}$ is obtained by composing $SS_2$ S-box with $\mathbf{u_0}$.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

There are no linear structures.

#### $\mathbf{y_1}$

In the Step 3, $\mathbf{y_1}$ is obtained by composing $SS_3$ S-box with $\mathbf{u_1}$.

Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

There are no linear structures.

### 8.2.4 S1

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$: Trace representation

Polynomial representation in ANF

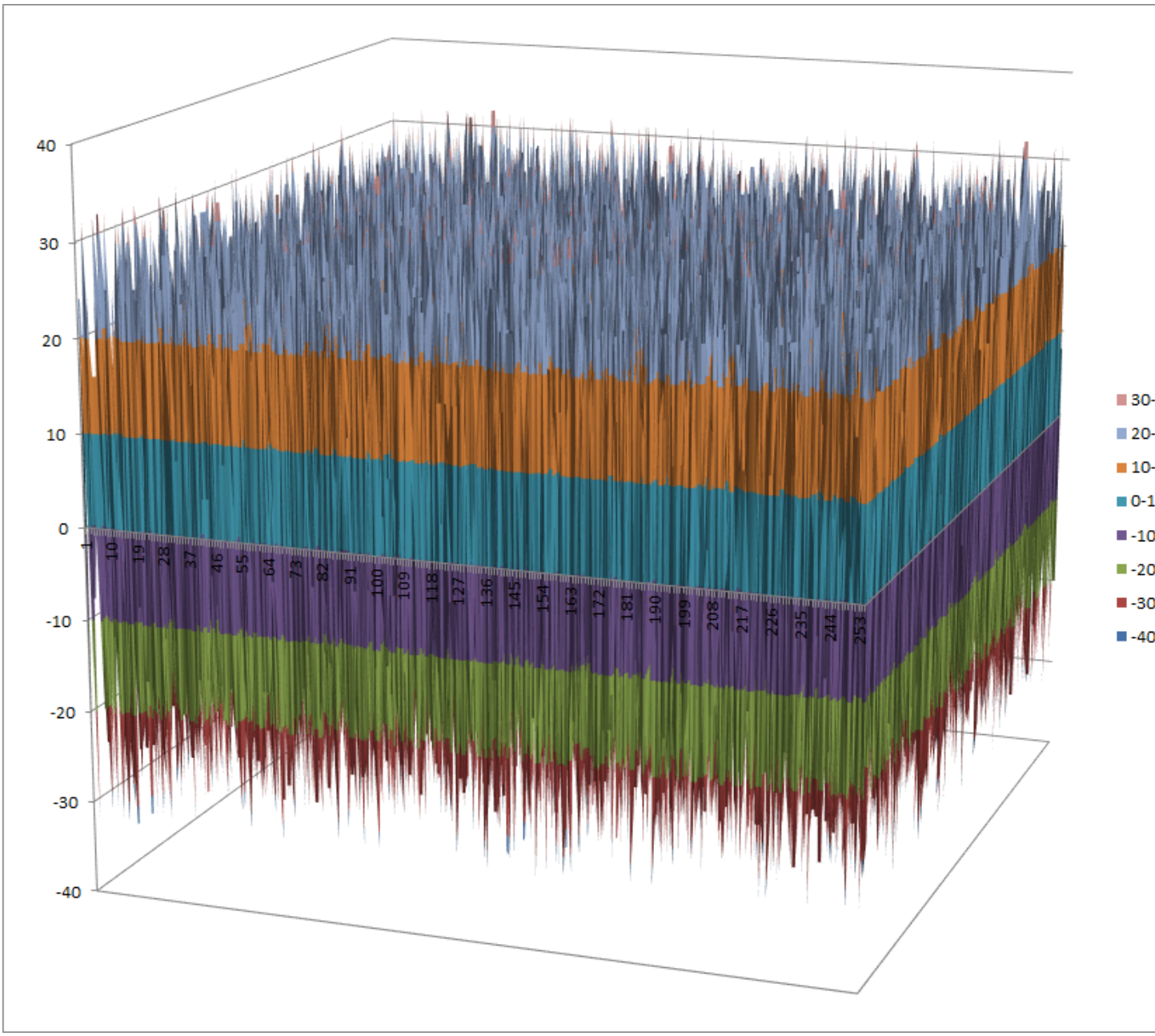
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 256 | 1 |

There are no linear structures

It has no fixed points.

It has 1 negated fixed point: (0,0,1,1,1,0,1,0)

## 8.3 Hierocrypt3

### 8.3.1 Description

*Hierocrypt3* and Hierocrypt-L1 are block ciphers created by Toshiba in 2000. They were submitted to the NESSIE project, but were not selected. Both algorithms were among the cryptographic techniques recommended for Japanese government use by CRYPTREC in 2003, however, both have been dropped to "candidate" by CRYPTREC revision in 2013. Both of them have the same 8x8 S-box: S

### 8.3.2 Summary

| S-box | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|
| S | 112 | 56v | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |

### 8.3.3 S

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$: Trace representation

Polynomial representation in ANF

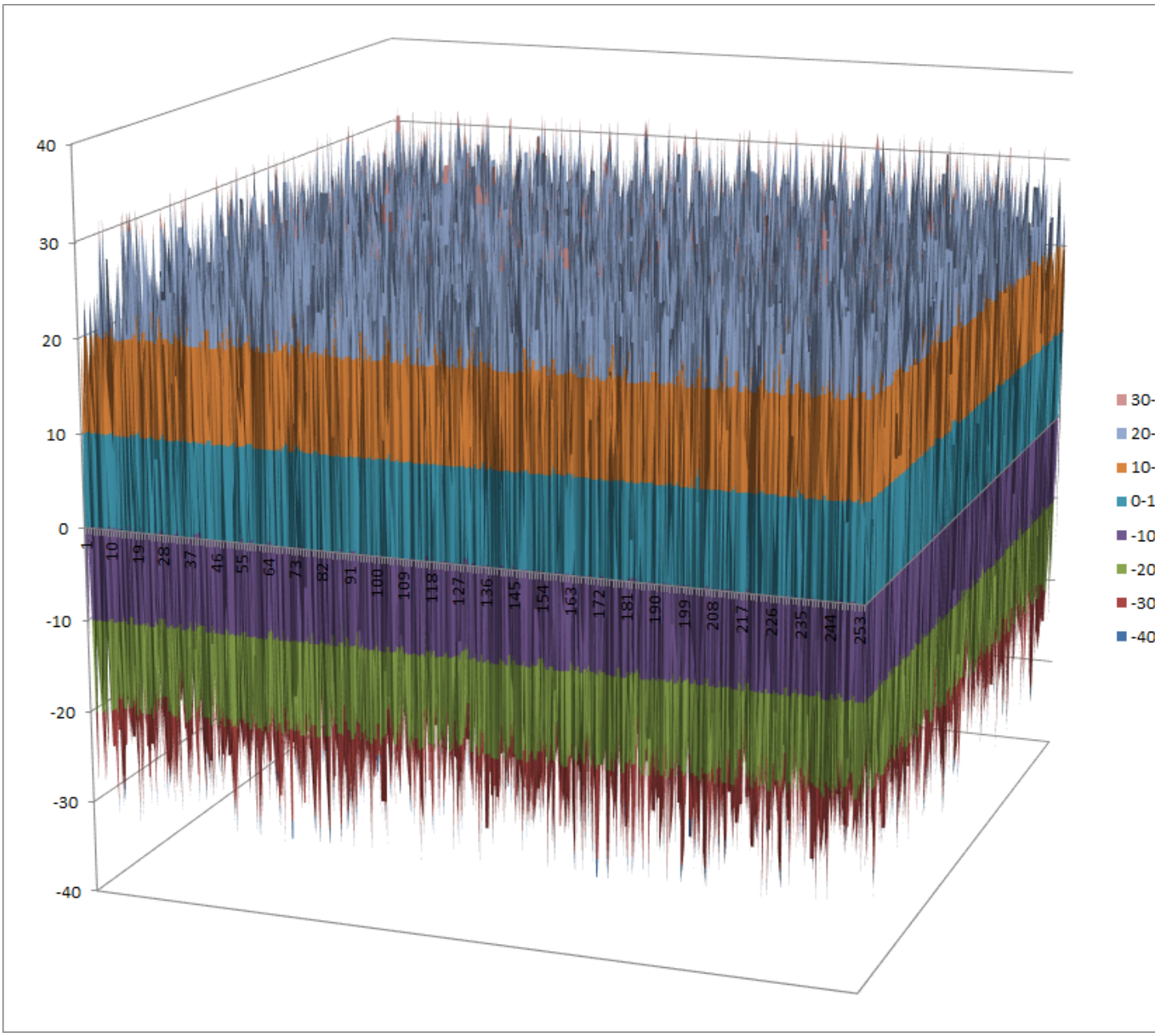
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 109 | 1 |
| 141 | 1 |

There are no linear structures

It has 1 fixed point: (0,1,1,0,0,1,1,1)

It has 2 negated fixed points: (0,0,0,1,0,1,1,1), (1,0,0,1,0,1,0,0)

## 8.4 SC2000

### 8.4.1 Description

*SC2000* is a block cipher invented by a research group at Fujitsu Labs. It was submitted to the NESSIE project, but was not selected. It was among the cryptographic techniques recommended for Japanese government use by CRYPTREC in 2003, however, has been dropped to "candidate" by CRYPTREC revision in 2013. It has three 3 S-boxes: S4,S5,S6

### 8.4.2 Summary

| S-box | size | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|---|
| S4 | 4x4 | 4 | 0 | 0 | 2 | 2 | 16 | 1024 | 0.25 | 0.25 |
| S5 | 5x5 | 12 | 6 | 6 | 3 | 3 | 8 | 2048 | 0.0625 | 0.0625 |
| S6 | 6x6 | 24 | 14 | 12 | 5 | 3 | 16 | 8704 | 0.0625 | 0.0625 |

### 8.4.3 S4

#### Representations

Polynomial representation in ANF

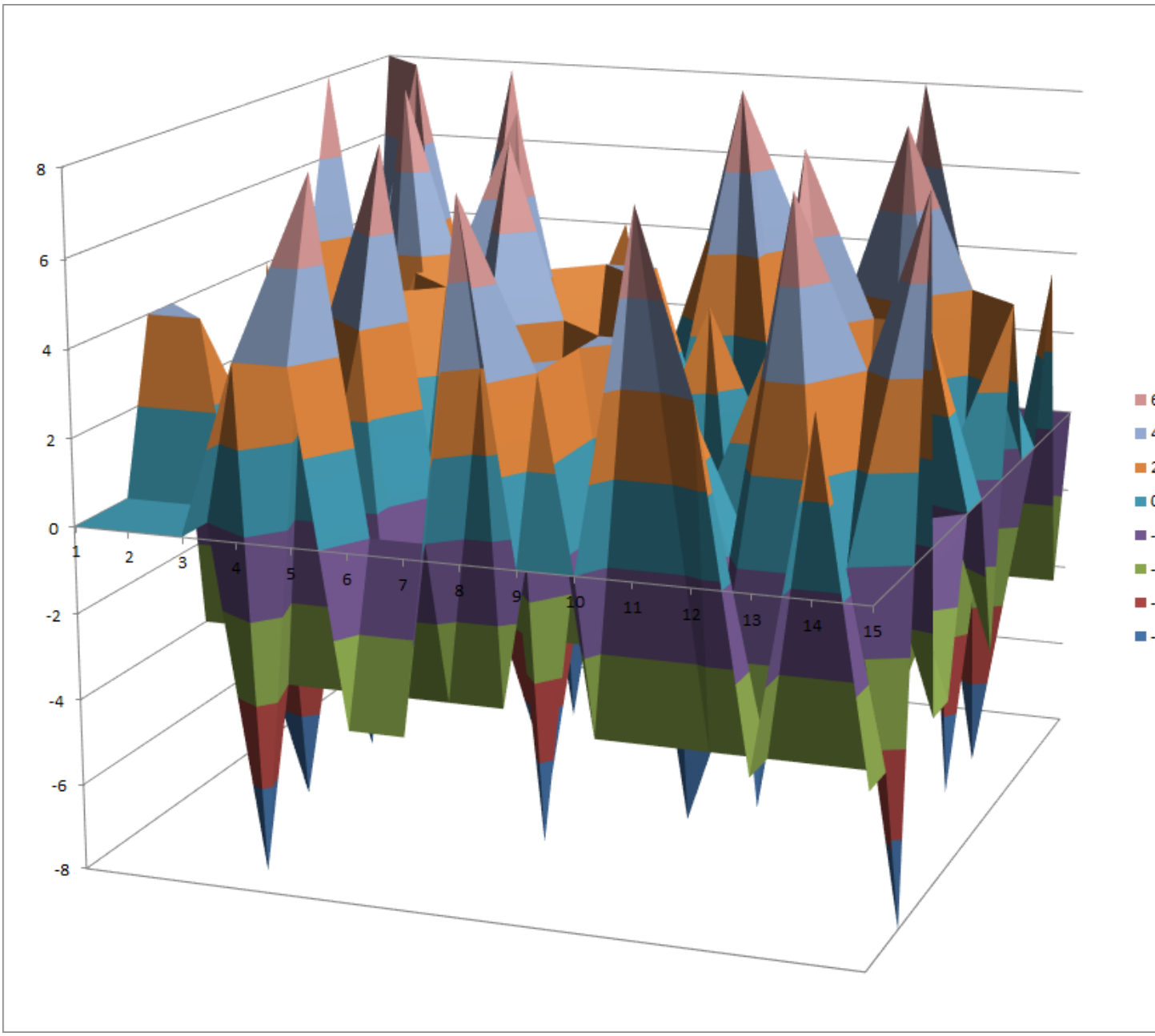
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 3 | 1 |
| 11 | 1 |

There are 3 linear structures:

```
([0 1 0 1],[0 0 1 1])
([1 0 0 1],[0 0 1 1])
([1 1 0 0],[0 0 1 1])
```

It has no fixed points.

It has 2 negated fixed points: (0,0,1,1), (1,0,0,1)

### 8.4.4 S5

#### Representations

Polynomial representation in ANF

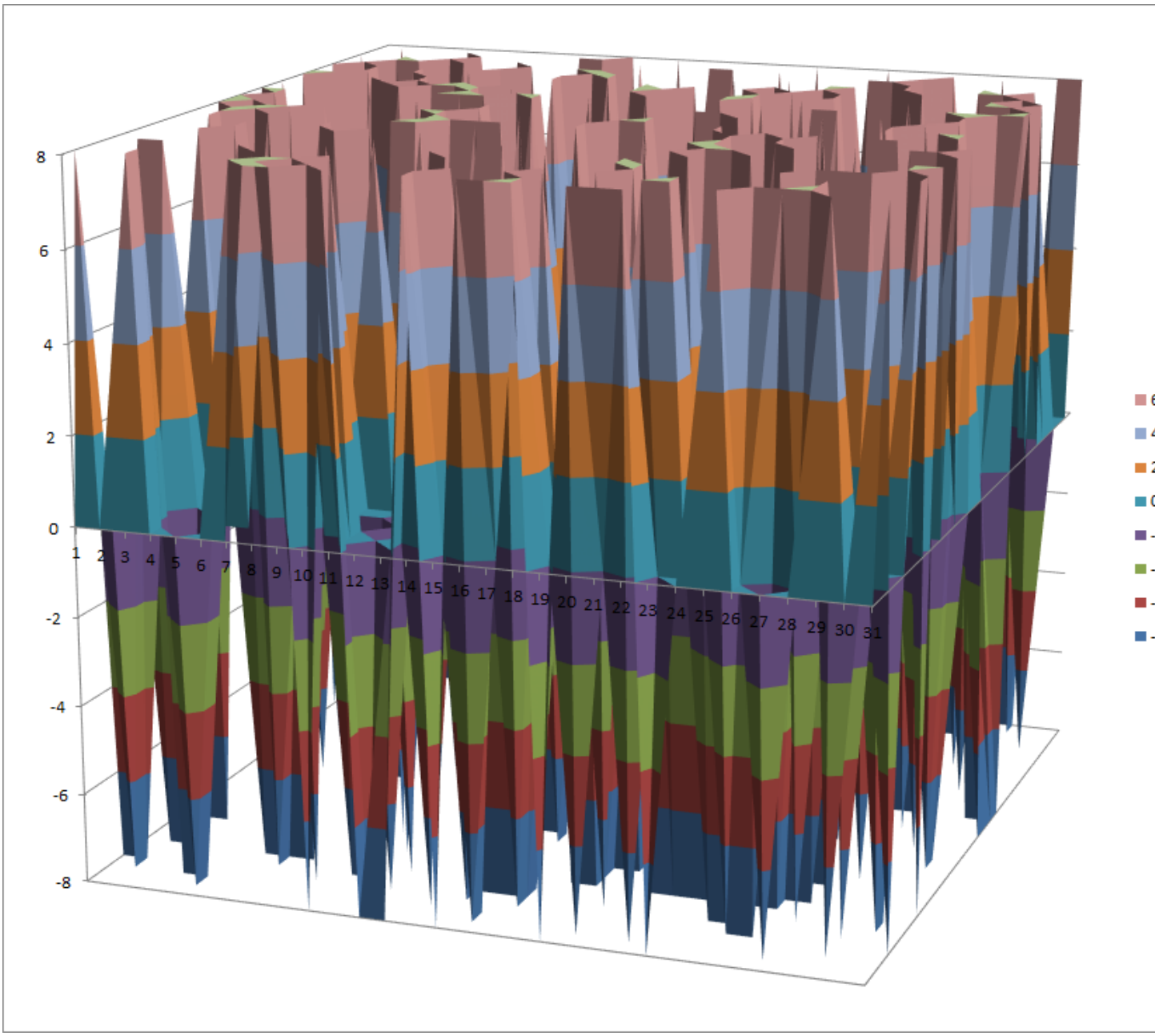
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 6 | 1 |
| 8 | 2 |
| 10 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points

### 8.4.5 S6

#### Representations

Polynomial representation in ANF

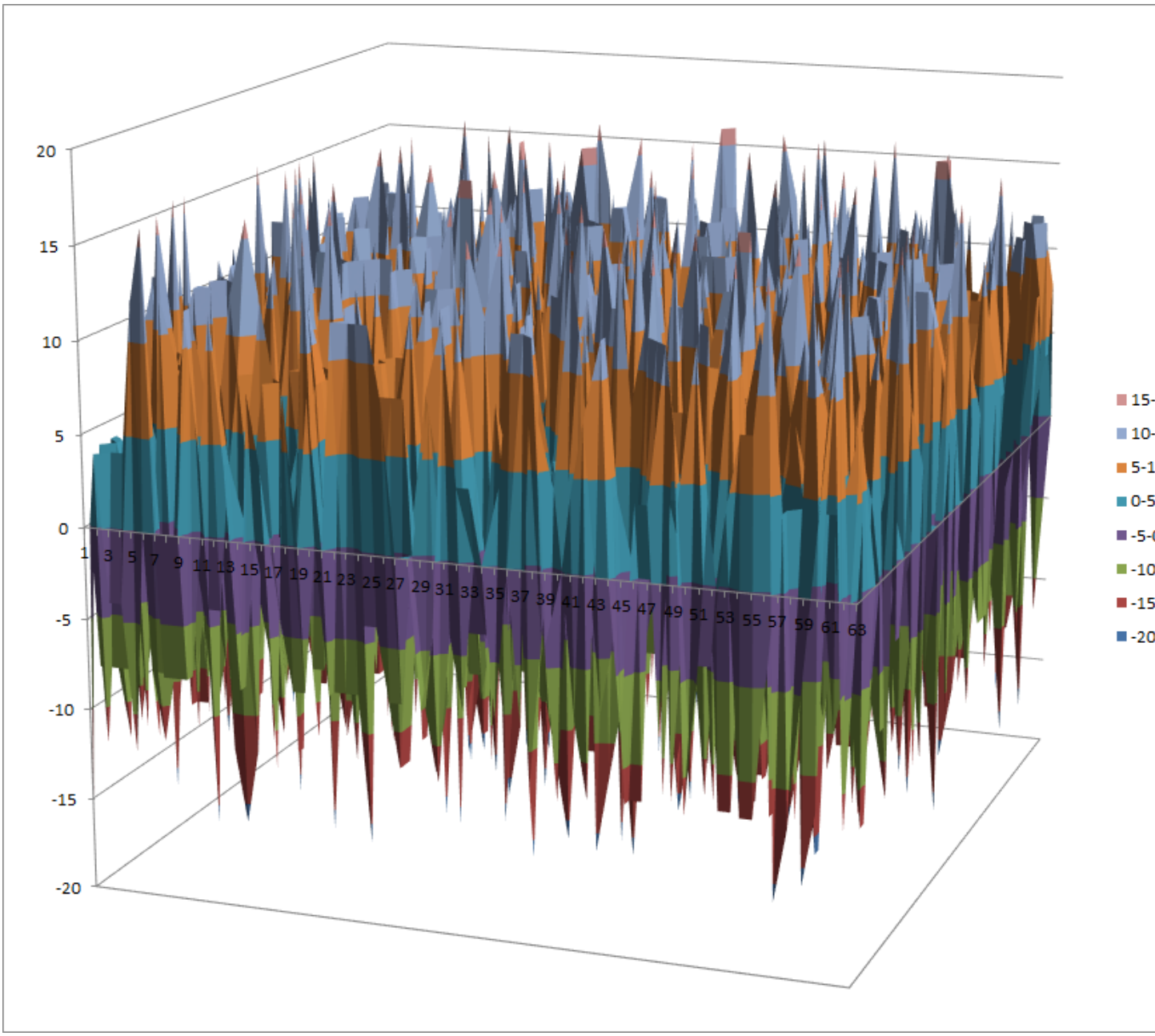
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 2 |
| 9 | 1 |
| 17 | 1 |
| 34 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points

# 9 Analysis of NESSIE project cryptographic algorithms

The NESSIE call includes a request for a broad set of algorithms providing date confidentiality, data authentication, and entity authentication. These algorithms include block ciphers, stream ciphers, hash functions, MAC algorithms, digital signature schemes, and public-key encryption and identification schemes. In this chapter, several cryptographic algorithms from NESSIE (New European Schemes for Signature, Integrity, and Encryption) research project candidates are analysed.

Below you can find a legend describing the cryptographic criteria used in this chapter:

| | |
|---|---|
| **NL** | Nonlinearity |
| **NL2** | 2-nd order nonlinearity |
| **LD** | Linearity distance |
| **DEG** | Algebraic degree |
| **AI** | Algebraic immunity |
| **MAXAC** | Absolute indicator |
| $\sigma$ | Sum-of-squares indicator |
| **LP** | Linear potential |
| **DP** | Differential Potential |

**Hyperlinks to representations**

Open the hyperlinks to representations below in a new browser window or in a new tab.

## 9.1 Anubis

### 9.1.1 Description

Anubis is a block cipher designed by Vincent Rijmen and Paulo S. L. M. Barreto as an entrant in the NESSIE project, a former research program initiated by the European Commission in 2000 for the identification of new cryptographic algorithms. Although the cipher has not been included in the final NESSIE portfolio, its design is considered very strong, and no attacks have been found by 2004 after the project had been concluded. It has a 8x8 S-box called S.

