Vector Boolean Functions (VBF) Library*

User Manual
Edition 1.5
March 2016

written by
José Antonio Álvarez Cubero

revised by
Pedro J. Zufiria

*An electronic version of this User Manual, together with the analysis of some of the algorithms, can be found in [38].

Contents

Chapter 1

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.

1.1 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:

- Vector Boolean Function representations and characterizations
- Cryptographic criteria calculation
- 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.

1.2 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).

1.3 Software implementation

The package included consists of:

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

$$\begin{array}{l} pol.h, vbf\_GF2EX.h, vbf\_GF2X.h, vbf\_ZZ.h, vbf\_mat\_GF2.h, \\ vbf\_mat\_RR.h, vbf\_mat\_ZZ.h, vbf\_tools.h, vbf\_vec\_GF2.h, \\ vbf\_vec\_GF2E.h, vbf\_vec\_RR.h, vbf\_vec\_ZZ.h, vec\_pol.h \end{array} \tag{1.1}$$

2. Main class (VBF.h) with the functions,
3. A makefile to ease the compilation of example (Makefile),
4. A set of files associated with the decimal representation of KASUMI [1] S-boxes (S7.dec and S9.dec).

The Output files can be found within "KASUMI Analysis" in the "Examples" menu at the Web site http://vbflibrary.tk.

1.4 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.

1.5 Installation

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

1. **Download the last version of the library** from https://github.com/jacubero/VBF/tree/master/src 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 http://www.shoup.net/ntl/download.html, 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.

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.

1.6 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 χ_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:

$$\hat{\chi}_f(\mathbf{u}) = \langle \xi_f, \xi_{l_\mathbf{u}} \rangle = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{f(\mathbf{x}) + \mathbf{u}\mathbf{x}} \tag{1.2}$$

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 *:

$$r_f(\mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} \chi_f(\mathbf{x}) \chi_f(\mathbf{x} + \mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{f(\mathbf{x}) + f(\mathbf{u} + \mathbf{x})} \tag{1.3}$$

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

$$\Delta_\mathbf{u} f(\mathbf{x}) = f(\mathbf{x} + \mathbf{u}) + f(\mathbf{x}), \quad \mathbf{x} \in \mathrm{V_n} \; . \tag{1.4}$$

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.

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

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 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 [8] as:

$$\theta_F(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{y} = F(\mathbf{x}) \\ 0 & \text{if } \mathbf{y} \neq F(\mathbf{x}) \end{cases} \tag{1.5}$$

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:

$$\hat{\theta}_F(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{x}\in\mathrm{V_n}} \sum_{\mathbf{y}\in\mathrm{V_m}} \theta_F(\mathbf{x}, \mathbf{y})\chi_{(\mathbf{u},\mathbf{v})}(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{x}\in\mathrm{V_n}} (-1)^{\mathbf{u}\mathbf{x}+\mathbf{v}F(\mathbf{x})} \tag{1.6}$$

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 [31]:

$$r_F(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{x}\in\mathrm{V_n}} \chi_{\mathbf{v}F}(\mathbf{x} + \mathbf{u})\chi_{\mathbf{v}F}(\mathbf{x}) = \sum_{\mathbf{x}\in\mathrm{V_n}} (-1)^{\mathbf{v}F(\mathbf{x}+\mathbf{u})+\mathbf{v}F(\mathbf{x})} \tag{1.7}$$

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.

1.7 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 1.1. 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 1.2.

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

Table 1.1: NTL modules used in VBF

CLASS NAME	*DESCRIPTION*
GF2	Galois Field of order 2 denoted by GF(2)
vec_GF2	Vectors over GF(2)
mat_GF2	Matrices over 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

GF2X of degree less than the degree of P. GF2EX can be used, for example, for arithmetic in GF(2^n)[X].

Table 1.2: 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.

Chapter 2

Using the library

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

2.1 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);
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();

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();

  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 would be:

- S7.ac (Autocorrelation Spectrum)
- S7.anf (ANF Table)
- S7.cy (Cycle structure)
- S7.dp (Differential Profile)
- S7.lp (Linear Profile)
- S7.ls (Linear structures): It is an empty vector because there is no linear structures
- S7.pol (Polynomial representation)
- S7.tt (Truth Table)
- S7.wal (Walsh Spectrum)

The same applies to S9 S-box analysis.

2.2 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
```

2.3 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).
 *
 *      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,
        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.
 *              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                      */
```

```
        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 )
{
        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;
        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();
   }

}
```

Chapter 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.