### 9.1.2 Summary

| S-box | size | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|---|
| S | 8x8 | 94 | • | 40 | 7 | 4 | 96 | 272896 | 0.0705566406 | 0.0063125 |

### 9.1.3 S

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$: Trace representation

Polynomial representation in ANF

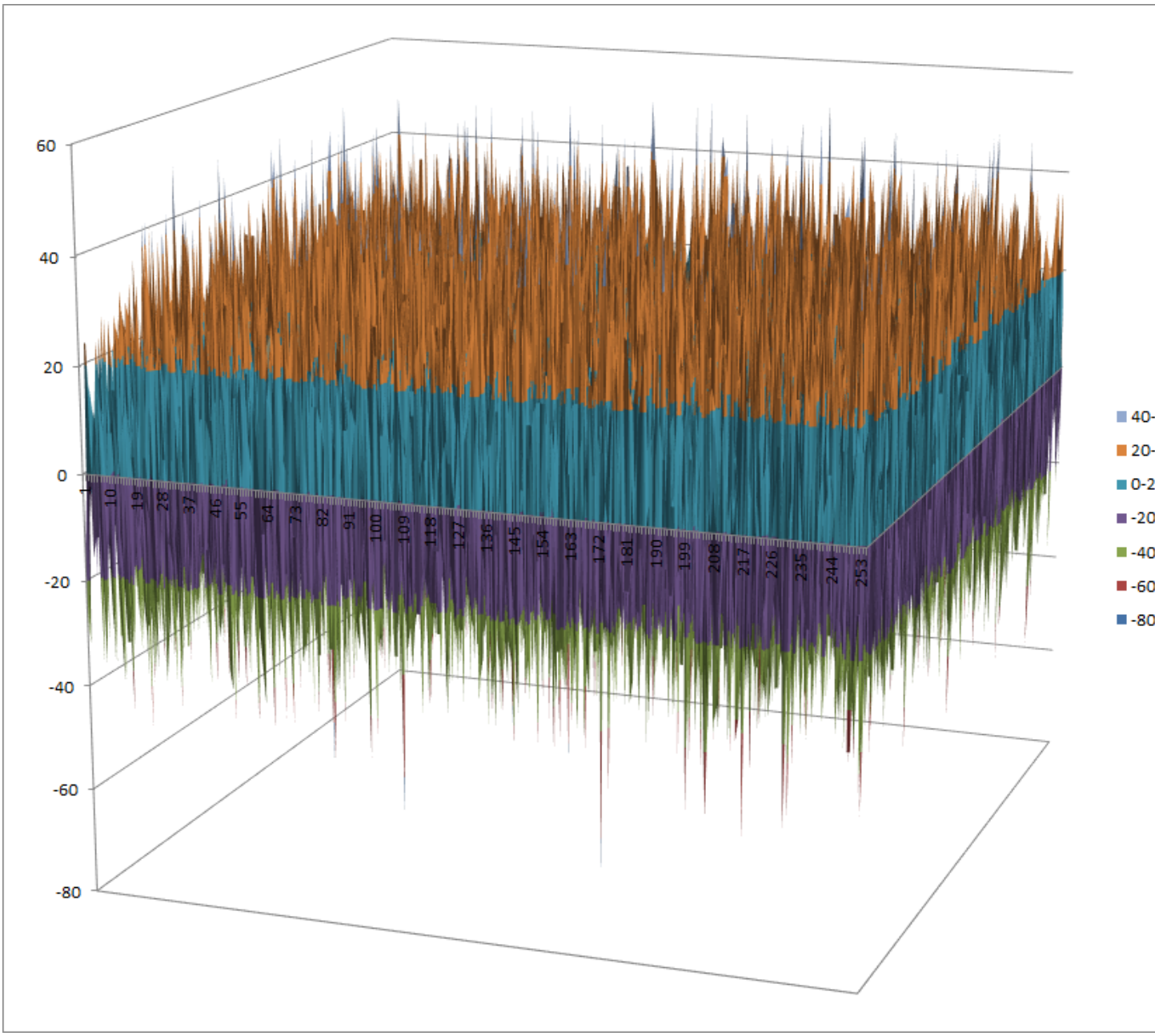
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 128 |

There are no linear structures

It has no fixed points

It has 2 negated fixed points: (0,1,1,0,1,1,0,0), (1,0,0,1,0,0,1,1)

## 9.2 Camellia

### 9.2.1 Description

Camellia is a symmetric key block cipher with a block size of 128 bits and key sizes of 128, 192 and 256 bits. It was jointly developed by Mitsubishi and NTT of Japan. The cipher has been approved for use by the ISO/IEC, the European Union's NESSIE project and the Japanese CRYPTREC project. It has four 8x8 S-boxes called S1, S2, S3, S4.

### 9.2.2 Summary

| S-box | size | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S1 | 8x8 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| S2 | 8x8 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| S3 | 8x8 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |
| S4 | 8x8 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |

### 9.2.3 S1

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^3 + 1$: Trace representation

Polynomial representation in ANF

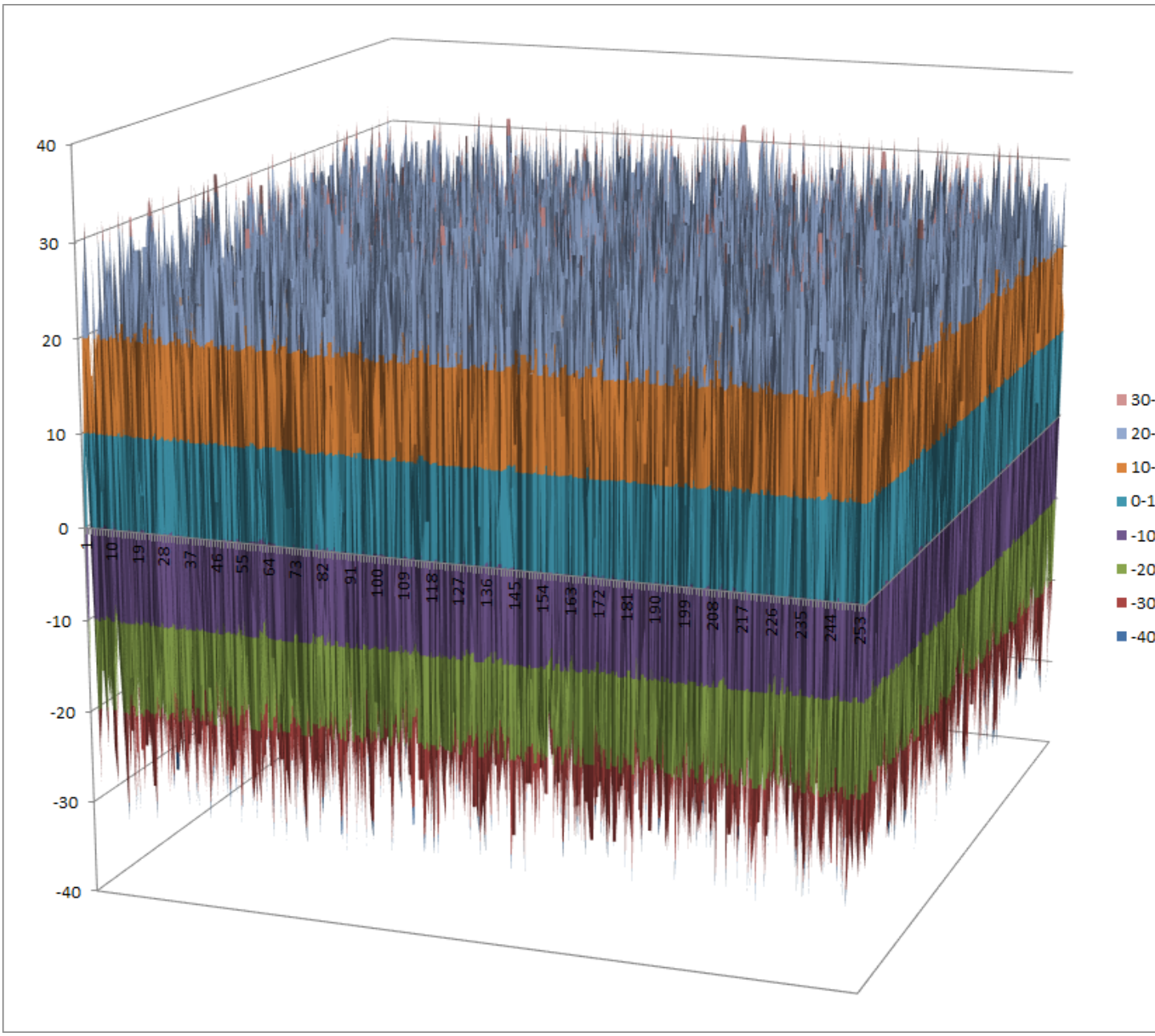
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 5 | 1 |
| 249 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points

### 9.2.4 S2

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^3 + 1$: Trace representation

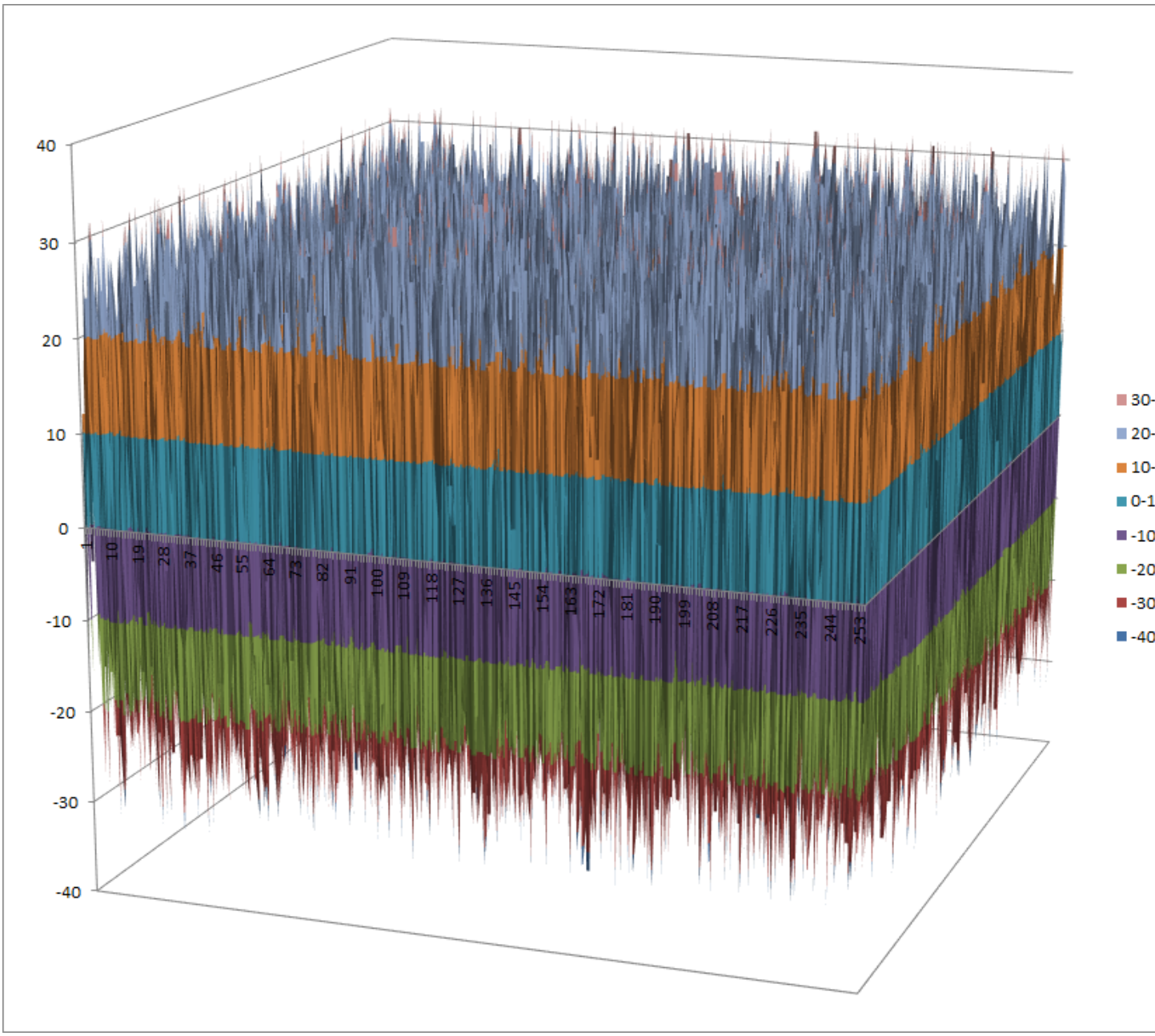
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 10 | 1 |
| 13 | 1 |
| 51 | 1 |
| 71 | 1 |
| 111 | 1 |

There are no linear structures

It has no fixed points.

It has 3 negated fixed points: (0,1,0,1,1,1,0,0), (1,0,1,1,0,0,1,1), (1,1,1,1,1,1,1,0)

### 9.2.5 S3

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^3 + 1$: Trace representation

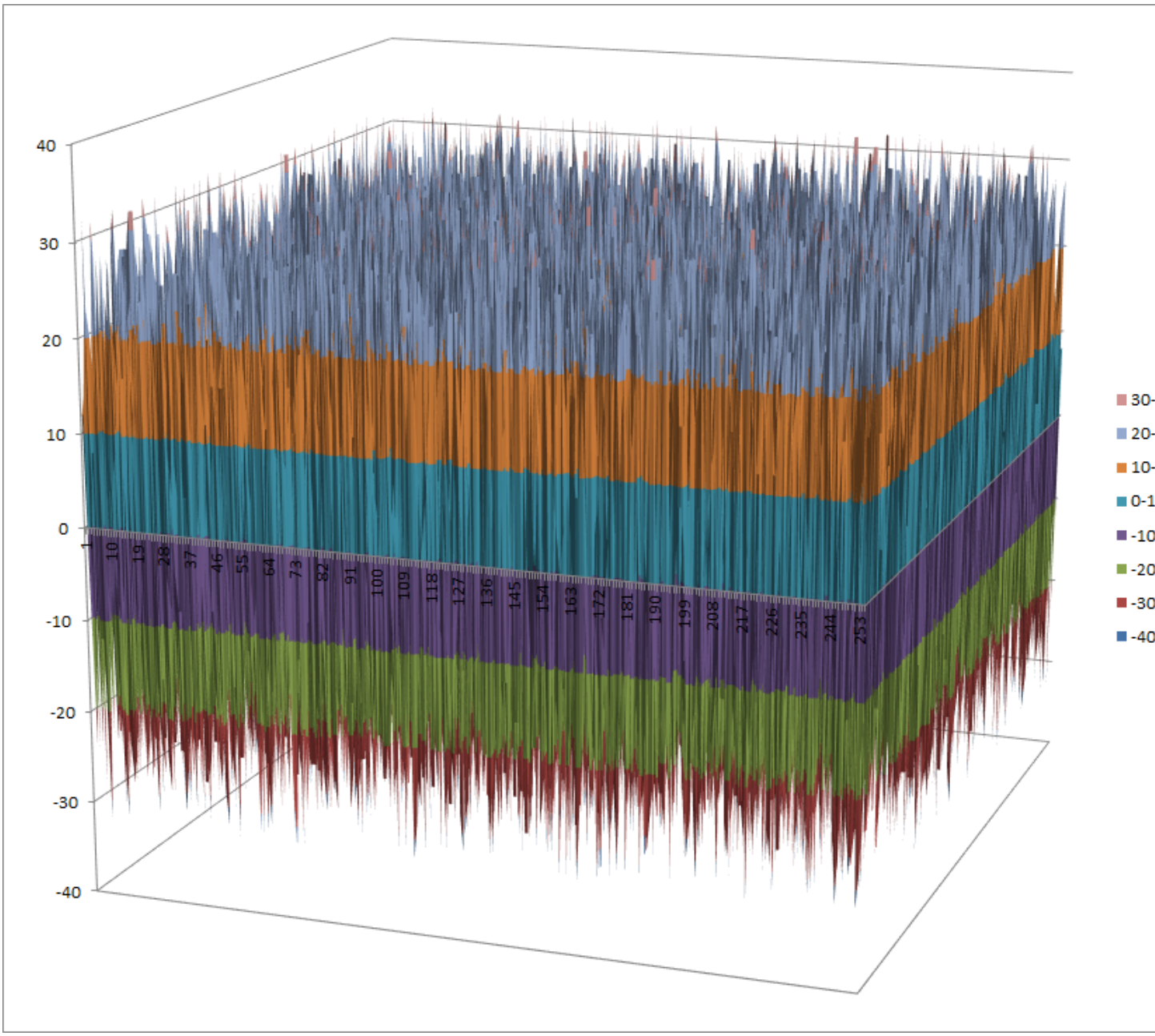
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 5 | 1 |
| 68 | 1 |
| 183 | 1 |

There are no linear structures

It has no fixed points.

It has 1 negated fixed point: (1,1,1,0,1,0,1,0)

### 9.2.6 S4

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^6 + x^5 + x^3 + 1$: Trace representation

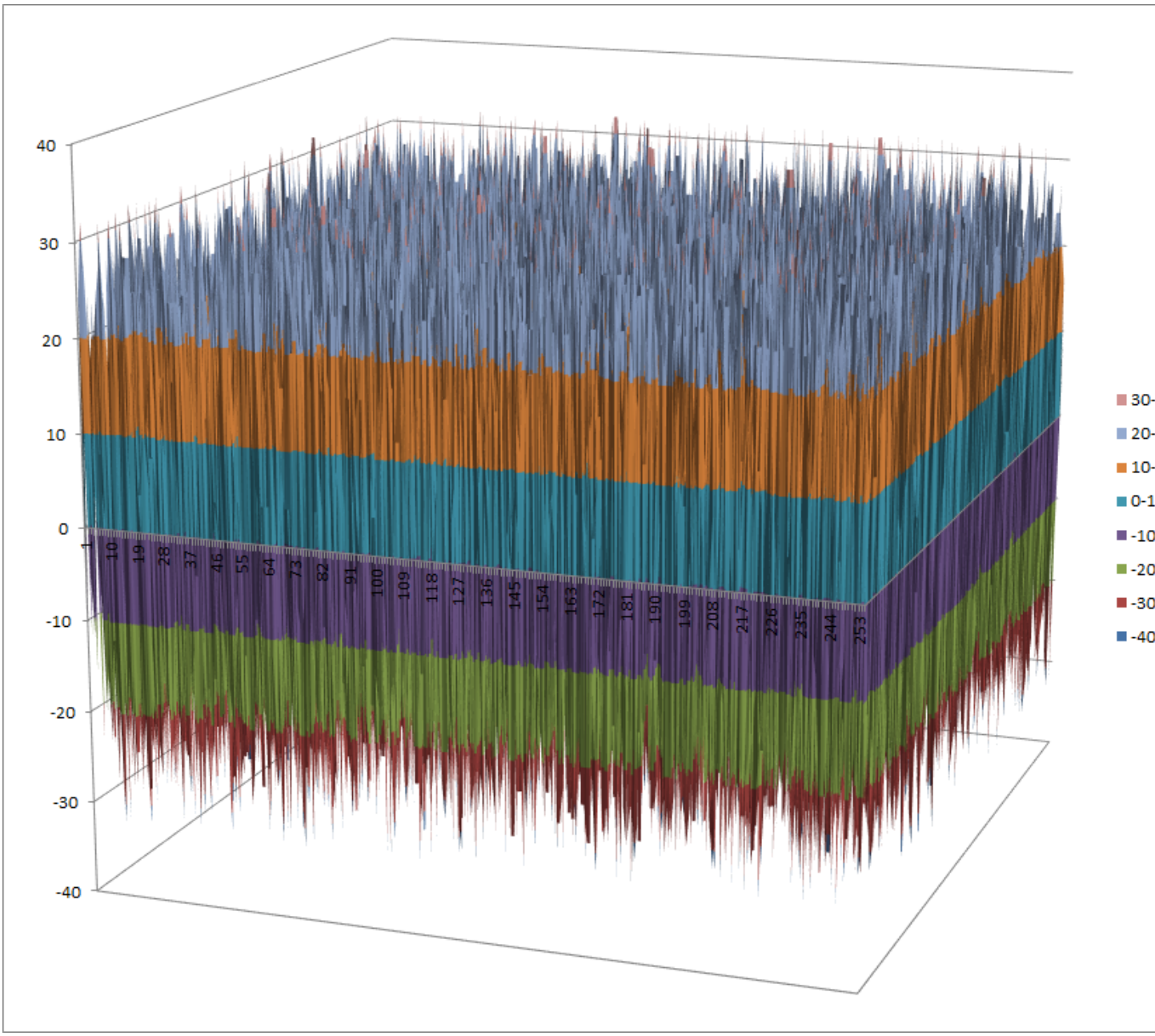
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 10 | 1 |
| 13 | 1 |
| 51 | 1 |
| 71 | 1 |
| 111 | 1 |

There are no linear structures

It has no fixed points.

It has 3 negated fixed points: (0,0,1,0,1,1,1,0), (0,1,1,1,1,1,1,1), (1,1,0,1,1,0,0,1)

## 9.3 Grand Cru

### 9.3.1 Description

Grand Cru is a block cipher invented in 2000 by Johan Borst. It was submitted to the NESSIE project, but was not selected. It has a Non-linear Substitution Transformation which uses a 8x8 S-box called S.

### 9.3.2 Summary

| S-box | size | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S | 8x8 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |

### 9.3.3 S

#### Representations

Polynomial function over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$: Trace representation

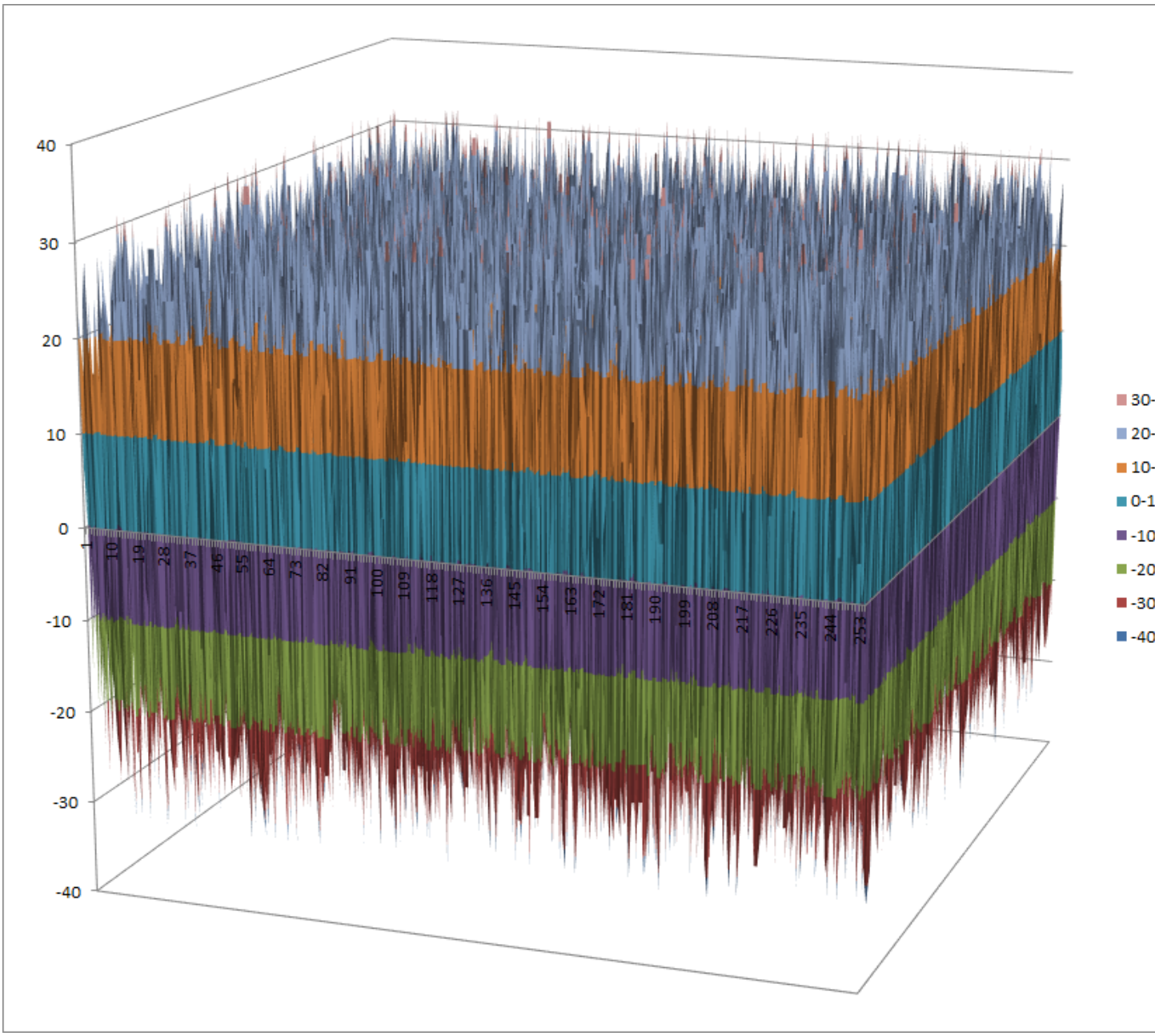
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 27 | 1 |
| 59 | 1 |
| 81 | 1 |
| 87 | 1 |

There are no linear structures

It has no fixed points. It has no negated fixed points

# 10 Analysis of other cryptographic algorithms

There are some block ciphers that are relevant either because its wide use or because of its importance in the development of cryptanalysis techniques. In this chapter, some cryptographic algorithms from other block ciphers are analysed.

Below you can find a legend describing the cryptographic criteria used in this chapter:

| | |
|---|---|
| **NL** | Nonlinearity |
| **NL2** | 2-nd order nonlinearity |
| **LD** | Linearity distance |
| **DEG** | Algebraic degree |
| **AI** | Algebraic immunity |
| **MAXAC** | Absolute indicator |
| $\sigma$ | Sum-of-squares indicator |
| **LP** | Linear potential |
| **DP** | Differential Potential |

---

**Hyperlinks to representations**

Open the hyperlinks to representations below in a new browser window or in a new tab.

---

## 10.1 DES

### 10.1.1 Description

*DES* is a block cipher used for securing sensitive but unclassified material by U.S. Government agencies and became the de facto encryption standard for commercial transactions in the private sector up to December 6, 2001. It was superseded by AES. DES has eight 6x4 S-boxes: S1, S2, S3, S4, S5, S6, S7, S8.

### 10.1.2 Summary

| S-box | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S1 | 14 | 8 | 4 | 4 | 2 | 48 | 36736 | 0.31640625 | 0.25 |
| S2 | 16 | 10 | 2 | 4 | 3 | 56 | 25984 | 0.25 | 0.25 |
| S3 | 16 | 10 | 4 | 4 | 3 | 48 | 24064 | 0.25 | 0.25 |
| S4 | 16 | 8 | 0 | 3 | 2 | 64 | 40960 | 0.25 | 0.25 |
| S5 | 12 | 10 | 6 | 4 | 2 | 40 | 47104 | 0.390625 | 0.25 |
| S6 | 18 | 10 | 4 | 5 | 3 | 48 | 19456 | 0.19140625 | 0.25 |
| S7 | 14 | 10 | 4 | 5 | 3 | 48 | 34048 | 0.31640625 | 0.25 |
| S8 | 16 | 10 | 4 | 4 | 3 | 48 | 24064 | 0.25 | 0.25 |

### 10.1.3 S1

#### Representations

Polynomial representation in ANF

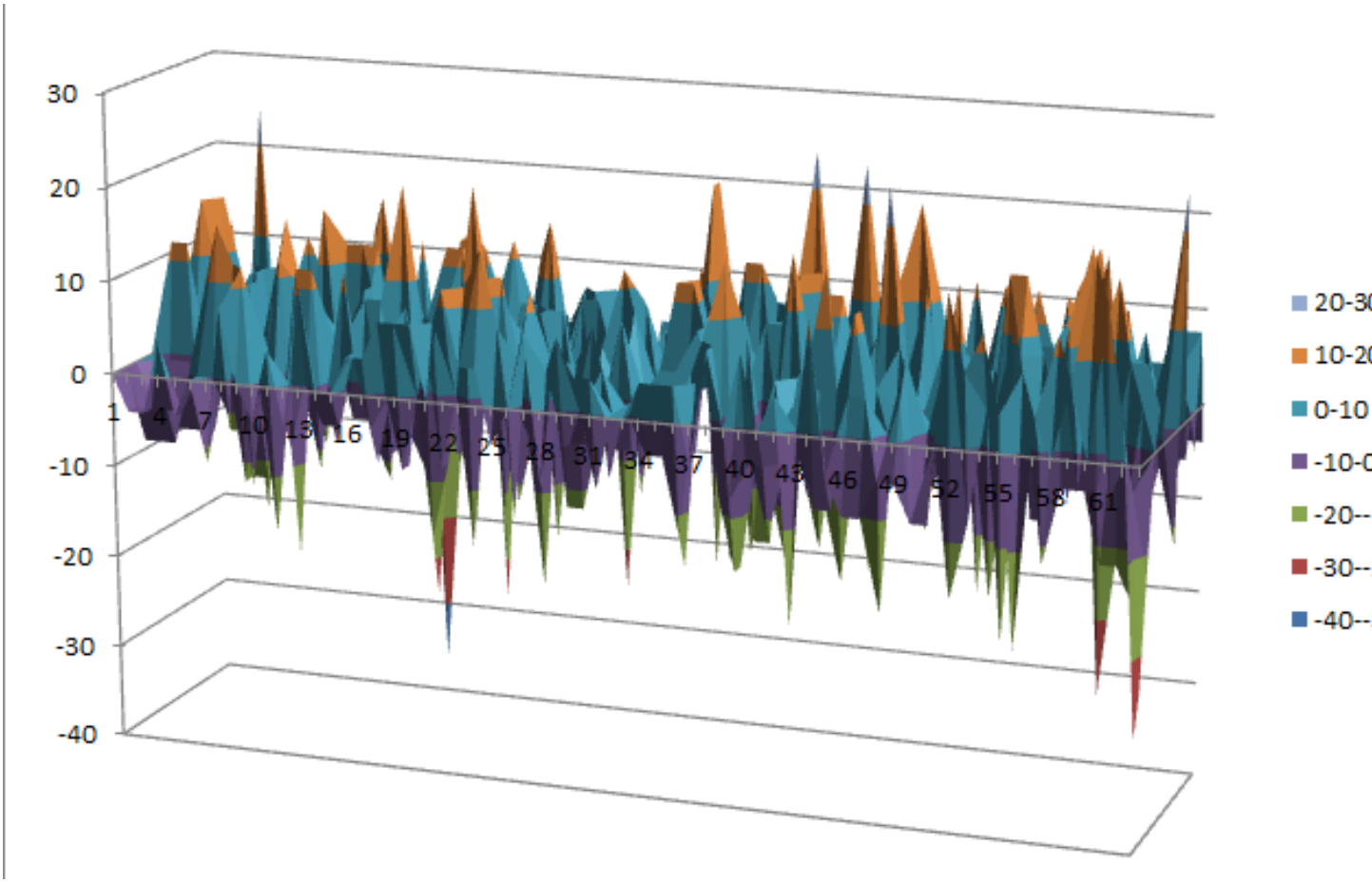
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.1.4 S2

### Representations

Polynomial representation in ANF

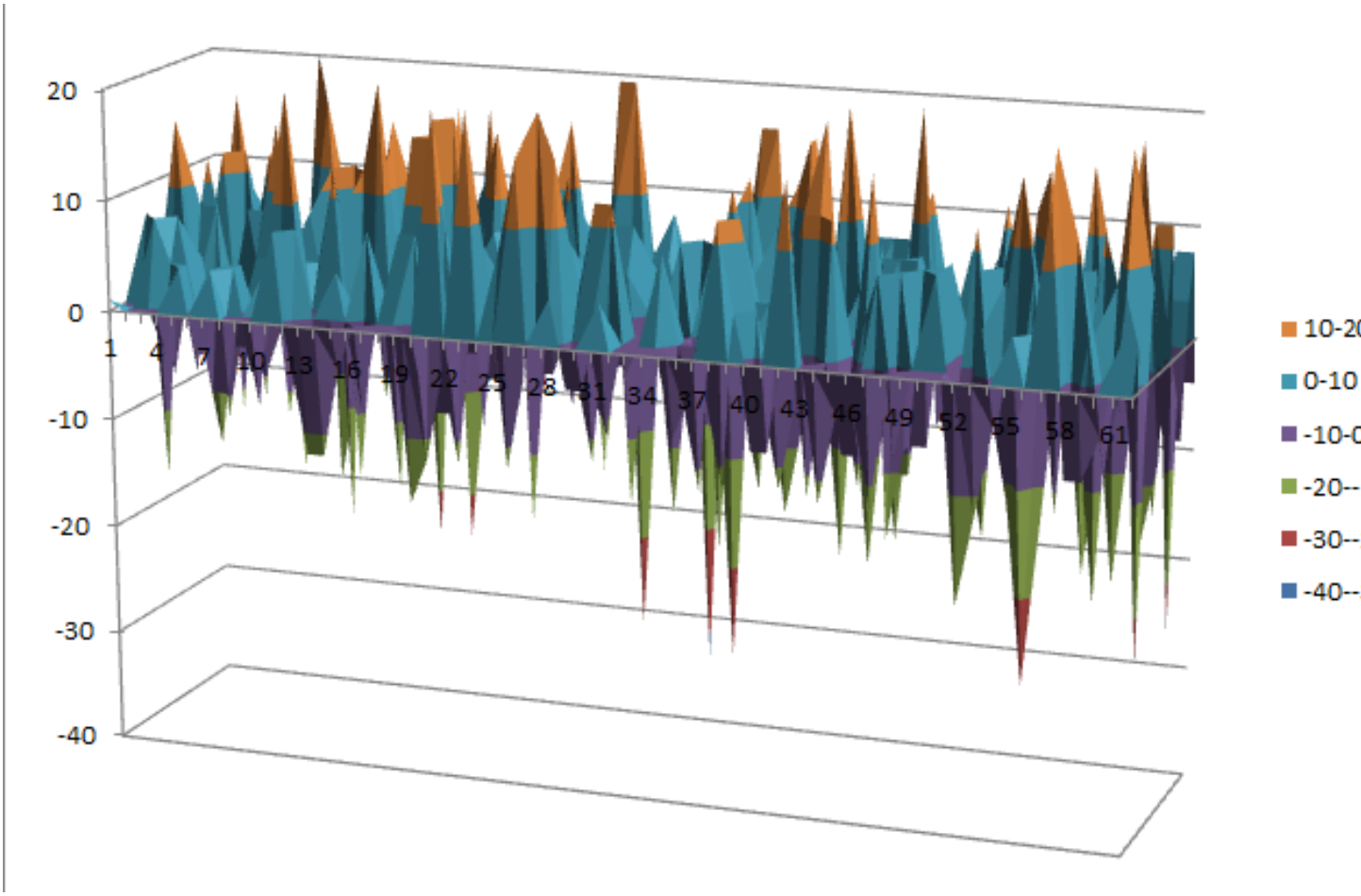
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.1.5 S3

### Representations

Polynomial representation in ANF

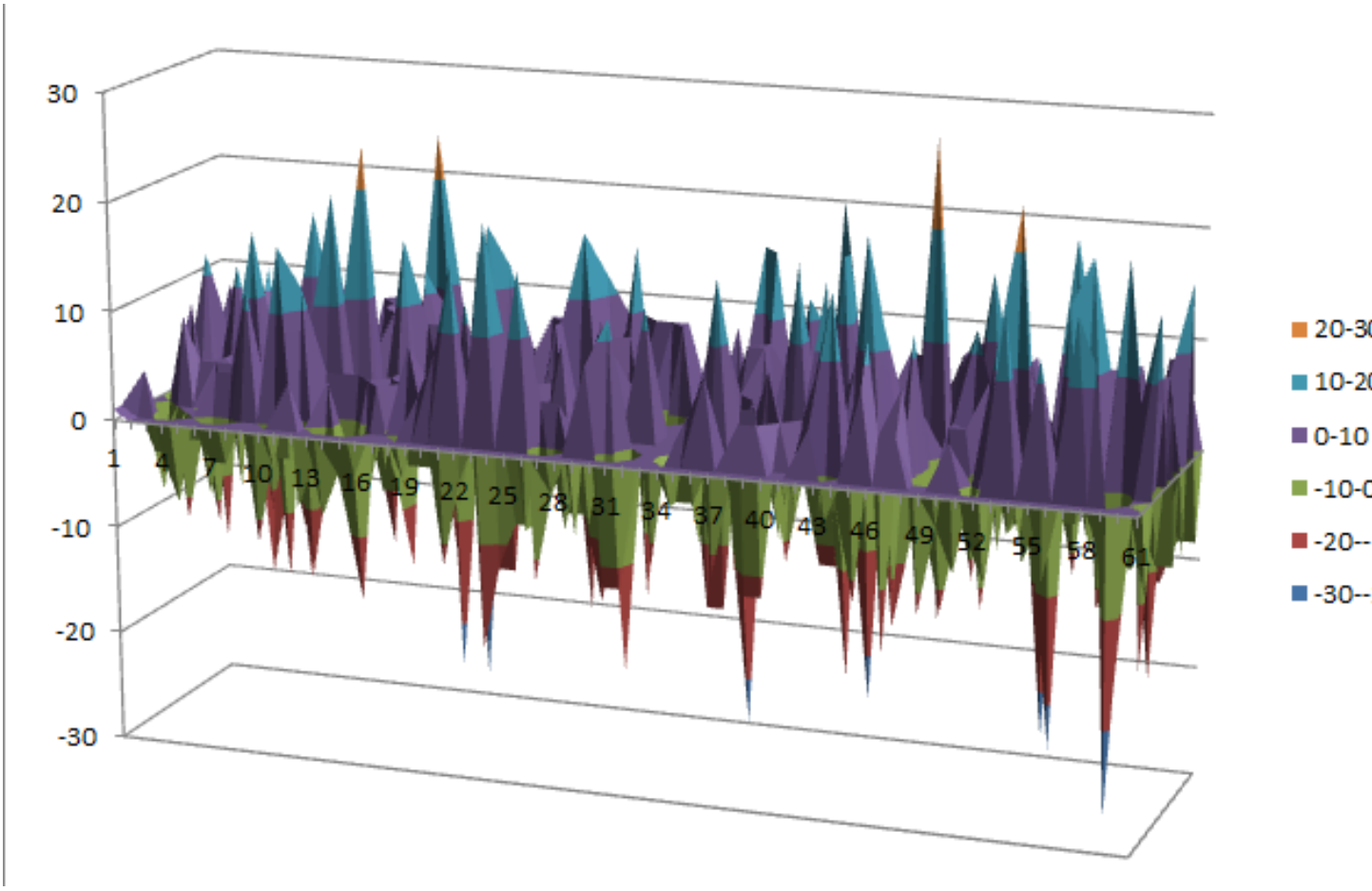
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.1.6 S4

### Representations

Polynomial representation in ANF

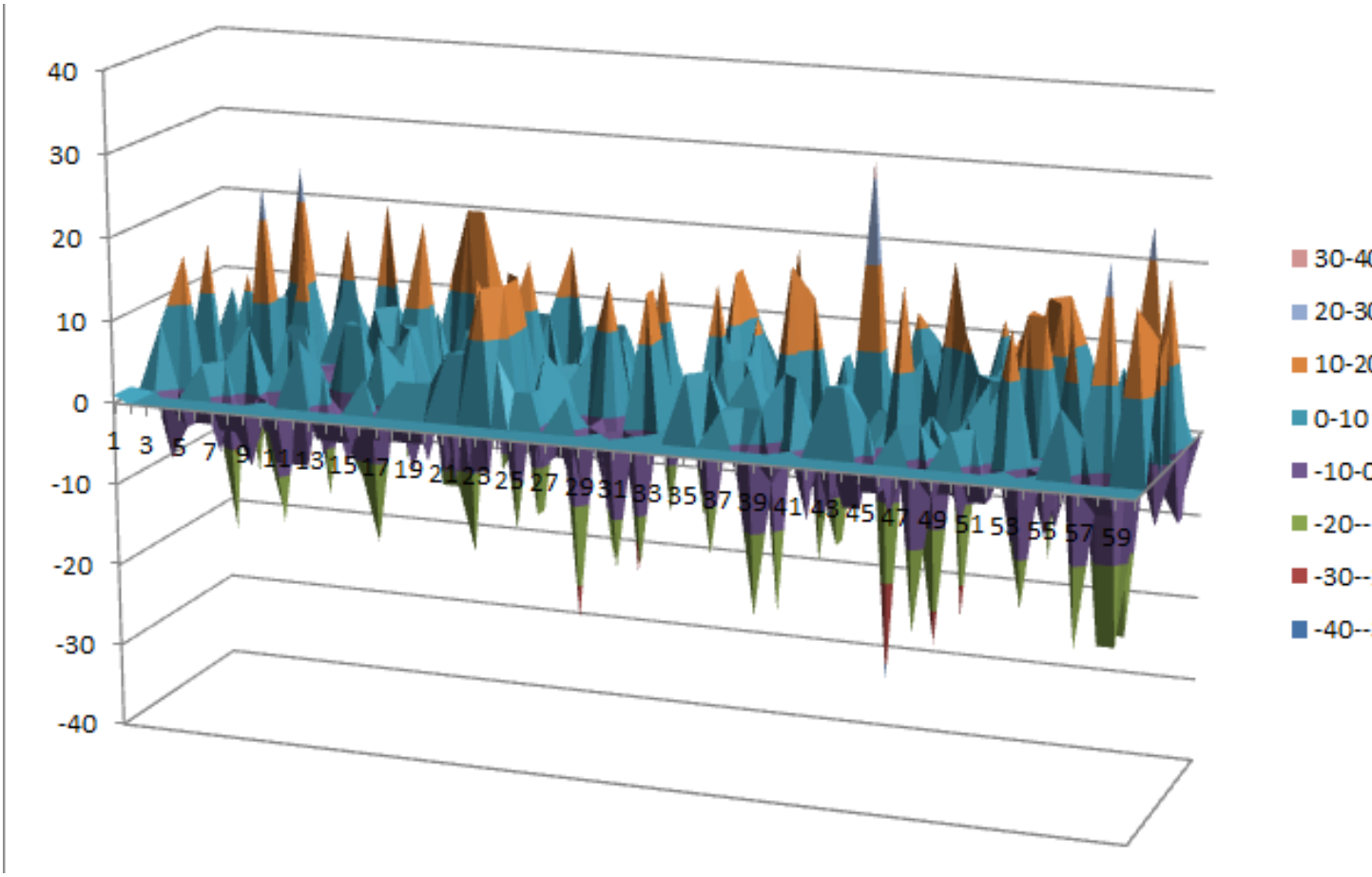
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are 9 linear structures:

```
([0 0 0 0 0 1],[0 0 1 1])
([0 0 0 0 0 1],[1 1 0 0])
([0 0 0 0 0 1],[1 1 1 1])
([1 0 1 1 1 0],[0 1 0 1])
([1 0 1 1 1 0],[1 0 1 0])
([1 0 1 1 1 0],[1 1 1 1])
([1 0 1 1 1 1],[0 1 1 0])
([1 0 1 1 1 1],[1 0 0 1])
([1 0 1 1 1 1],[1 1 1 1])
```

### 10.1.7 S5

### Representations

Polynomial representation in ANF

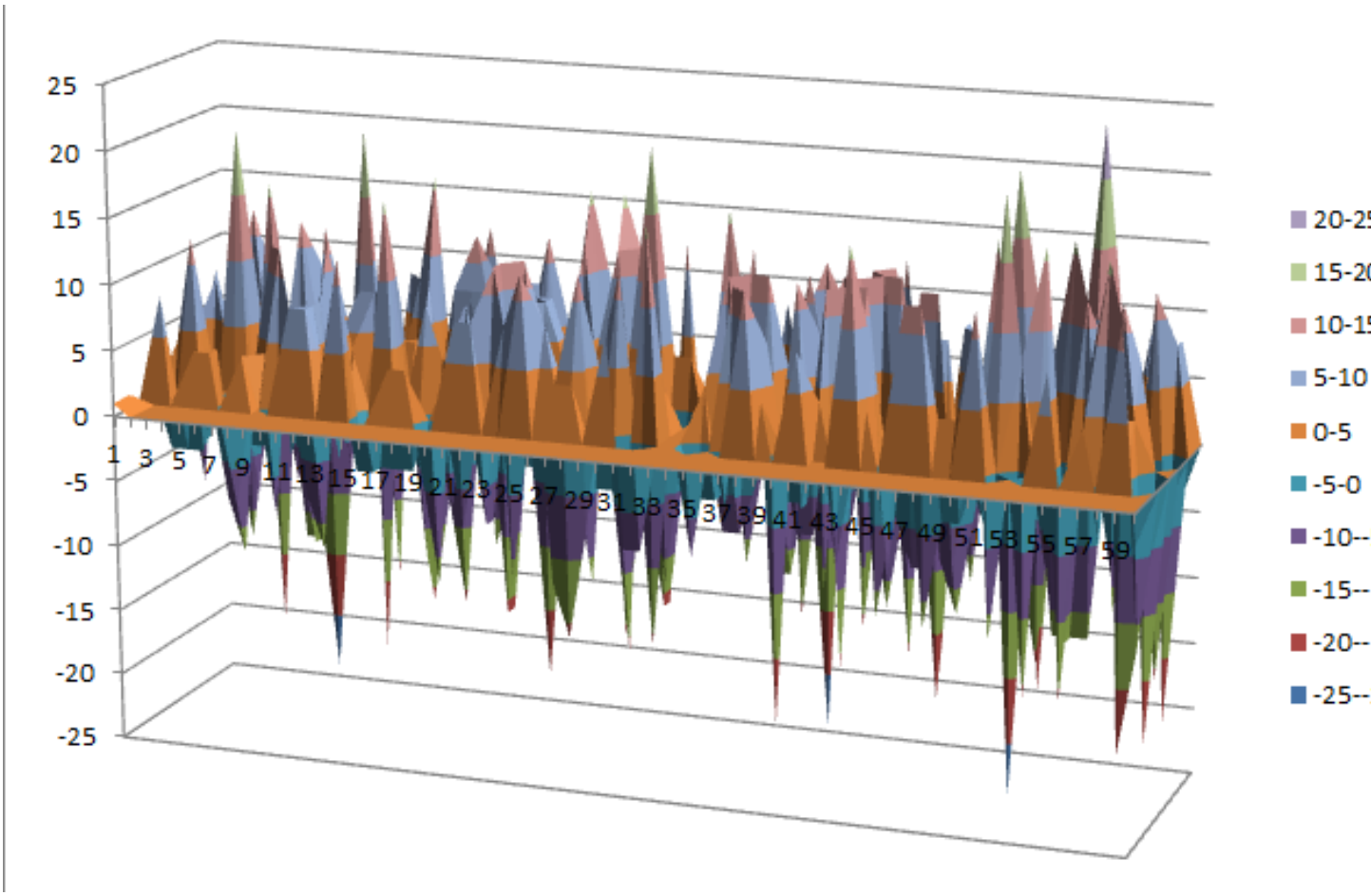
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.1.8 S6

### Representations

Polynomial representation in ANF

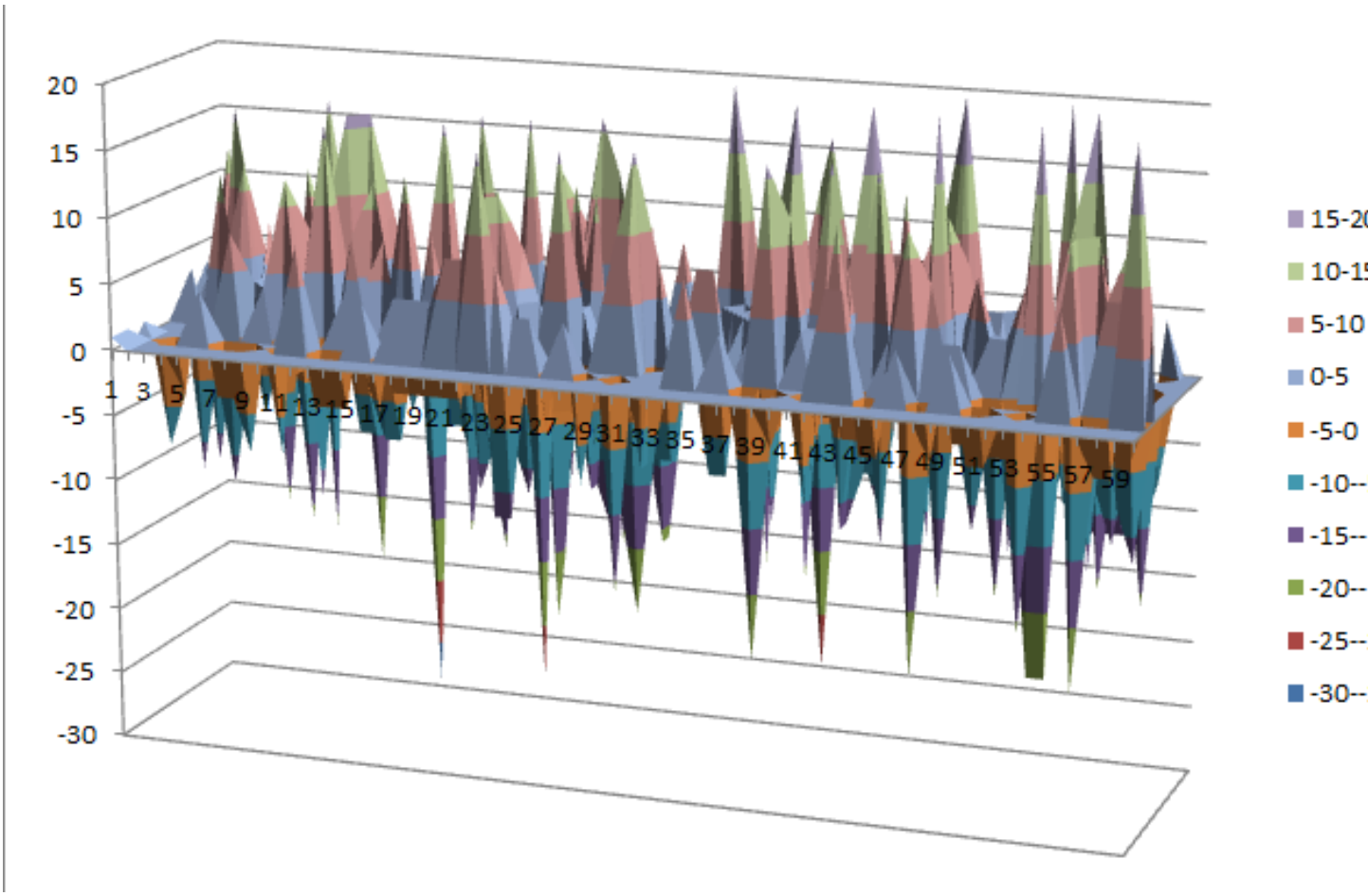
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.1.9 S7

### Representations

Polynomial representation in ANF

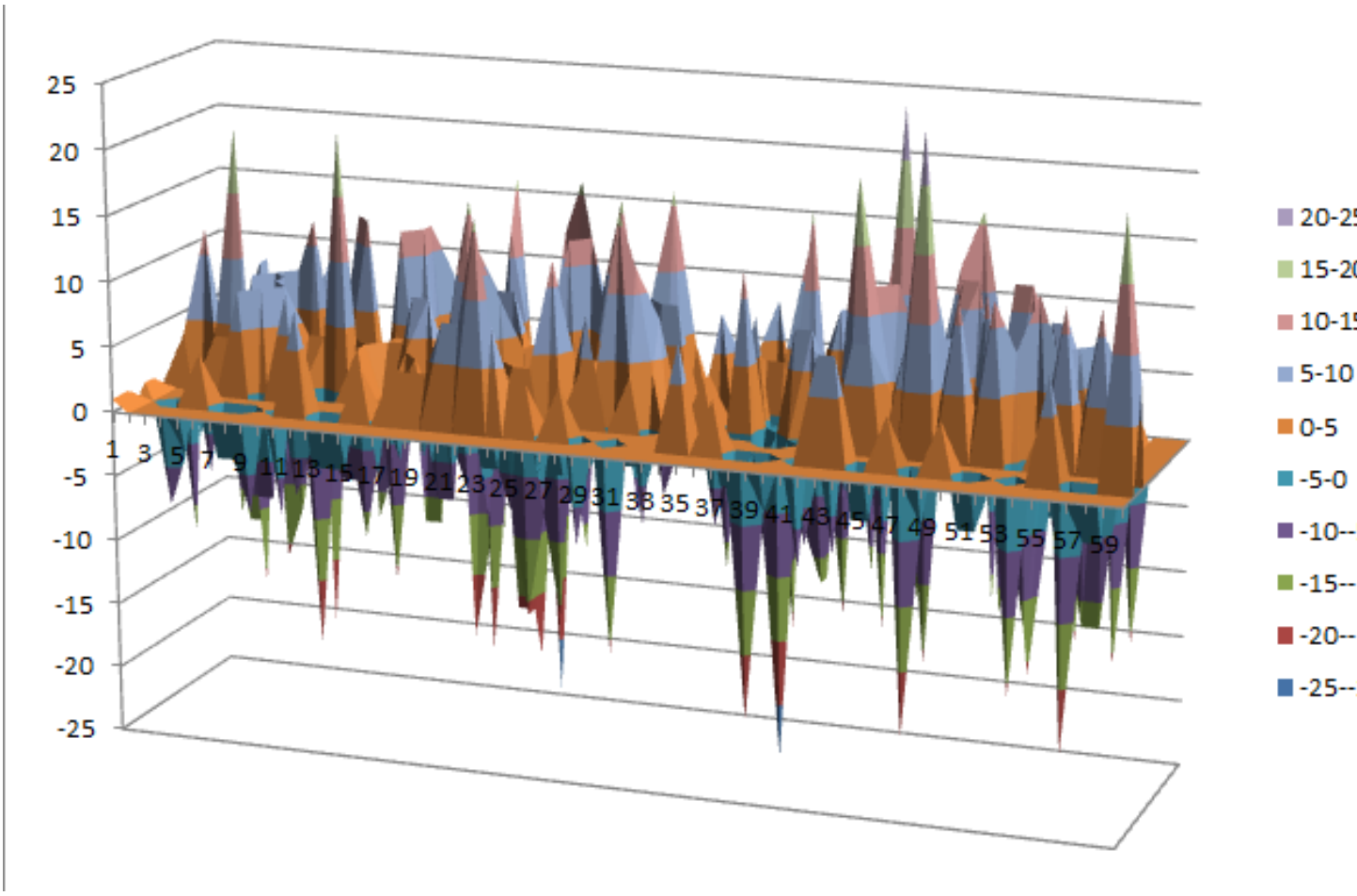
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.1.10 S8

### Representations

Polynomial representation in ANF

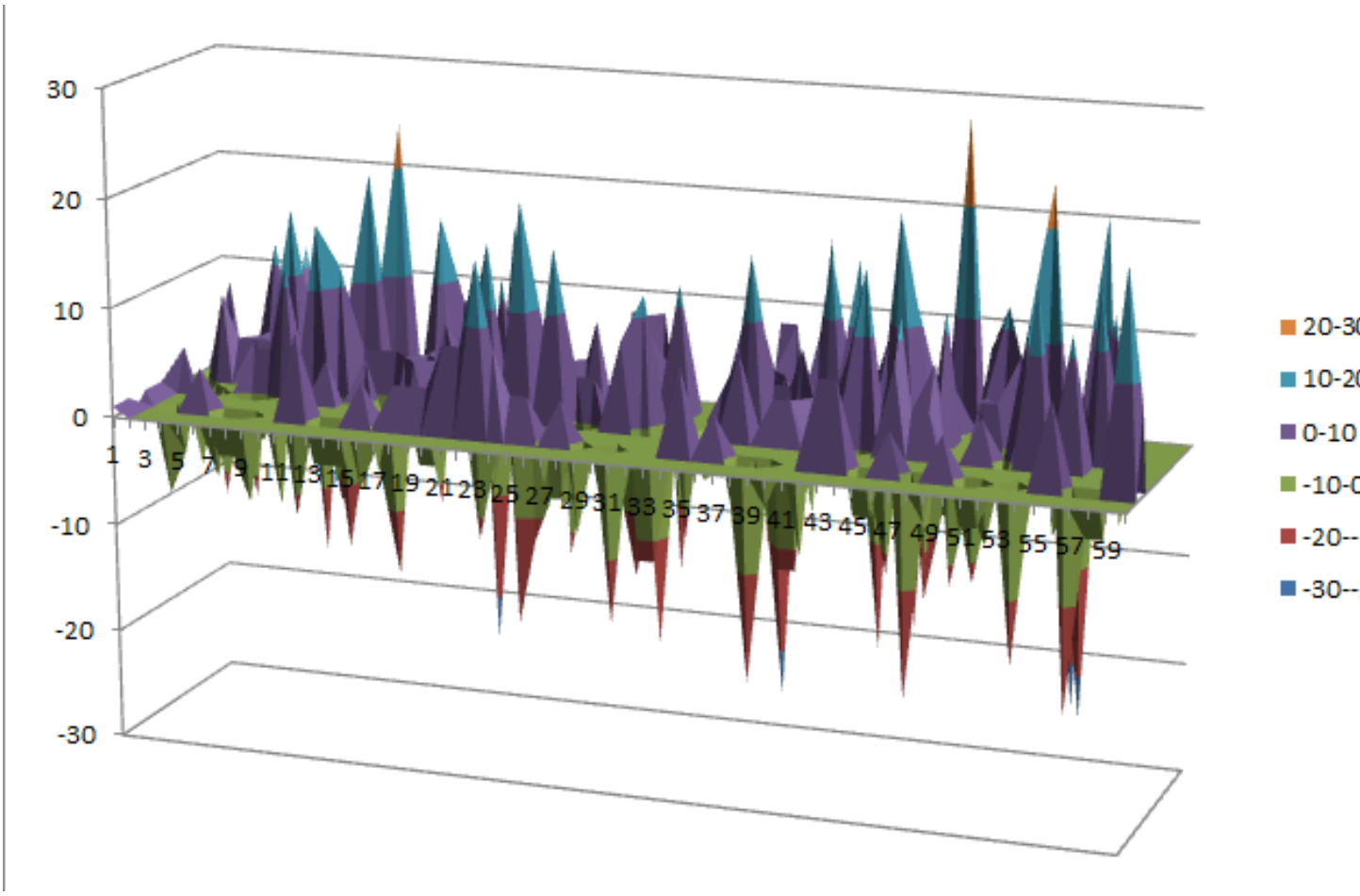
Truth Table

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

## 10.2 KASUMI

### 10.2.1 Description

KASUMI is a block cipher used in UMTS, GSM, and GPRS mobile communications systems. UMTS uses KASUMI in the confidentiality (f8) and integrity algorithms (f9) named UEA1 and UIA1, respectively. GSM employs KASUMI in the A5/3 key stream generator whereas GPRS does so in the GEA3 key stream generator. This cipher has two S-boxes: a 7x7 S-box called S7 and a 9x9 S-box called S9.

### 10.2.2 Summary

| S-box | size | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|---|
| S7 | 7x7 | 56 | 36 | 28 | 3 | 3 | 16 | 32768 | 0.015625 | 0.015625 |
| S9 | 9x9 | 240 | 0 | 0 | 2 | 2 | 512 | 524288 | 0.00390625 | 0.00390625 |

### 10.2.3 S7

#### Representations

Polynomial representation in ANF

Truth Table

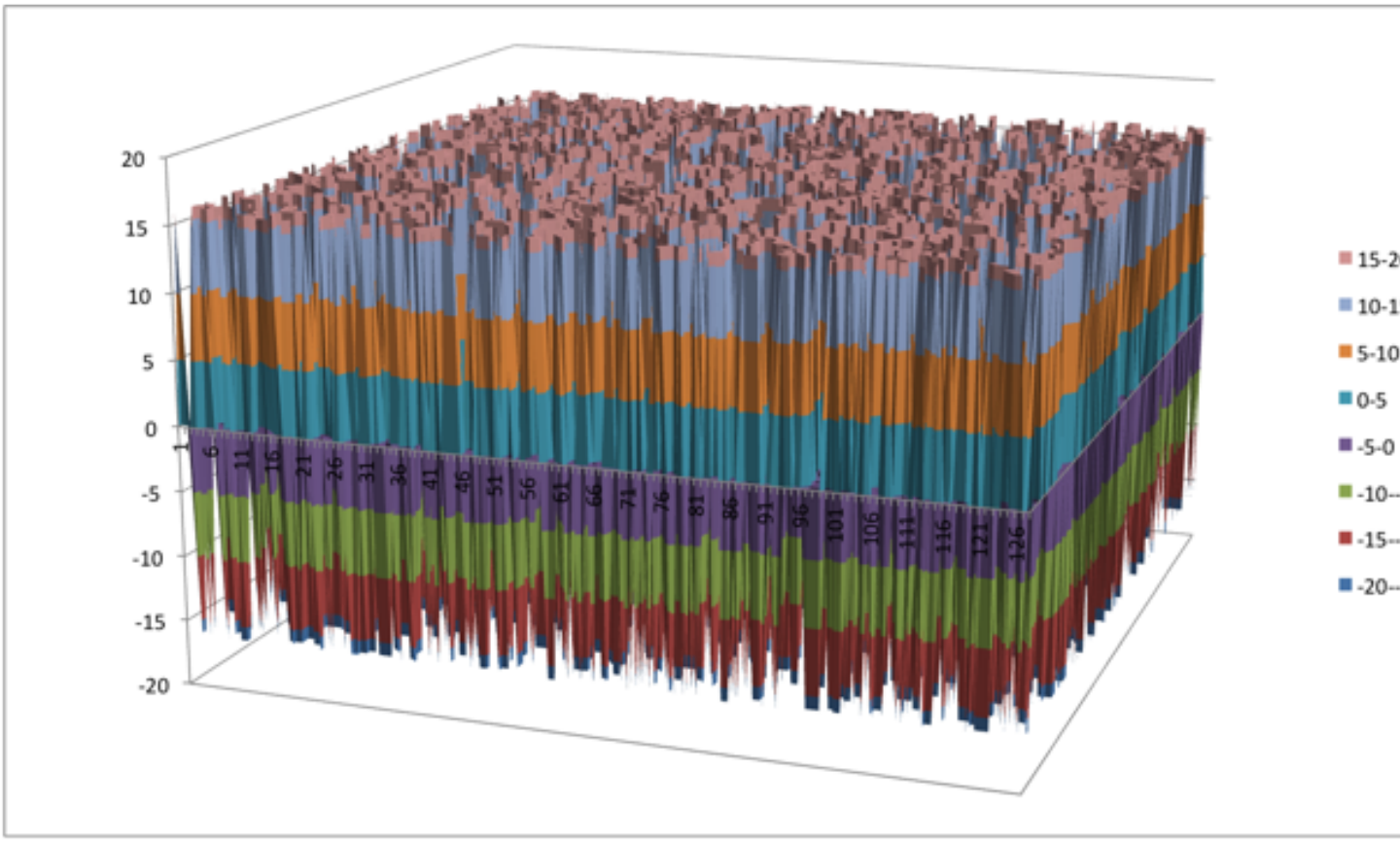
Decimal Representation

ANF Table

Characteristic function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 13 | 1 |
| 22 | 1 |
| 92 | 1 |

There are no linear structures

It has 1 fixed point: (0,0,1,1,0,1,1)

It has no negated fixed points

### 10.2.4 S9

### Representations

Polynomial representation in ANF

Truth Table

Decimal Representation

ANF Table

Characteristic function

Walsh Spectrum

Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 12 | 1 |
| 26 | 1 |
| 74 | 1 |
| 121 | 1 |
| 275 | 1 |

There are 511 linear structures:

Linear structures

It has 1 fixed point: (0,1,0,0,1,0,1,1,1)

It has 1 negated fixed points (1,0,0,0,1,1,0,0,0)

### 10.2.5 FI

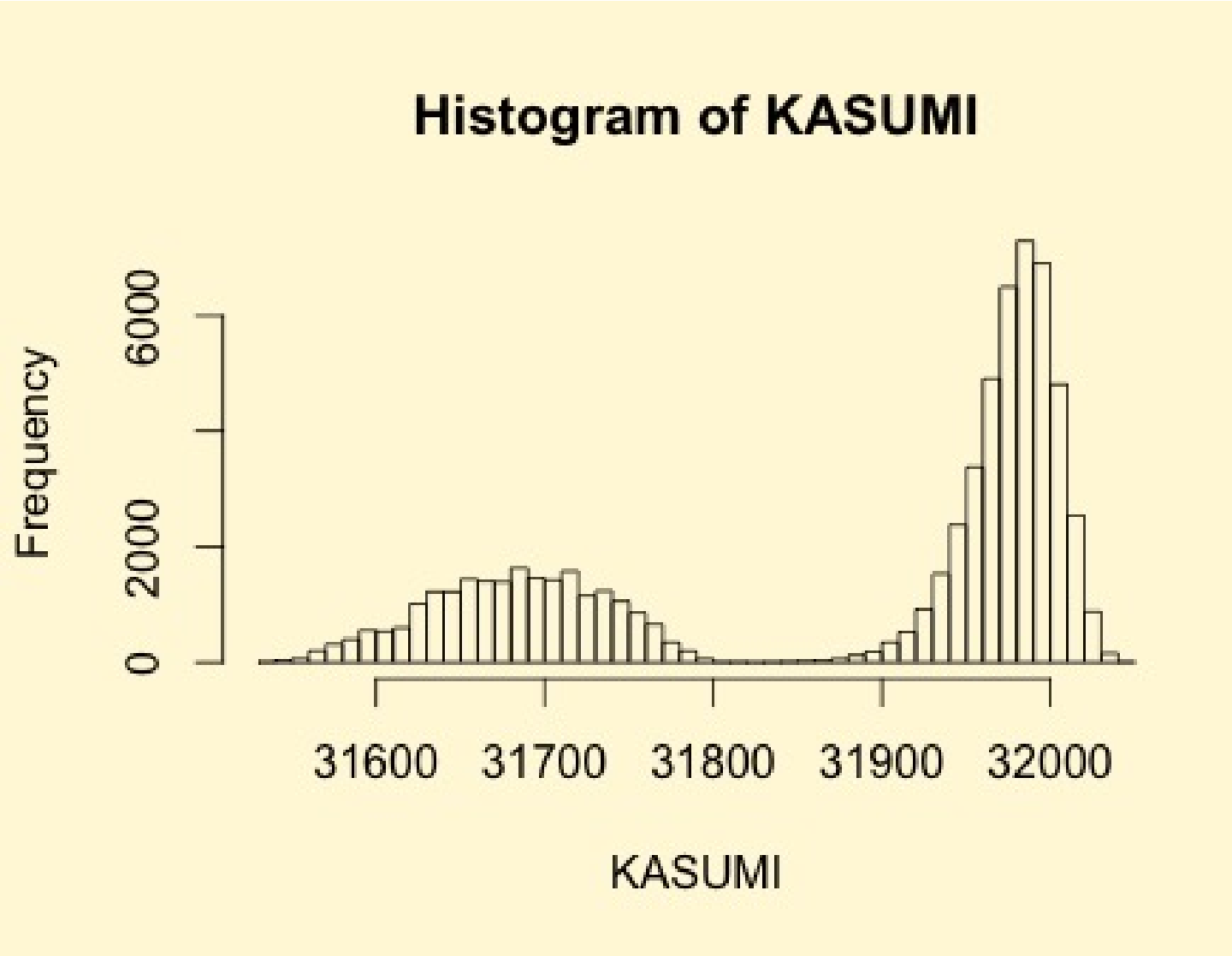
Algebraic degree from key 00000 to 65535

Cycle structure from key 00000 to 65535

Fixed and negated points from key 00000 to 65535

Nonlinearities from key 00000 to 65535

Graphical display of the distribution of the nonlinearities of FI:


Histogram of KASUMI
Frequency
0
2000
6000
31600
31700
31800
31900
32000
KASUMI

| Descriptive Statistics of FI nonlinearities | |
|---|---|
| Unique Values | 478 |
| Min | 31534 |
| Max | 32049 |
| Mean | 31878.7986 |
| Mean Deviation | 132.083019 |
| 1st Quartile | 31720 |
| Median | 31963 |
| 3rd Quartile | 31989 |
| Mode | 31995 |
| Range | 515 |
| Variance | 20879.009 |
| Standard Deviation | 144.4957 |
| Kkewness | -0.7622 |
| Kurtosis | -1.1463 |
| P0.5 | 31572 |
| P1 | 31582 |
| P5 | 31627 |
| P95 | 32011 |
| P99 | 32023 |
| P99.5 | 32027 |

## 10.3 MacGuffin

### 10.3.1 Description

MacGuffin is a block cipher uses a F-function which consists of the 8 S-boxes of the DES, but the two middle output bits of each S-box are neglected in order to obtain a 16-bit output. MacGuffin has eight 6x2 S-boxes: S1, S2, S3, S4, S5, S6, S7, S8.

### 10.3.2 Summary

| S-box | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S1 | 18 | 12 | 8 | 5 | 3 | 32 | 20224 | 0.19140625 | 0.53125 |
| S2 | 18 | 10 | 6 | 5 | 3 | 40 | 19840 | 0.19140625 | 0.5625 |
| S3 | 18 | 10 | 4 | 5 | 3 | 48 | 21376 | 0.19140625 | 0.53125 |
| S4 | 16 | 8 | 0 | 4 | 2 | 64 | 25600 | 0.25 | 0.5 |
| S5 | 20 | 12 | 8 | 5 | 3 | 32 | 16000 | 0.140625 | 0.5 |
| S6 | 20 | 10 | 4 | 5 | 3 | 48 | 19456 | 0.140625 | 0.4375 |
| S7 | 18 | 10 | 4 | 5 | 3 | 48 | 23296 | 0.19140625 | 0.53125 |
| S8 | 20 | 10 | 6 | 5 | 3 | 40 | 14848 | 0.140625 | 0.5 |

### 10.3.3 S1

#### Representations

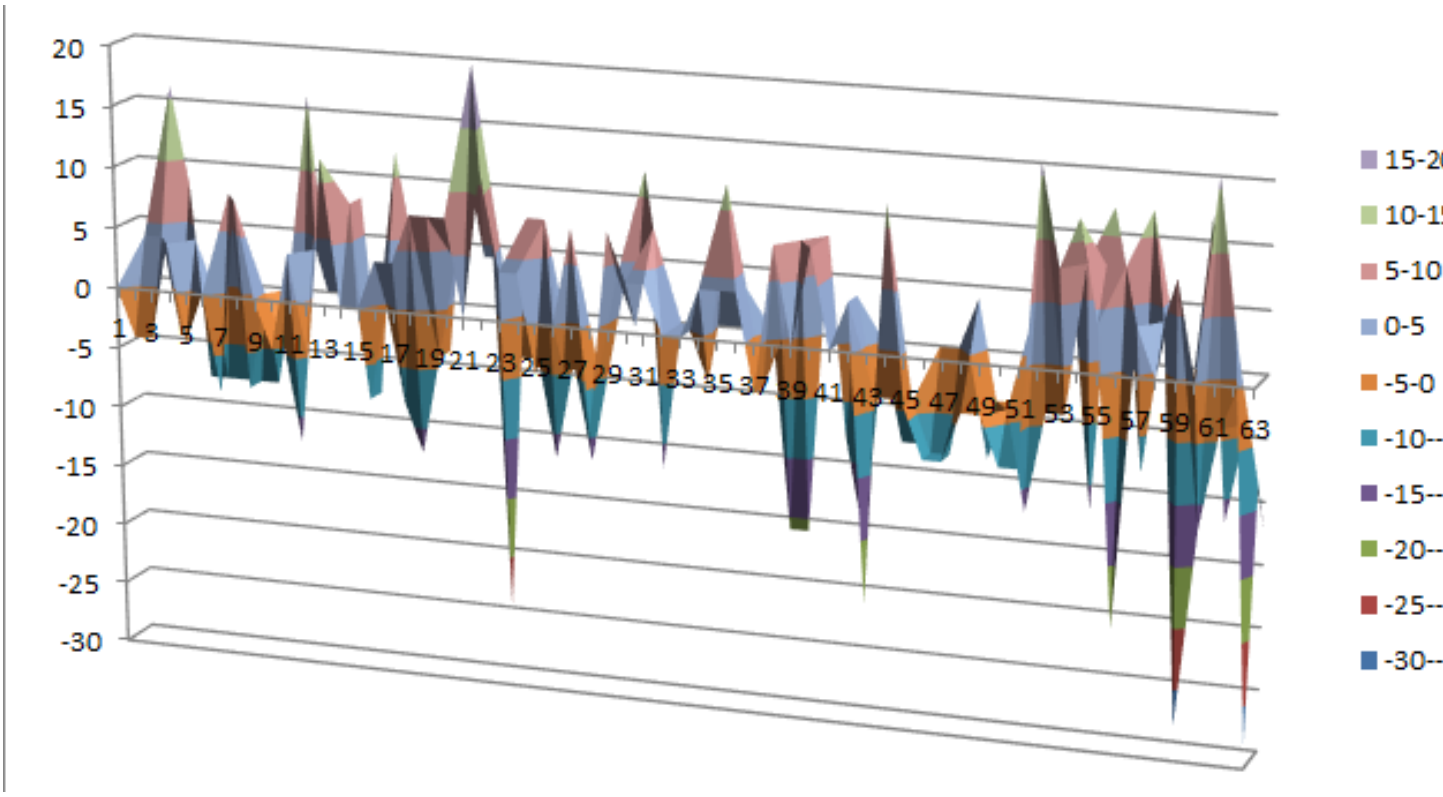
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.3.4 S2

### Representations

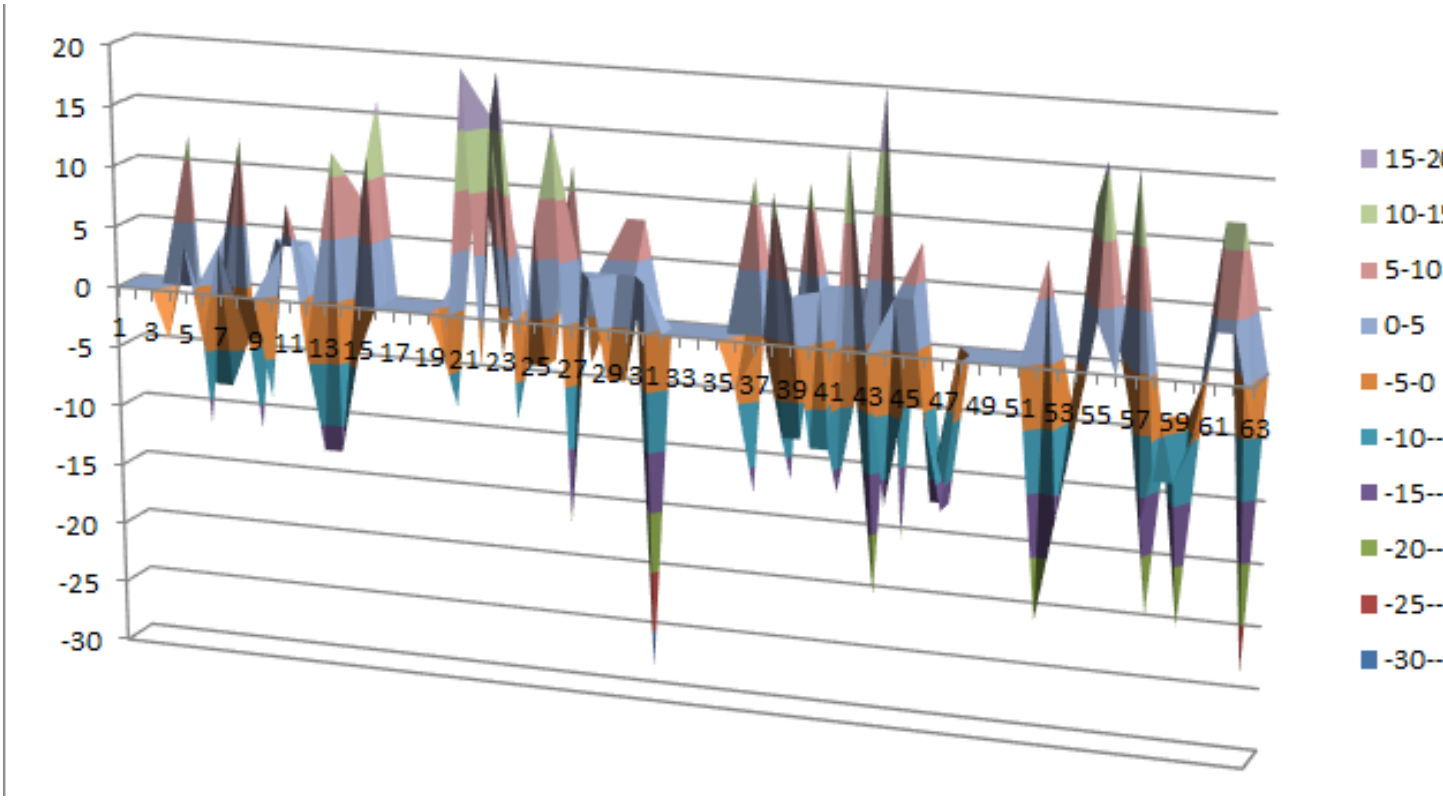
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.3.5 S3

### Representations

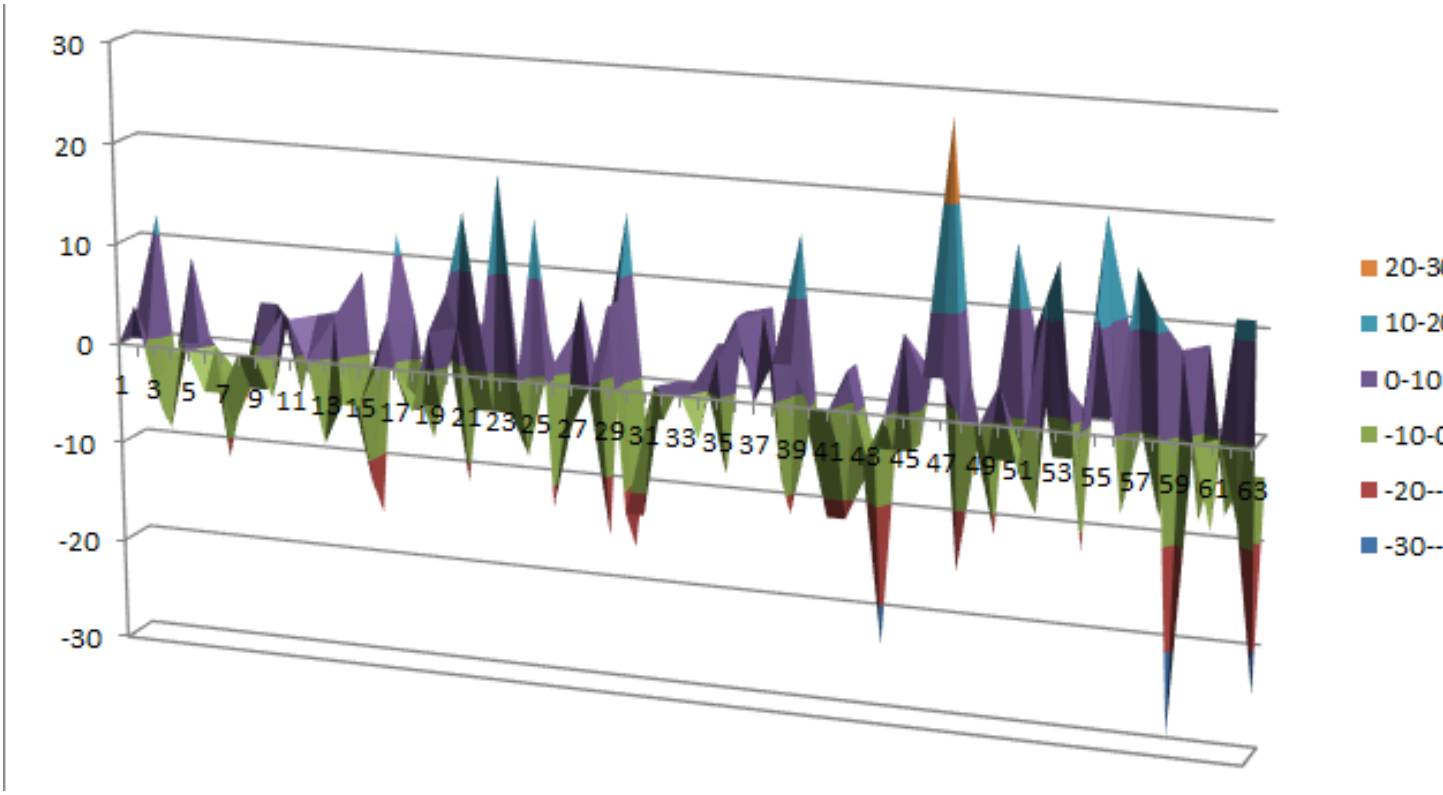
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.3.6 S4

### Representations

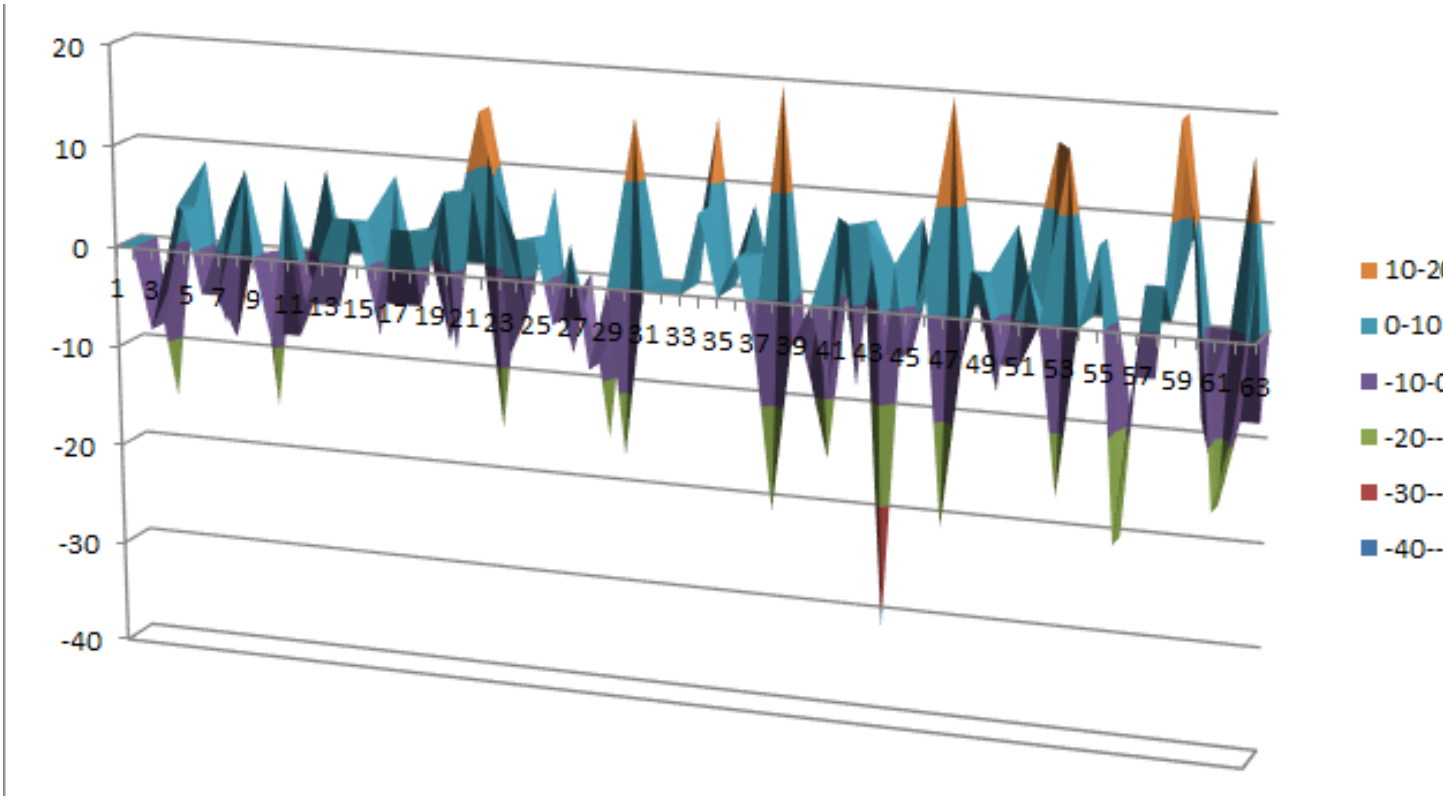
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are 1 linear structure:

```
([1 0 1 1 1 1],[1 1])
```

### 10.3.7 S5

### Representations

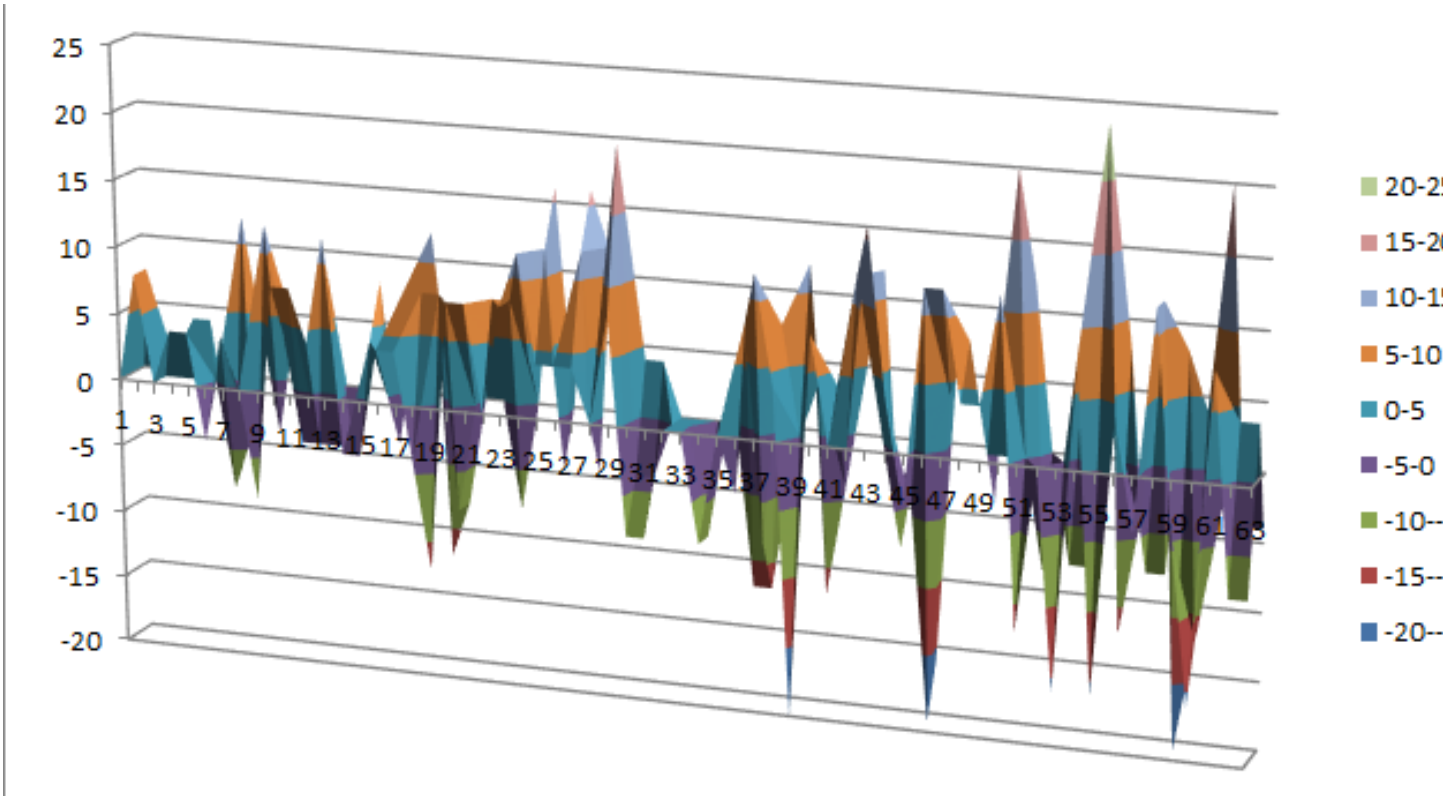
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.3.8 S6

### Representations

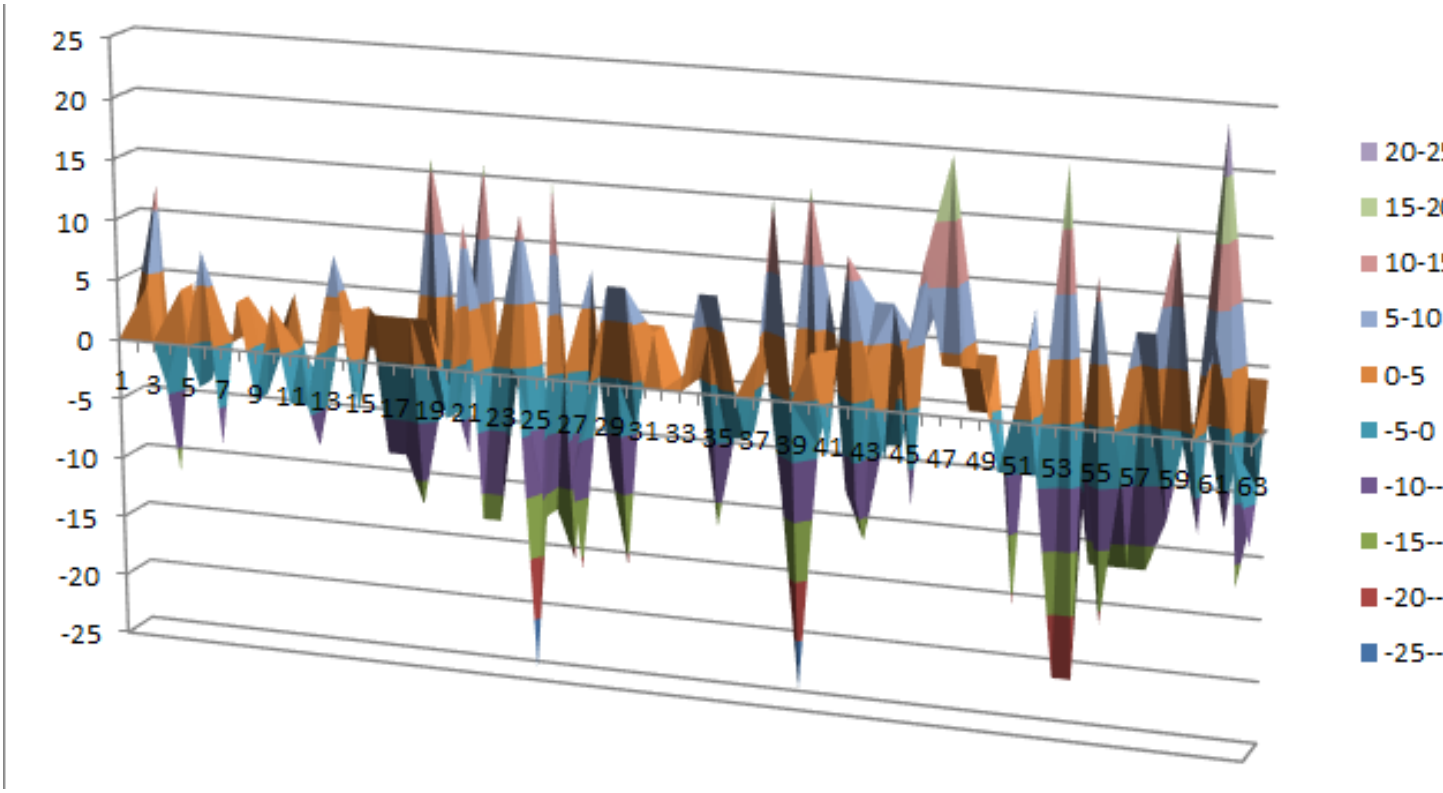
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.3.9 S7

### Representations

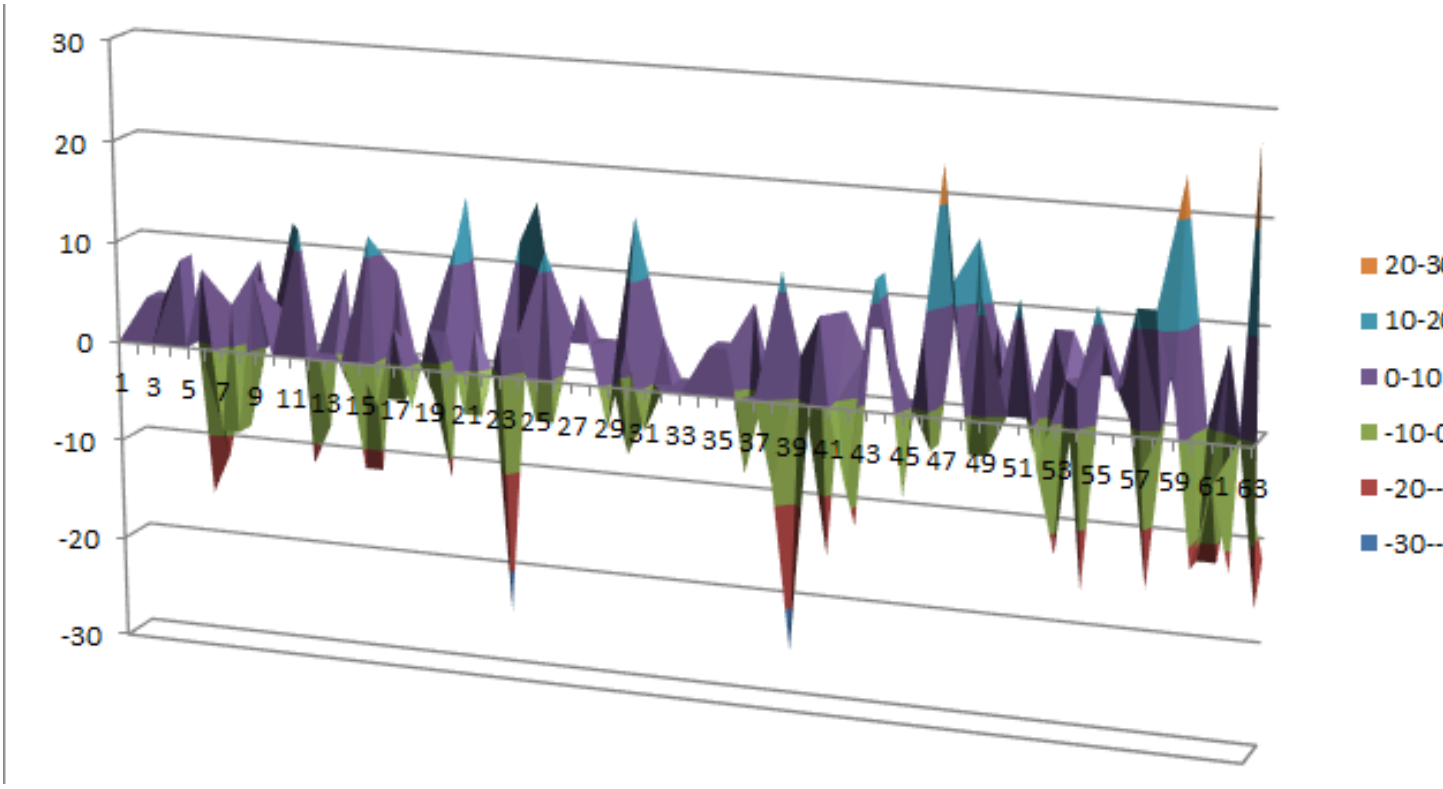
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

### 10.3.10 S8

### Representations

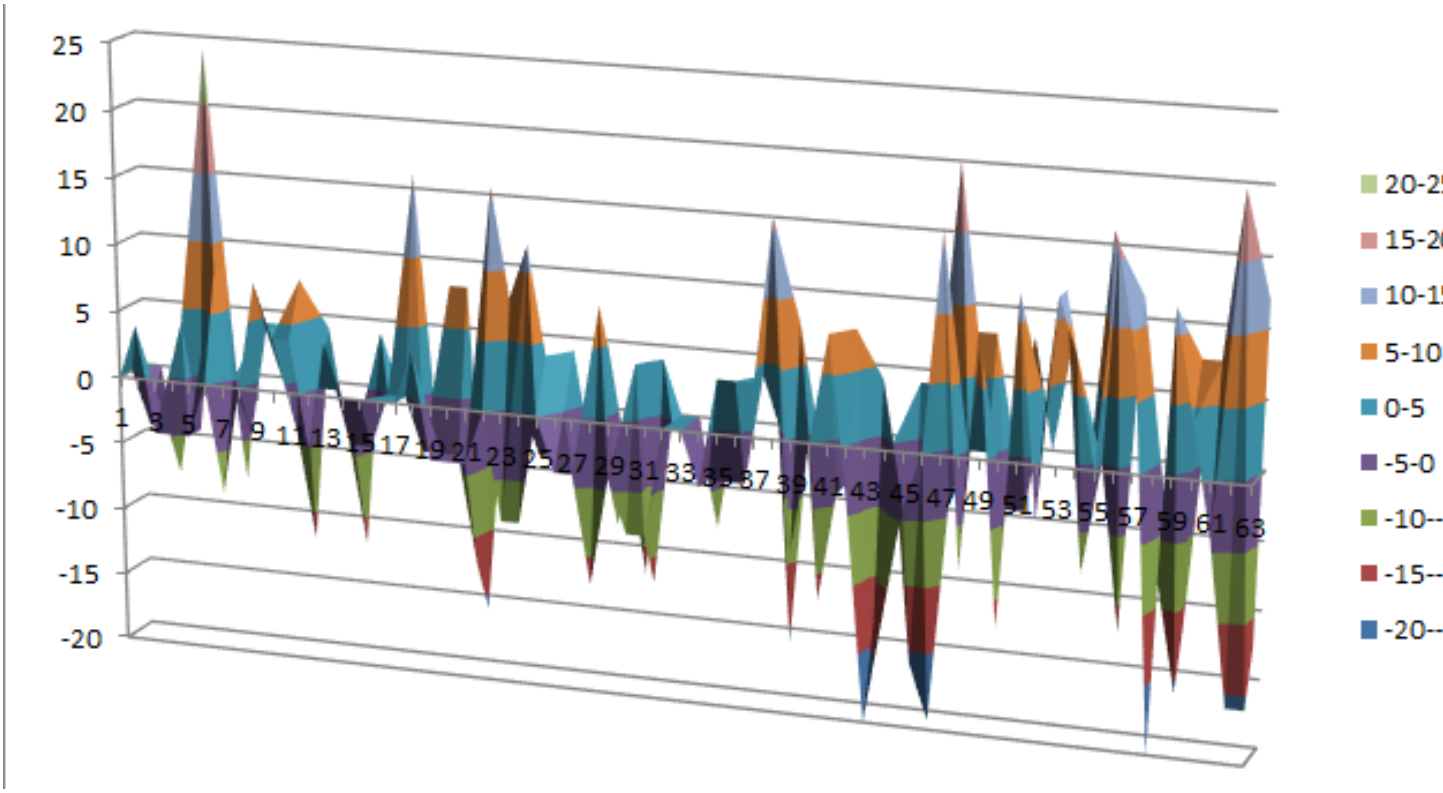
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

There are no linear structures

## 10.4 Mini-AES

### 10.4.1 Description

Raphael Chung-Wei Phan presented a version of the AES [Phan02miniadvanced], with all the parameters significantly reduced while preserving its original structure. This Mini version is purely educational and is designed to grasp the underlying concepts of Rijndael-like ciphers. It may also serve as a testbed for starting cryptanalysts to experiment with various cryptanalytic attacks. The Mini-AES cipher is a $16 \times 16$ vector Boolean function and the Mini-AES encryption is performed with a secret key of 16 bits.

The Mini-AES S-box is called *NibbleSub*, and defines a simple operation that substitutes each input with an output according to a 4x4 substitution table (S-box) given in the table below. These values are, in fact, taken from the first row of the first S-box in DES.

| NibbleSub Truth Table | |
|---|---|
| Input | Output |
| 0000 | 1110 |
| 0001 | 0100 |
| 0010 | 1101 |
| 0011 | 0001 |
| 0100 | 0010 |
| 0101 | 1111 |
| 0110 | 1011 |
| 0111 | 1000 |
| 1000 | 0011 |
| 1001 | 1010 |
| 1010 | 0110 |
| 1011 | 1100 |
| 1100 | 0101 |
| 1101 | 1001 |
| 1110 | 0000 |
| 1111 | 0111 |

The inverse of the previous table is easily computed by interchanging the input nibble with the output nibble, and then resorting it based on the new input nibble, as given in the table below.

| NibbleSubInv Truth Table | |
|---|---|
| Input | Output |
| 0000 | 1110 |
| 0001 | 0011 |
| 0010 | 0100 |
| 0011 | 1000 |
| 0100 | 0001 |
| 0101 | 1100 |
| 0110 | 1010 |
| 0111 | 1111 |
| 1000 | 0111 |
| 1001 | 1101 |
| 1010 | 1001 |
| 1011 | 0110 |
| 1100 | 1011 |
| 1101 | 0010 |
| 1110 | 0000 |
| 1111 | 0101 |

### 10.4.2 Summary

| S-box | size | *NL* | *NL2* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|---|
| NibbleSub | 4x4 | 2 | 0 | 0 | 2 | 2 | 16 | 1408 | 0.5625 | 0.5 |
| NibbleSubInv | 4x4 | 2 | 0 | 0 | 2 | 2 | 16 | 11408 | 0.5625 | 0.5 |
| MixColumns | 8x8 | 0 | • | 0 | 1 | 1 | 256 | 16777216 | 1 | 1 |

### 10.4.3 NibbleSub

#### Representations

Polynomial representation in ANF

Truth Table

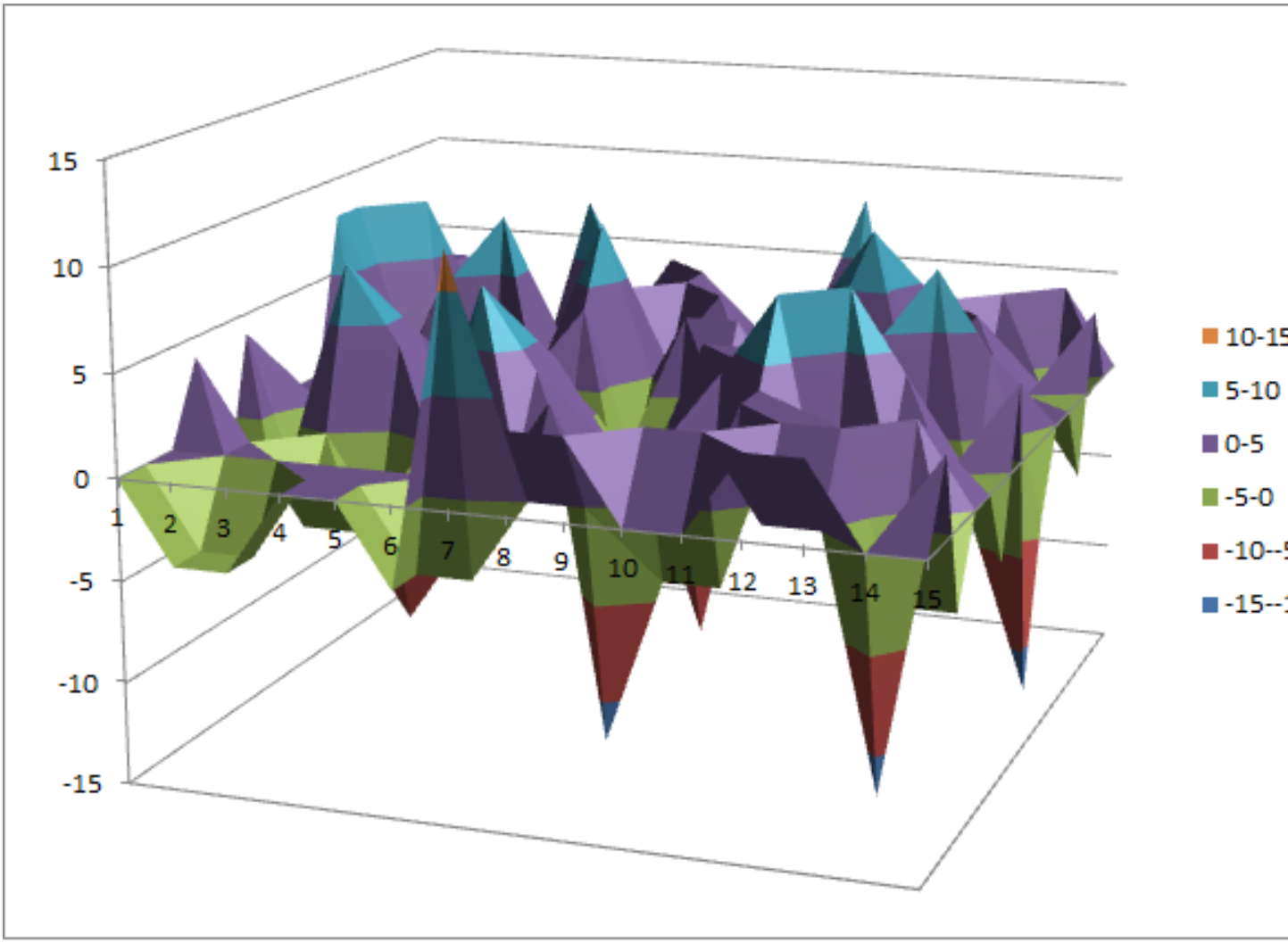
ANF Table

Characteristic Function

Walsh Spectrum

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | -4 | -4 | 0 | 0 | -4 | 12 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 |
| 0 | 0 | -4 | -4 | 0 | 0 | -4 | -4 | 0 | 0 | 4 | 4 | 0 | 0 | -12 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | -12 | -4 | -4 | 4 | 4 | -4 | -4 |
| 0 | 4 | 0 | -4 | -4 | -8 | -4 | 0 | 0 | -4 | 0 | 4 | 4 | -8 | 4 | 0 |
| 0 | -4 | -4 | 0 | -4 | 0 | 8 | 4 | -4 | 0 | -8 | 4 | 0 | -4 | -4 | 0 |
| 0 | 4 | -4 | 8 | 4 | 0 | 0 | 4 | 0 | -4 | 4 | 8 | -4 | 0 | 0 | -4 |
| 0 | -4 | 0 | 4 | 4 | -8 | 4 | 0 | -4 | 0 | 4 | 0 | 8 | 4 | 0 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -4 | 4 | 4 | -4 | 4 | -4 | -4 | -12 |
| 0 | 0 | -4 | -4 | 0 | 0 | -4 | -4 | -8 | 0 | -4 | 4 | 0 | 8 | 4 | -4 |
| 0 | 8 | -4 | 4 | -8 | 0 | 4 | -4 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 |
| 0 | 8 | 0 | -8 | 8 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | -4 | 8 | -4 | -4 | 0 | 4 | 0 | 4 | 0 | 4 | 8 | 0 | 4 | 0 | -4 |
| 0 | 4 | 4 | 0 | -4 | 8 | 0 | 4 | -8 | -4 | 4 | 0 | 4 | 0 | 0 | 4 |
| 0 | 4 | 4 | 0 | -4 | -8 | 0 | 4 | -4 | 0 | 0 | -4 | -8 | 4 | -4 | 0 |
| 0 | -4 | -8 | -4 | -4 | 0 | 4 | 0 | 0 | -4 | 8 | -4 | -4 | 0 | 4 | 0 |

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 14 | 1 |

There are 7 linear structures:

```
([0 0 1 1],[0 1 1 1])
([0 1 0 0],[0 1 0 1])
([1 0 1 0],[1 1 1 1])
```

(continues on next page)

(continued from previous page)

```
([1 0 1 1],[0 1 0 1])
([1 1 0 1],[1 0 0 1])
([1 1 1 0],[1 1 1 0])
([1 1 1 1],[0 1 0 1])
```

It has no fixed points and 2 negated fixed points: (0,0,1,0), (0,1,1,1)

### 10.4.4 NibbleSubInv

#### Representations

Polynomial representation in ANF

Truth Table

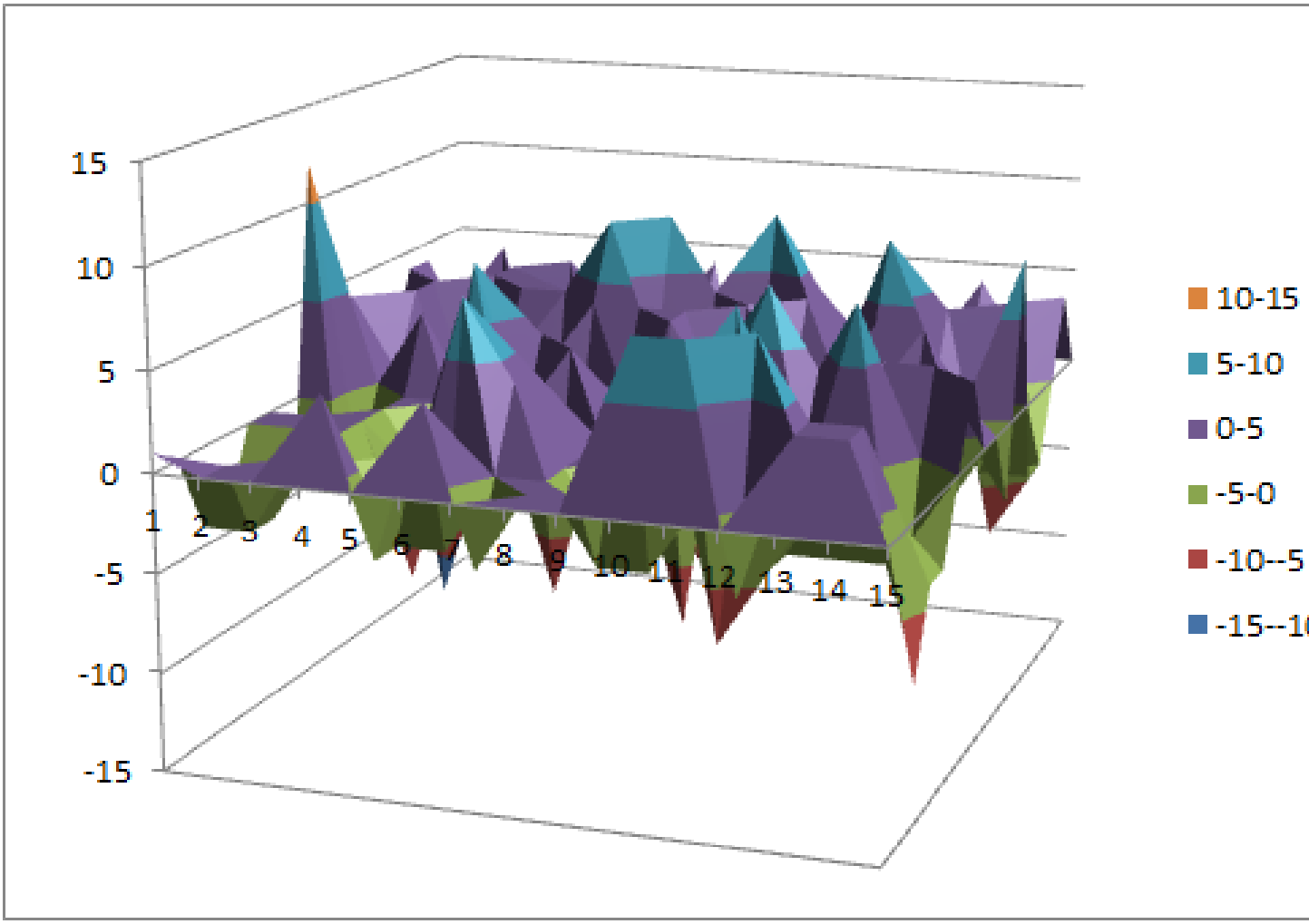
ANF Table

Characteristic Function

Walsh Spectrum

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 4 | -4 | 4 | -4 | 0 | 0 | 8 | 8 | -4 | 4 | 4 | -4 |
| 0 | -4 | -4 | 0 | 0 | -4 | -4 | 0 | 0 | -4 | -4 | 0 | 8 | 4 | 4 | -8 |
| 0 | -4 | -4 | 0 | -4 | 0 | 8 | 4 | 0 | -4 | 4 | -8 | -4 | 0 | 0 | -4 |
| 0 | 0 | 0 | 0 | -4 | -4 | 4 | 4 | 0 | 0 | -8 | 8 | -4 | -4 | -4 | -4 |
| 0 | 0 | 0 | 0 | -8 | 0 | 0 | -8 | 0 | 0 | 0 | 0 | 0 | 8 | -8 | 0 |
| 0 | -4 | -4 | 0 | -4 | 8 | 0 | 4 | 0 | -4 | 4 | 8 | 4 | 0 | 0 | 4 |
| 0 | 12 | -4 | 0 | 0 | 4 | 4 | 0 | 0 | -4 | -4 | 0 | 0 | 4 | 4 | 0 |
| 0 | 4 | 0 | 4 | 0 | -4 | 0 | -4 | -4 | -8 | 4 | 0 | 4 | -8 | -4 | 0 |
| 0 | 4 | 0 | -12 | -4 | 0 | -4 | 0 | 4 | 0 | 4 | 0 | 0 | -4 | 0 | -4 |
| 0 | 0 | 4 | -4 | 0 | -8 | 4 | 4 | 4 | -4 | 0 | 0 | 4 | 4 | 0 | 8 |
| 0 | 0 | 4 | -4 | 4 | 4 | 8 | 0 | -4 | 4 | 0 | 0 | 8 | 0 | -4 | -4 |
| 0 | 4 | 0 | 4 | 4 | 0 | -4 | 8 | 4 | 0 | 4 | 0 | 0 | 4 | -8 | -4 |
| 0 | 4 | 0 | 4 | -8 | -4 | 0 | 4 | -4 | 8 | 4 | 0 | 4 | 0 | 4 | 0 |
| 0 | 0 | -12 | -4 | 4 | -4 | 0 | 0 | -4 | 4 | 0 | 0 | 0 | 0 | -4 | 4 |
| 0 | 0 | 4 | -4 | 0 | 0 | -4 | 4 | -12 | -4 | 0 | 0 | -4 | 4 | 0 | 0 |

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 2 | 1 |
| 14 | 1 |

There are 7 linear structures:

```
([0 0 1 0],[0 0 1 0])
([0 1 0 1],[1 0 1 1])
([1 0 0 0],[0 0 1 1])
([1 0 0 0],[1 0 0 0])
([1 0 0 0],[1 0 1 1])
```

(continues on next page)

(continued from previous page)

```
([1 0 1 0],[0 0 0 1])
([1 1 0 1],[1 0 1 1])
```

It has no fixed points and 2 negated fixed points: (1,0,0,0), (1,1,0,1)

### 10.4.5 MixColumn

#### Representations

Polynomial representation in ANF

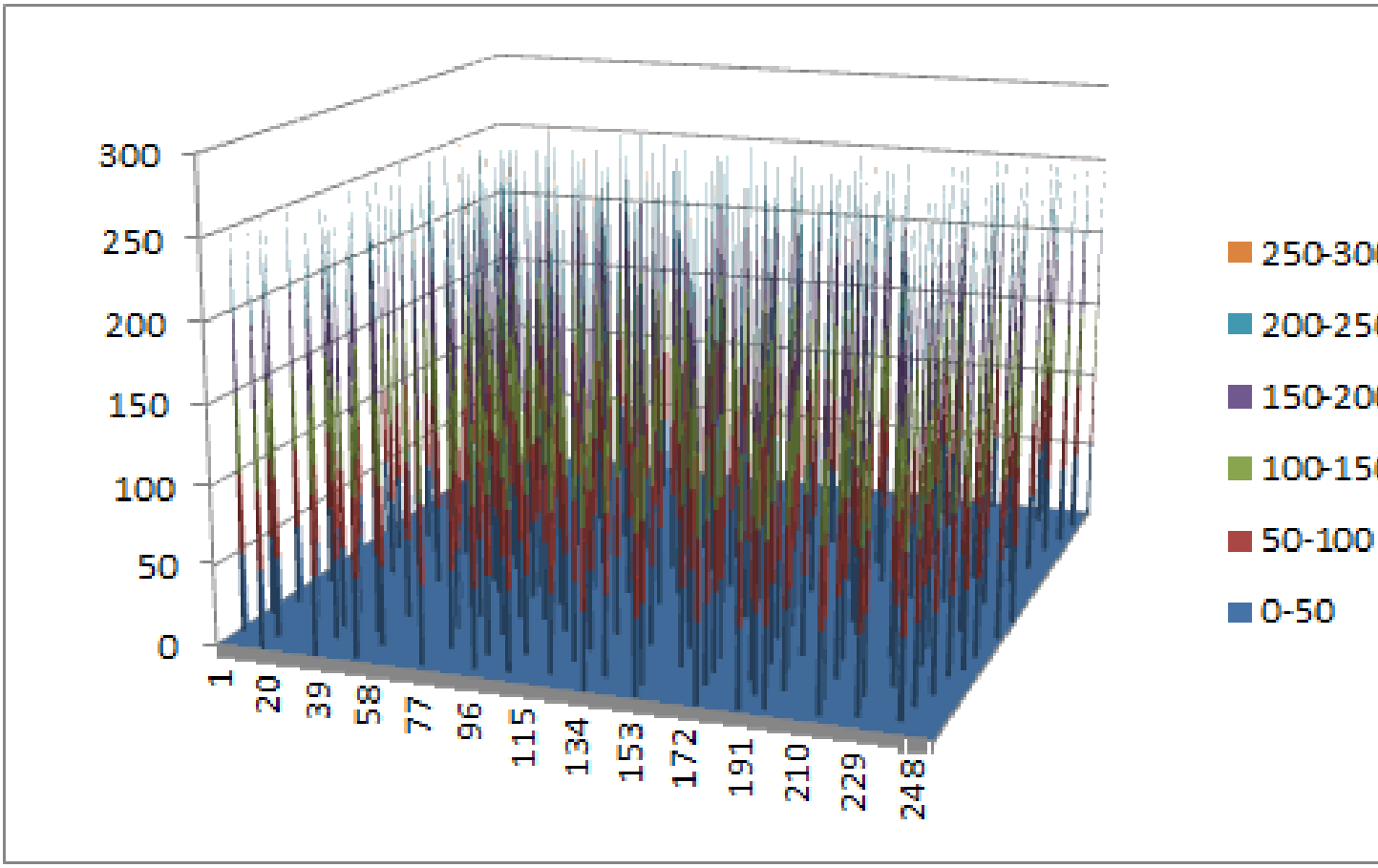
Truth Table

ANF Table

Characteristic Function

Walsh Spectrum

Walsh Spectrum representation (except first row and column):




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 16 |
| 2 | 120 |

There 65025 linear structures

It has 15 fixed points: (0,0,0,0,0,0,0,0), (0,0,0,1,0,0,0,1), (0,0,1,0,0,0,1,0), (0,0,1,1,0,0,1,1), (0,1,0,0,0,1,0,0), (0,1,0,1,0,1,0,1), (0,1,1,0,0,1,1,0), (0,1,1,1,0,1,1,1), (1,0,0,0,1,0,0,0), (1,0,0,1,1,0,0,1), (1,0,1,0,1,0,1,0), (1,0,1,1,1,0,1,1), (1,1,0,0,1,1,0,0), (1,1,0,1,1,1,0,1), (1,1,1,0,1,1,1,0)

It has 16 negated fixed points: (0,0,0,0,1,1,1,0), (0,0,0,1,1,1,1,1), (0,0,1,0,1,1,0,0), (0,0,1,1,1,1,0,1), (0,1,0,0,1,0,1,0), (0,1,0,1,1,0,1,1), (0,1,1,0,1,0,0,0), (0,1,1,1,1,0,0,1), (1,0,0,0,0,1,1,0), (1,0,0,1,0,1,1,1), (1,0,1,0,0,1,0,0), (1,0,1,1,0,1,0,1), (1,1,0,0,0,0,1,0), (1,1,0,1,0,0,1,1), (1,1,1,0,0,0,0,0), (1,1,1,1,0,0,0,1)

### 10.4.6 ks0

### Representations

Polynomial representation in ANF

Truth Table

ANF Table

Walsh Spectrum (each row represents a column of Walsh Spectrum)

Linear Profile (each row represents a column of Linear Profile)

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 65536 |

### 10.4.7 ks1

### Representations

Polynomial representation in ANF

Truth Table

ANF Table

Walsh Spectrum (each row represents a column of Walsh Spectrum)

Linear Profile (each row represents a column of Linear Profile)

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 5 | 2 |
| 10 | 2 |
| 28 | 1 |
| 60 | 1 |
| 1223 | 1 |
| 26097 | 1 |
| 38097 | 1 |

## 10.4.8 ks2

### Representations

Polynomial representation in ANF:

f1

f2

f3

f4

f5

f6

f7

f8

f9

f10

f11

f12

f13

f14

f15

f16

Truth Table

ANF Table

Walsh Spectrum (each row represents a column of Walsh Spectrum)

Linear Profile (each row represents a column of Linear Profile)

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 1 |
| 12 | 1 |
| 15 | 3 |
| 30 | 1 |
| 109 | 1 |
| 385 | 1 |
| 831 | 1 |
| 2472 | 1 |
| 3617 | 1 |
| 9775 | 1 |
| 16777 | 1 |
| 31482 | 1 |

### 10.4.9 mini-AES

Algebraic degree from key 00000 to 65535 is equal to 14

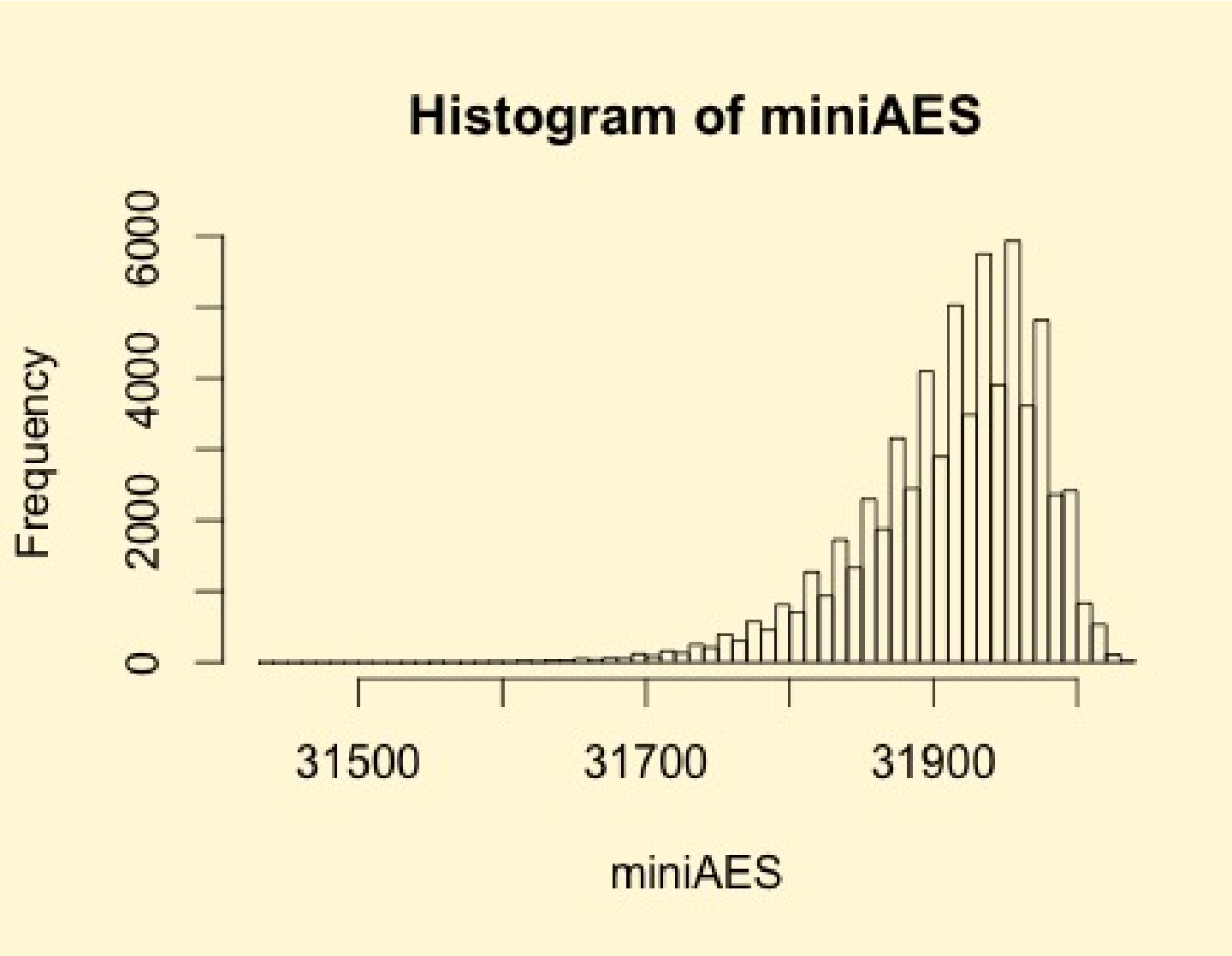
Cycle structure from key 00000 to 65535

Fixed and negated points from key 00000 to 65535

Nonlinearities from key 00000 to 65535

Nonlinearities in ascendent order

Graphical display of the distribution of the nonlinearities of mini-AES:


Histogram of miniAES
Frequency
0
2000
4000
6000
31500
31700
31900
miniAES

| Descriptive Statistics of mini-AES nonlinearities | |
|---|---|
| Unique Values | 130 |
| Min | 31432 |
| Max | 32040 |
| Mean | 31912.9894 |
| Mean Deviation | 8.6571 |
| 1st Quartile | 31880 |
| Median | 31924 |
| 3rd Quartile | 31960 |
| Mode | 31952 |
| Range | 608 |
| Variance | 3903.8642 |
| Standard Deviation | 62.4809 |
| Kkewness | -1.092059 |
| Kurtosis | 1.79284 |
| P0.5 | 31692 |
| P1 | 31720 |
| P5 | 31796 |
| P95 | 31992 |
| P99 | 32012 |
| P99.5 | 32016 |

## 10.5 Square

### 10.5.1 Description

*Square* is a block cipher invented by Joan Daemen and Vincent Rijmen. The design, published in 1997, is a forerunner to Rijndael, which has been adopted as the Advanced Encryption Standard. It has a 8x8 S-box called S.

### 10.5.2 Summary

| S-box | size | *NL* | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ | *LP* | *DP* |
|---|---|---|---|---|---|---|---|---|---|
| S | 8x8 | 112 | 56 | 7 | 4 | 32 | 133120 | 0.015625 | 0.015625 |

### 10.5.3 S

#### Representations

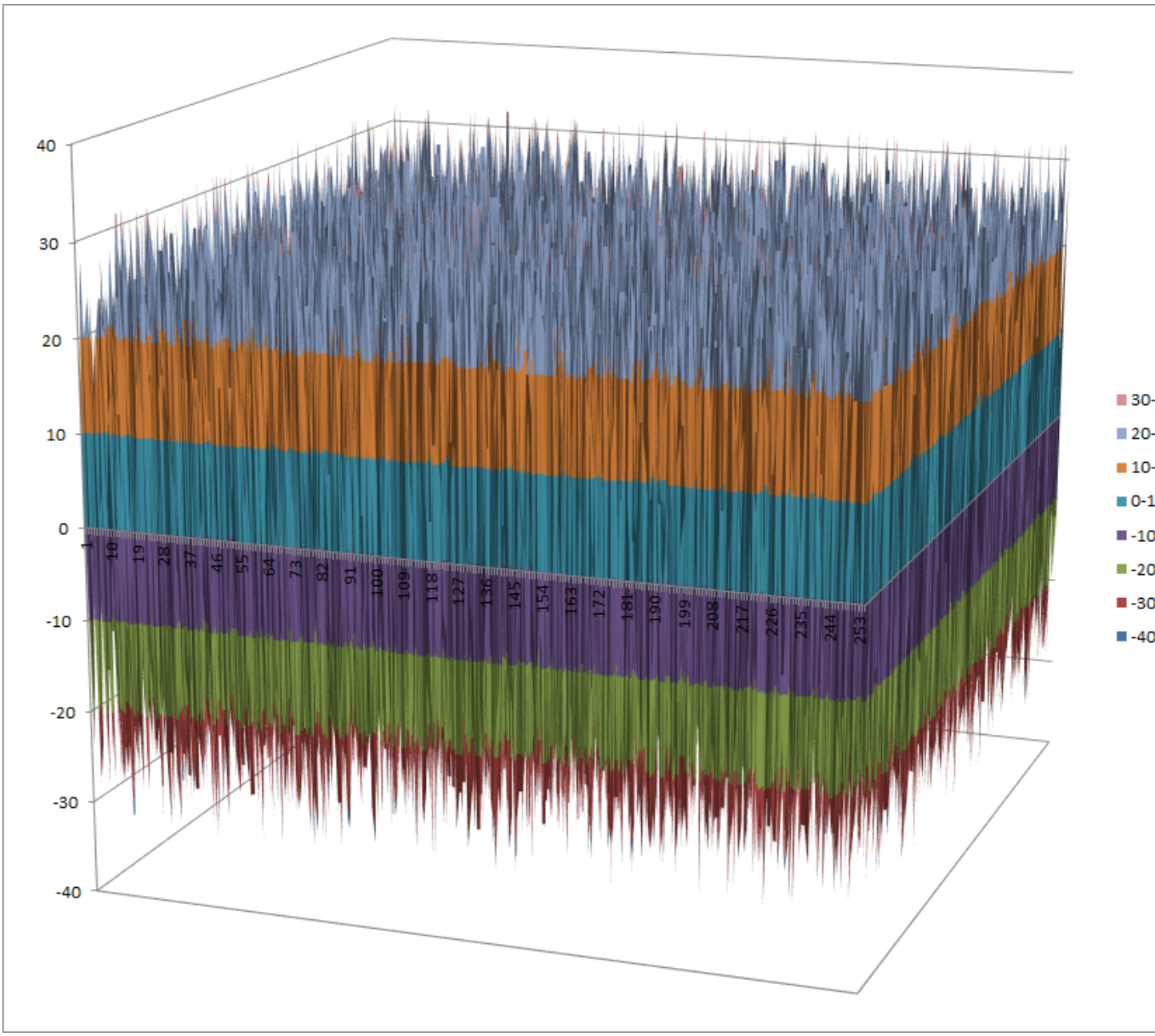
Polynomial representation in ANF

Truth Table

ANF Table

Characteristic function

Walsh Spectrum




Linear Profile

Differential Profile

Autocorrelation Spectrum

### Other useful information in cryptanalysis

Cycle structure:

| Cycle length | Number of cycles |
|---|---|
| 1 | 2 |
| 6 | 1 |
| 8 | 1 |
| 20 | 1 |
| 23 | 1 |
| 197 | 1 |

There are no linear structures

It has 1 fixed point: (0,0,0,0,1,1,0,0)

It has 1 negated fixed point: (1,0,0,1,0,1,1,1)

# 11 Design of cryptographically robust Vector Boolean functions



## 11.1 Bent functions

### 11.1.1 Two variables

There are 8 bent functions:

Polynomial representation in ANF

### 11.1.2 Four variables

There are 896 bent functions:

Polynomial representation in ANF

## 11.2 Optimization of Boolean functions via Genetic Algorithms

### 11.2.1 Description

## Galib

The Genetic Algorithm Library called GAlib was (straightforwardly) linked with our VBF library to perform a search of Boolean functions with good combined cryptographic criteria. In using the GAlib library we will work primarily with two classes: a genome and a genetic algorithm. Each genome instance represents a single solution to our optimization problem. The genetic algorithm object defines how the evolution should take place. The genetic algorithm uses an objective function to determine how 'fit' each genome is for survival. It uses the genome operators (built into the genome) and selection/replacement strategies (built into the genetic algorithm) to generate new individuals.

The following three items must be defined in order to solve an optimization problem using a genetic algorithm: A representation, the genetic operators and the objective function.

The *genetic algorithm* object determines which individuals should survive, which should reproduce, and which should die. It also records statistics and decides how long the evolution should continue. The algorithm updates the population of solutions over a number of iterations (or generations*). We have used the number of generations as a stopping measure. In each iteration a number of steps are involved:

1. Selection of parents from the current population of solutions.
2. Crossover of parents to produce offspring.
3. Mutation of the offspring.
4. Selection from the mutated offspring and the current population of solutions to determine the population of solutions for the next iteration.

Among the many different types of genetic algorithms offered by GAlib we have chosen the standard 'simple genetic algorithm' described by Goldberg in his book [Goldberg:1989]. This algorithm uses non-overlapping populations and optional elitism. Each generation the algorithm creates an entirely new population of individuals.

## Representation

When you use a genetic algorithm to solve an optimization problem, you must be able to represent a single solution to your problem in a single *data structure*. The genetic algorithm will create a population of solutions based on a sample data structure that you provide. The genetic algorithm then operates on the population to evolve the best solution. In GAlib, the sample data structure is called a GAGenome (some people refer to it as a chromosome). We have used a type of genome called GA2DBinaryStringGenome. This class is derived from the base GAGenome class and a data structure class which consists of a 2-dimensional array of Boolean with $2^n$ elements (the binary string is the Truth Table of the Boolean function).

## Genetic operators

Each genome has three primary *operators*: initialization, mutation, and crossover. With these operators you can bias an initial population, define a mutation or crossover specific to our representation, or evolve parts of the genetic algorithm as our population evolves.

The *initialization operator* determines how the genome is initialized. It is called when you initialize a population or the genetic algorithm. This operator does not actually create new genomes, rather it 'stuffs' the genomes with the primordial genetic material from which all solutions will evolve. We have used a uniform random initialization operator.

The *mutation operator* defines the procedure for mutating each genome. The mutation operation introduces randomness to the population of solutions. Mutation is generally applied to the children which result from the breeding process. We have used the typical mutator for a binary string genome which flips the bits in the string with a given probability (uniform random bit flip).

The *crossover operator* defines the procedure for generating a child from two parent genomes in order to obtain offspring. The crossover operation involves selecting two "parents" from the current population of solutions, picking a random point in the binary string representing each of the parents and swapping the values beyond that point between the two parents. This process results in two "children" with some characteristics of each of the parents.

### Weighted Objective Functions

In addition to the three primary operators, each genome must also contain an *Objective Function*. The Objective Function is used to evaluate the genome in order to know how good it is compared to the other genomes. Several objective functions were employed gradually involving more criteria in a weighted manner:

1. Nonlinearity of the Boolean function: $NL(f)$. This cryptographic criterion is represented by a locally smooth fitness function:

$$o_1 = NL(f)$$

2. The sum of the nonlinearity and linearity distance of the Boolean function, normalized with respect to their (a priori known) maximum values:

$$o_2 = \frac{NL(f)}{maxNL} + \frac{LD(f)}{maxLD}$$

where *maxNL* and *maxLD* are the maximum values of nonlinearity and linearity distance which can be achieved by a Boolean function with the same number of input variables as *f* respectively.

3. The sum of nonlinearity, algebraic degree and linearity distance of the Boolean function normalized with respect to their (a priori known) maximum values:

$$o_3 = \frac{NL(f)}{maxNL} + \frac{deg(f)}{maxDEG} + \frac{LD(f)}{maxLD}$$

where *maxNL*, *maxDEG* and *maxLD* are respectively the maximum values of nonlinearity, algebraic degree and linearity distance which can be achieved by a Boolean function with the same number of input variables as *f*.

4. The sum of nonlinearity, algebraic degree, algebraic immunity and linearity distance of the Boolean function normalized with respect to their (a priori known) maximum values:

$$o_4 = \frac{NL(f)}{maxNL} + \frac{deg(f)}{maxDEG} + \frac{AI(f)}{maxAI} + \frac{LD(f)}{maxLD}$$

where *maxNL*, *maxDEG*, *maxAI* and *maxLD* are respectively the maximum values of nonlinearity, algebraic degree, algebraic immunity and linearity distance which can be achieved by a Boolean function with the same number of input variables as *f*.

5. The sum of nonlinearity, algebraic degree, algebraic immunity and linearity distance of the Boolean function normalized with respect to their (a priori known) maximum values:

$$o_5 = \frac{NL(f)}{maxNL} + \frac{deg(f)}{maxDEG} + \frac{AI(f)}{maxAI} + \frac{LD(f)}{maxLD} + \frac{2^{3n} - \sigma(f)}{max\sigma - min\sigma}$$

where *maxNL*, *maxDEG*, *maxAI*, *maxLD*, $max\sigma$ are respectively the maximum values of nonlinearity, algebraic degree, algebraic immunity, linearity distance and sum-of-square indicator which can be achieved by a Boolean function with the same number of input variables as $f$; and $min\sigma$ is the minimum value of the sum-of-square indicator achievable by a Boolean function with the same number of input variables as *f*.

### 11.2.2 Implementation

In order to use Galib library, the installation instructions described in Installation Instructions for GAlib must be followed. Once Galib is installed, we can use this library in conjunction with VBF library by using the following Makefile:

```
include makevars

############# User settings #####################

# set your C++ compiler and options here...
GPP=g++
LIBS=-lntl
NTLINC= -I/usr/local/include -L/usr/local/lib
GA_LIB_DIR= ga
LIB_DIRS= -L$(GA_LIB_DIR)

############# End of user settings ###############

o1: o1.cpp VBF.h
        $(GPP) $(NTLINC) -Wall o1.cpp -o o1.exe $(LIBS) $(LIB_DIRS) -lga

o2: o2.cpp VBF.h
        $(GPP) $(NTLINC) -Wall o2.cpp -o o2.exe $(LIBS) $(LIB_DIRS)  -lga

o3: o3.cpp VBF.h
        $(GPP) $(NTLINC) -Wall o3.cpp -o o3.exe $(LIBS) $(LIB_DIRS) -lga

o4: o4.cpp VBF.h
        $(GPP) $(NTLINC) -Wall o4.cpp -o o4.exe $(LIBS) $(LIB_DIRS) -lga

o5: o5.cpp VBF.h
        $(GPP) $(NTLINC) -Wall o5.cpp -o o5.exe $(LIBS) $(LIB_DIRS) -lga

clean:
        rm -f *.exe
```

Note that a file called "makevars" must be present in the same directory as the Makefile file.

Example program for each of the objective functions could be the following:

Listing 1: o1.cpp

```
     #include <ga/GASimpleGA.h>      // we're going to use the simple GA
     #include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome
     #include <ga/std_stream.h>
     #include "VBF.h"

     #define cout STD_COUT

     float Objective(GAGenome &);    // This is the declaration of our obj function.

     int
     main(int argc, char **argv)
     {
       int n = 0, m = 0;
       int popsize;
       int ngen;
       float pmut;
       float pcross;

   // We generate random seed

       for(int ii=1; ii<argc; ii++) {
```

(continues on next page)

(continued from previous page)

```
      if(strcmp(argv[ii++],"seed") == 0) {
        GARandomSeed((unsigned int)atoi(argv[ii]));
      }
    }

    n = atoi(argv[1]);
    m = atoi(argv[2]);
    int width    = 1 << n;
    int height   = m;
    popsize  = atoi(argv[3]);
    ngen     = atoi(argv[4]);
    pmut   = atof(argv[5]);
    pcross = atof(argv[6]);

    GA2DBinaryStringGenome genome(width, height, Objective);

    GASimpleGA ga(genome);
    ga.populationSize(popsize);
    ga.nGenerations(ngen);
    ga.pMutation(pmut);
    ga.pCrossover(pcross);
    ga.evolve();

  // Now we print out the best genome that the GA found.

    cout << ga.statistics() << "\n";
    cout << ga.statistics().bestIndividual() << "\n";

    return 0;
  }

  float
  Objective(GAGenome& g) {

    using namespace VBFNS;

    VBF F;
    NTL::mat_GF2 T;
    NTL::RR nonlin;
    long spacen, m;

    GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
    float score=0.0;

    spacen = genome.width();
    m = genome.height();

    T.SetDims(spacen,m);
    for(int i=0; i<spacen; i++){
      for(int j=0; j<m; j++){
        T[i][j] = to_GF2(genome.gene(i,j));
      }
    }

    F.puttt(T);
    nonlin = nl(F);
    conv(score,nonlin);
```

(continues on next page)

(continued from previous page)

```
    return score;
  }
```

Listing 2: o2.cpp

```
  #include <ga/GASimpleGA.h>      // we're going to use the simple GA
  #include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome
  #include <ga/std_stream.h>
  #include "VBF.h"

  #define cout STD_COUT

  float Objective(GAGenome &);    // This is the declaration of our obj function.

  int
  main(int argc, char **argv)
  {
    int n = 0, m = 0;
    int popsize;
    int ngen;
    float pmut;
    float pcross;

// We generate random seed

    for(int ii=1; ii<argc; ii++) {
      if(strcmp(argv[ii++],"seed") == 0) {
        GARandomSeed((unsigned int)atoi(argv[ii]));
      }
    }

    n = atoi(argv[1]);
    m = atoi(argv[2]);
    int width    = 1 << n;
    int height   = m;
    popsize  = atoi(argv[3]);
    ngen     = atoi(argv[4]);
    pmut   = atof(argv[5]);
    pcross = atof(argv[6]);

    GA2DBinaryStringGenome genome(width, height, Objective);

    GASimpleGA ga(genome);
    ga.populationSize(popsize);
    ga.nGenerations(ngen);
    ga.pMutation(pmut);
    ga.pCrossover(pcross);
    ga.evolve();

  // Now we print out the best genome that the GA found.

    cout << ga.statistics() << "\n";
    cout << ga.statistics().bestIndividual() << "\n";

    return 0;
```

(continues on next page)

(continued from previous page)

```
}

float
Objective(GAGenome& g) {

  using namespace VBFNS;

  VBF F;
  NTL::mat_GF2 T;
  NTL::RR nonlin, lind, suma, nlm, ldm;
  long spacen, m;

  GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
  float score=0.0;

  spacen = genome.width();
  m = genome.height();

  T.SetDims(spacen,m);
  for(int i=0; i<spacen; i++){
    for(int j=0; j<m; j++){
      T[i][j] = to_GF2(genome.gene(i,j));
    }
  }

  F.puttt(T);
  nonlin = nl(F);
  lind = ld(F);
  nlm = nlmax(F);
  ldm = ldmax(F);

  suma = nonlin/nlm+lind/ldm;
  conv(score,suma);

  return score;
}
```

Listing 3: o3.cpp

```
#include <ga/GASimpleGA.h>      // we're going to use the simple GA
#include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome
#include <ga/std_stream.h>
#include "VBF.h"

#define cout STD_COUT

float Objective(GAGenome &);    // This is the declaration of our obj function.

int
main(int argc, char **argv)
{
  int n = 0, m = 0;
  int popsize;
  int ngen;
  float pmut;
  float pcross;
```

(continues on next page)

(continued from previous page)

```
// We generate random seed

    for(int ii=1; ii<argc; ii++) {
      if(strcmp(argv[ii++],"seed") == 0) {
        GARandomSeed((unsigned int)atoi(argv[ii]));
      }
    }

    n = atoi(argv[1]);
    m = atoi(argv[2]);
    int width    = 1 << n;
    int height   = m;
    popsize  = atoi(argv[3]);
    ngen     = atoi(argv[4]);
    pmut   = atof(argv[5]);
    pcross = atof(argv[6]);

    GA2DBinaryStringGenome genome(width, height, Objective);

    GASimpleGA ga(genome);
    ga.populationSize(popsize);
    ga.nGenerations(ngen);
    ga.pMutation(pmut);
    ga.pCrossover(pcross);
    ga.evolve();

  // Now we print out the best genome that the GA found.

    cout << ga.statistics() << "\n";
    cout << ga.statistics().bestIndividual() << "\n";

    return 0;
  }

  float
  Objective(GAGenome& g) {

    using namespace VBFNS;

    VBF F;
    NTL::mat_GF2 T;
    NTL::RR nonlin, lind, suma, nlm, ldm;
    long spacen, m;
    int d, n;

    GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
    float score=0.0;

    spacen = genome.width();
    m = genome.height();

    T.SetDims(spacen,m);
    for(int i=0; i<spacen; i++){
      for(int j=0; j<m; j++){
        T[i][j] = to_GF2(genome.gene(i,j));
      }
```

(continues on next page)

(continued from previous page)

```
    }

    F.puttt(T);
    nonlin = nl(F);
    lind = ld(F);
    nlm = nlmax(F);
    ldm = ldmax(F);
    d = deg(F);
    n = logtwo(spacen);

    suma = nonlin/nlm+lind/ldm+to_RR(d)/to_RR(n);
    conv(score,suma);

    return score;
  }
```

Listing 4: o4.cpp

```
  #include <ga/GASimpleGA.h>      // we're going to use the simple GA
  #include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome
  #include <ga/std_stream.h>
  #include "VBF.h"

  #define cout STD_COUT

  float Objective(GAGenome &);    // This is the declaration of our obj function.

  int
  main(int argc, char **argv)
  {
    int n = 0, m = 0;
    int popsize;
    int ngen;
    float pmut;
    float pcross;

// We generate random seed

    for(int ii=1; ii<argc; ii++) {
      if(strcmp(argv[ii++],"seed") == 0) {
        GARandomSeed((unsigned int)atoi(argv[ii]));
      }
    }

    n = atoi(argv[1]);
    m = atoi(argv[2]);
    int width    = 1 << n;
    int height   = m;
    popsize  = atoi(argv[3]);
    ngen     = atoi(argv[4]);
    pmut   = atof(argv[5]);
    pcross = atof(argv[6]);

    GA2DBinaryStringGenome genome(width, height, Objective);

    GASimpleGA ga(genome);
```

(continues on next page)

(continued from previous page)

```
    ga.populationSize(popsize);
    ga.nGenerations(ngen);
    ga.pMutation(pmut);
    ga.pCrossover(pcross);
    ga.evolve();

  // Now we print out the best genome that the GA found.

    cout << ga.statistics() << "\n";
    cout << ga.statistics().bestIndividual() << "\n";

    return 0;
  }

  float
  Objective(GAGenome& g) {

    using namespace VBFNS;

    VBF F;
    NTL::mat_GF2 T;
    NTL::RR nonlin, lind, suma, nlm, ldm;
    long spacen, m;
    int d, n, ai, maxai;

    GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
    float score=0.0;

    spacen = genome.width();
    m = genome.height();

    T.SetDims(spacen,m);
    for(int i=0; i<spacen; i++){
      for(int j=0; j<m; j++){
        T[i][j] = to_GF2(genome.gene(i,j));
      }
    }

    F.puttt(T);
    nonlin = nl(F);
    lind = ld(F);
    nlm = nlmax(F);
    ldm = ldmax(F);
    d = deg(F);
    n = logtwo(spacen);
    ai = AI(F);
    maxai = aimax(F);

    suma = nonlin/nlm+lind/ldm+to_RR(d)/to_RR(n)+to_RR(ai)/to_RR(maxai);
    conv(score,suma);

    return score;
  }
```

Listing 5: o5.cpp

```
#include <ga/GASimpleGA.h>      // we're going to use the simple GA
#include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome
#include <ga/std_stream.h>
#include "VBF.h"

#define cout STD_COUT

float Objective(GAGenome &);    // This is the declaration of our obj function.

int
main(int argc, char **argv)
{
  int n = 0, m = 0;
  int popsize;
  int ngen;
  float pmut;
  float pcross;

// We generate random seed

  for(int ii=1; ii<argc; ii++) {
    if(strcmp(argv[ii++],"seed") == 0) {
      GARandomSeed((unsigned int)atoi(argv[ii]));
    }
  }

  n = atoi(argv[1]);
  m = atoi(argv[2]);
  int width    = 1 << n;
  int height   = m;
  popsize  = atoi(argv[3]);
  ngen     = atoi(argv[4]);
  pmut   = atof(argv[5]);
  pcross = atof(argv[6]);

  GA2DBinaryStringGenome genome(width, height, Objective);

  GASimpleGA ga(genome);
  ga.populationSize(popsize);
  ga.nGenerations(ngen);
  ga.pMutation(pmut);
  ga.pCrossover(pcross);
  ga.evolve();

// Now we print out the best genome that the GA found.

  cout << ga.statistics() << "\n";
  cout << ga.statistics().bestIndividual() << "\n";

  return 0;
}

float
Objective(GAGenome& g) {
```

(continues on next page)

(continued from previous page)

```
        using namespace VBFNS;

        VBF F;
        NTL::mat_GF2 T;
        NTL::RR nonlin, lind, suma, nlm, ldm;
        NTL::ZZ s, smin, smax;
        long spacen, m;
        int d, n, ai, maxai;

        GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
        float score=0.0;

        spacen = genome.width();
        m = genome.height();

        T.SetDims(spacen,m);
        for(int i=0; i<spacen; i++){
          for(int j=0; j<m; j++){
            T[i][j] = to_GF2(genome.gene(i,j));
          }
        }

        F.puttt(T);
        nonlin = nl(F);
        lind = ld(F);
        nlm = nlmax(F);
        ldm = ldmax(F);
        d = deg(F);
        n = logtwo(spacen);
        ai = AI(F);
        maxai = aimax(F);
        s = sigma(F);
        smin = sigmamin(F);
        smax = sigmamax(F);

        suma = nonlin/nlm+lind/ldm+to_RR(d)/to_RR(n)+to_RR(ai)/to_RR(maxai)+1.0-(to_
↪RR(s-smin)/to_RR(smax-smin));
        conv(score,suma);

        return score;
      }
```

The following program illustrates how can be set the seed (included in a file called "seed.bin" with the binary representation of the Truth Table of the function) of the Genetic algorithm for the o4 objective function:

```
#include <ga/GASimpleGA.h>      // we're going to use the simple GA
#include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome
#include <ga/std_stream.h>
#include "VBF.h"

#define cout STD_COUT

float Objective(GAGenome &);    // This is the declaration of our obj function.

int
main(int argc, char **argv)
```

(continues on next page)

(continued from previous page)

```
{
  int n = 0, m = 0;
  int popsize;
  int ngen;
  float pmut;
  float pcross;

  n = atoi(argv[1]);
  m = atoi(argv[2]);
  int width    = 1 << n;
  int height   = m;
  popsize  = atoi(argv[3]);
  ngen     = atoi(argv[4]);
  pmut   = atof(argv[5]);
  pcross = atof(argv[6]);

  GA2DBinaryStringGenome genome(width, height, Objective);

  ifstream inStream("seed.bin");
  if(!inStream){
    cerr << "Cannot open " << "seed.bin" << " for input.\n";
    exit(1);
  }
  inStream >> genome;
  inStream.close();

  GASimpleGA ga(genome);
  ga.populationSize(popsize);
  ga.nGenerations(ngen);
  ga.pMutation(pmut);
  ga.pCrossover(pcross);
  ga.evolve();

// Now we print out the best genome that the GA found.

  cout << ga.statistics() << "\n";
  cout << ga.statistics().bestIndividual() << "\n";

  return 0;
}

float
Objective(GAGenome& g) {

  using namespace VBFNS;

  VBF F;
  NTL::mat_GF2 T;
  NTL::RR nonlin, lind, suma, nlm, ldm;
  long spacen, m;
  int d, n, ai, maxai;

  GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
  float score=0.0;

  spacen = genome.width();
  m = genome.height();
```

(continues on next page)

(continued from previous page)

```
  T.SetDims(spacen,m);
  for(int i=0; i<spacen; i++){
    for(int j=0; j<m; j++){
      T[i][j] = to_GF2(genome.gene(i,j));
    }
  }

  F.puttt(T);
  nonlin = nl(F);
  lind = ld(F);
  nlm = nlmax(F);
  ldm = ldmax(F);
  d = deg(F);
  n = logtwo(spacen);
  ai = AI(F);
  maxai = aimax(F);

  suma = nonlin/nlm+lind/ldm+to_RR(d)/to_RR(n)+to_RR(ai)/to_RR(maxai);
  conv(score,suma);

  return score;
}
```

## 11.3 9-variable Boolean functions with nonlinearity 242

Below you can find the Truth Tables of a set of 9-variable functions with the the best known nonlinearity (242). These Boolean functions can be grouped in five different affine equivalence classes:

A representative Boolean function of class f1

A representative Boolean function of class f2

A representative Boolean function of class f3

A representative Boolean function of class f4

A representative Boolean function of class f5

whose Frequency distribution of the absolute values of the Walsh Spectrum are the following:

| f | Values |
|---|---|
| f1 | (4,30),(12,46),(20,226),(28,210) |
| f2 | (4,30),(12,46),(20,226),(28,210) |
| f3 | (4,30),(12,46),(20,226),(28,210) |
| f4 | (4,56),(12,58),(20,154),(28,244) |
| f5 | (4,57),(12,91),(20,97),(28,267) |

whose Frequency distribution of the absolute values of the Autocorrelation Spectrum are the following:

| f | Values |
|---|---|
| f1 | (0,129),(8,298),(16,60),(24,9),(32,2),(40,13),(512,1) |
| f2 | (0,150),(8,196),(16,148),(24,12),(32,5),(512,1) |
| f3 | (0,183),(8,223),(16,84),(24,6),(32,4),(40,10),(56,1),(512,1) |
| f4 | (0,157),(8,232),(16,84),(24,8),(32,17),(40,10),(48,3),(512,1) |
| f5 | (0,192),(8,156),(16,129),(24,9),(32,13),(40,3),(48,6),(64,3),(512,1) |

The following table shows the values some criteria take for the functions within each class:

| Class | *LD* | *DEG* | *AI* | *MAXAC* | $\sigma$ |
|---|---|---|---|---|---|
| f1 | 118 | 7 | 4 | 40 | 324608 |
| f2 | 120 | 7 | 4 | 32 | 324608 |
| f3 | 114 | 7 | 4 | 56 | 324608 |
| f4 | 116 | 7 | 4 | 48 | 343424 |
| f5 | 112 | 7 | 4 | 64 | 354560 |

## 11.4 9-variable Balanced Boolean functions with nonlinearity 240

Below you can find the profiles of 9-variable balanced functions with the the best known nonlinearity (240). Each row represents a profile with four values separated by commmas: Algebraic Degree, Algebraic Immunity, Absolute Indicator and Sum-of-square Indicator:

### Profiles

567 different profiles were found with nonlinearity 240 and algebraic degree 8. The algebraic immunity takes values from the set {4, 5}.The linearity distance takes values from the set {110, 112, 114, 116, 118, 120, 122 }. The absolute indicator takes values from the set { 24, 32, 40, 48, 56, 64, 72 }. The sum-of-square indicator takes 137 different values between 323456 and 377600.

## 11.5 11-variable Balanced Boolean functions with nonlinearity 992

Below you can find the profiles of 11-variable balanced functions with the the best known nonlinearity (992). Each row represents a profile with four values separated by commmas: Algebraic Degree, Algebraic Immunity, Absolute Indicator and Sum-of-square Indicator:

### Profiles

3316 different profiles were found with nonlinearity 992. The algebraic degree takes values from the set { 9, 10 }. The algebraic immunity takes values from the set { 4, 5 }. The absolute indicator takes values from the set: { 120, 128, 136, 144, 160, 168, 176, 192, 200, 208, 224, 232, 240, 248, 256, 264 }. The sum-of-square indicator takes 682 different values between 5244800 and 5844608.

# 12 FAQ: Frequently Asked Questions

This is a list of Frequently Asked Questions about VBF Library. Feel free to suggest new entries!

## 12.1 General

### 12.1.1 Why does this library exist?

The stated mission of VBF is to be viable free open source library for analyzing Vector Boolean Functions used in ciphers. VBF uses best-of-breed free open source Number Theory Library called NTL by Victor Shoup.

I developed this library to support my PhD Thesis.

### 12.1.2 Who is behind this project?

At this moment, Jose Antonio Alvarez Cubero is the only person who is contributing to develop this code. If you are interested in contributing to it, please send a comment to the email: vbflibrary@gmail.com

### 12.1.3 Why is VBF free/open source?

I agree with Sage Authors: "A standard rule in the mathematics community is that everything is laid open for inspection. The Sage project believes that not doing the same for mathematics software is at best a gesture of impoliteness and rudeness, and at worst a violation against standard scientific practices. An underlying philosophical principle of Sage is to apply the system of open exchange and peer review that characterizes scientific communication to the development of mathematics software. Neither the Sage project nor the Sage Development Team make any claims to being the original proponents of this principle"

## 12.2 Using VBF

### 12.2.1 How do I get started?

To download VBF source code, go to VBF source code URL . You can read the documentation on http://vbf.rtfd.io.

### 12.2.2 What are VBF prerequisites?

The only prerequisite is to have a version of NTL library installed on the computer in which the VBF library is executed.

# 13 Bibliography

# 14 Indices and tables

- genindex
- modindex
- search

# References


[KASUMI:05] 3rd Generation Partnership Project. Specification of the 3GPP Confidentiality and Integrity Algorithms - Document 2: KASUMI specification (Release 6) no. 3GPP TS 35.202 V6.1.0 (2005-09). Technical report, 3GPP, 2005.

[BihamS:90] Biham, E. and Shamir, A. (1990). Differential cryptanalysis of DES-like cryptosystems. In CRYPTO, pages 2-21.

[Carlet:04] Carlet, C. (2004). On the secondary constructions of resilient and bent functions. In Progress in Computer Science and Applied Logic, pages 3-28.

[carlet2008higher] Carlet, C. (2008b). On the higher order nonlinearities of Boolean functions and S-boxes, and their generalizations. In Sequences and Their Applications SETA 2008, pages 345-367. Springer.

[CarletBF:08] Carlet, C. (2008a). Boolean functions for cryptography and error correcting codes.

[CAST:256] Adams, C. M. and Tavares, S. E. (1993). Designing s-boxes for ciphers resistant to differential cryptanalysis (extended abstract). In Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, pages 181-190.

[ChabaudV:94] F. Chabaud and S. Vaudenay. Links between differential and linear cryptanalysis. In Advances in Cryptology- EUROCRYPT 94, pages 356-365, 1995.

[Chaum:E85] Chaum, D. and Evertse, J.-H. (1985). Crytanalysis of des with a reduced number of rounds: Sequences of linear factors in block ciphers. In CRYPTO, pages 192-211.

[Chen:02] Chen, L., Fu, F.-W., and Wei, V. K. (2002). On the constructions and nonlinearity of binary vector correlation-immune functions. In Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on Information Theory, page 39.

[Courtois:03] N. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In Advances in cryptology CRYPTO 2003, Lecture Notes in Computer Science 2729, pages 177-194, 2003.

[courtois2002cryptanalysis] Courtois, N. and Meier, W. (2002). Algebraic attacks on stream ciphers with linear feedback. In Advances in cryptology EUROCRYPT 2003, Lecture Notes in Computer Science 2656, pages 346-359.

[CourtoisM:02] N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. In Advances in cryptology EUROCRYPT 03, Lecture Notes in Computer Science 2656, pages 346-359, 2002.

[DaemenR:02] Joan Daemen and Vincent Rijmen. The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[DES:77] DES. Data Encryption Standard. In FIPS PUB 46, Federal Information Processing Standards Publication, pages 46-2, 1977.

[DingXS:91] C. Ding, G. X. and Shan, W. (1991). The stability theory of stream ciphers. Berlin. Springer-Verlag. Lecture Notes in Computer Science Volume 561.

[Evertse:87] Evertse, J.-H. (1987). Linear structures in blockciphers. In EUROCRYPT, pages 249-266.

[Evertse:88] Evertse, J. H. (1988). Linear structures in block ciphers. In Advances in Cryptology - EUROCRYPT 87, no. 304 in Lecture Notes in Computer Science, pages 249-266.

[FaugereA:03] J.-C. Faugere and G. Ars. An algebraic cryptanalysis of nonlinear filter generators using Grobner bases. Technical report, INRIA 4739, 2003.

[Goldberg:1989] Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[GuptaS:05] Gupta, K. and Sarkar, P. (2005). Improved construction of nonlinear resilient S-boxes. Information Theory, IEEE Transactions on, 51(1):339-348.

[heys-tutorial] Heys, H. (1999). A tutorial on linear and differential cryptanalysis. Technical report, Electrical and Computer Engineering, Faculty of Engineering and Applied Science, Memorial University of Newfoundland, St. Johns, NF, Canada A1B 3X5.

[Hou:97] Hou, X. (1997). On the norm and covering radius of the first order reed-muller codes. In IEEE Transactions on Information Theory, Volume IT-43(3), pages 1025-1027.

[JakobsenK:97] Jakobsen, T. and Knudsen, L. R. (1997). The interpolation attack on block ciphers. In SAC 97, pages 28-40.

[Lai:94] Lai, X. (1994). Higher order derivatives and differential cryptanalysis. In Proceedings of the Symposium on Communication, Coding and Cryptography.

[Lai:95] Lai, X. (1995). Additive and linear structures of cryptographic functions. In Preneel, B., editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pages 75-85. Springer Berlin Heidelberg.

[linCaFEAL] Matsui, M. and Yamagishi, A. (1993). A New Method for Known Plaintext Attack of FEAL Cipher. In Rueppel, R. A., editor, Advances in Cryptology EUROCRYPT92, volume 658 of Lecture Notes in Computer Science, chapter 7, pages 81-91. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Matsui:93] Matsui, M. (1993). Linear cryptanalysis method for DES cipher. In EUROCRYPT, pages 386-397.

[Matsui:94] Matsui, M. (1994). The first experimental cryptanalysis of the Data Encryption Standard. In CRYPTO, pages 1-11.

[MeierS:89] Meier, W. and Staffelbach, O. (1989). Nonlinearity criteria for cryptographic functions. In EUROCRYPT, pages 549-562.

[Nyberg:91] Nyberg, K. (1991). Perfect nonlinear s-boxes. In EUROCRYPT, pages 378-386.

[Nyberg:92] Kaisa Nyberg. On the construction of highly nonlinear permutations. In Rainer A. Rueppel, editor, Advances in Cryptology EUROCRYPT 92, volume 658 of Lecture Notes in Computer Science, pages 92-98. Springer Berlin Heidelberg, 1993.

[Nyberg:93] Nyberg, K. (1993). Differentially uniform mappings for cryptography. In EUROCRYPT, pages 55-64.

[fse-Nyberg:94] Kaisa Nyberg. S-boxes and round functions with controllable linearity and differential uniformity. In Bart Preneel, editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pages 111-130. Springer Berlin / Heidelberg, 1995.

[Phan02miniadvanced] PHAN, R. C.-W. 2002. Mini advanced encryption standard (mini-AES): A testbed for cryptanalysis. Students, Cryptologia, 283-306.

[PreneelLLGV90] Preneel, B., Leekwijck, W. V., Linden, L. V., Govaerts, R., and Vandewalle, J. (1990). Propagation characteristics of boolean functions. In EUROCRYPT, pages 161-173.

[Preneel:93] Preneel, B. (1993). Analysis and design of cryptographic hash functions. Ph.D. dissertation, Katholieke Universiteit Leuven.

[PieprzykF:88] Pieprzyk, J. and Finkelstein, G. (1988). Towards effective nonlinear cryptosystem design. Computers and Digital Techniques, IEEE Proceedings, 135(6):325-335.

[Pommerening:05] Pommerening, K. (2005a). Linearitatsmase fur boolesche abbildungen. Technical report, Fachbereich Mathematik der Johannes-Gutenberg-Universitaet.

[Rothaus:76] O. S. Rothaus. On bent functions. J. Comb. Theory, Ser. A, 20(3):300-305, 1976.

[SarkarMaitra:00] Sarkar, P. and Maitra, S. (2000a). Construction of nonlinear boolean functions with important cryptographic properties. In EUROCRYPT, pages 488-511.

[Siegenthaler:84] Siegenthaler, T. (1984). Correlation-immunity of nonlinear combining functions for cryptographic applications. IEEE Transactions on Information Theory, 30(5):776-.

[Siegenthaler:85] Thomas Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. IEEE Transactions on Computers, 34(1):81-85, 1985.

[TardyG:91] Tardy-Corfdir, A. and Gilbert, H. (1992). A known plaintext attack of feal-4 and feal-6. In Feigenbaum, J., editor, Advances in Cryptology CRYPTO91, volume 576 of Lecture Notes in Computer Science, pages 172-182. Springer Berlin Heidelberg.

[c85-Webster-Tavares] Webster, A. F. and Tavares, S. E. (1986). On the design of S-boxes. In Williams, H. C., editor, Advances in Cryptology - Crypto85, pages 523-534, Berlin. Springer-Verlag. Lecture Notes in Computer Science Volume 218.

[XiaoM:88] Xiao, G.-Z. and Massey, J. L. (1988). A spectral characterization of correlation- immune combining functions. IEEE Transactions on Information Theory, 34(3):569-.

[zhang95gac] Xian-Mo Zhang and Yuliang Zheng. GAC: the criterion for global avalanche characteristics of cryptographic functions. Journal of Universal Computer Science, 1(5):320-337, 1995.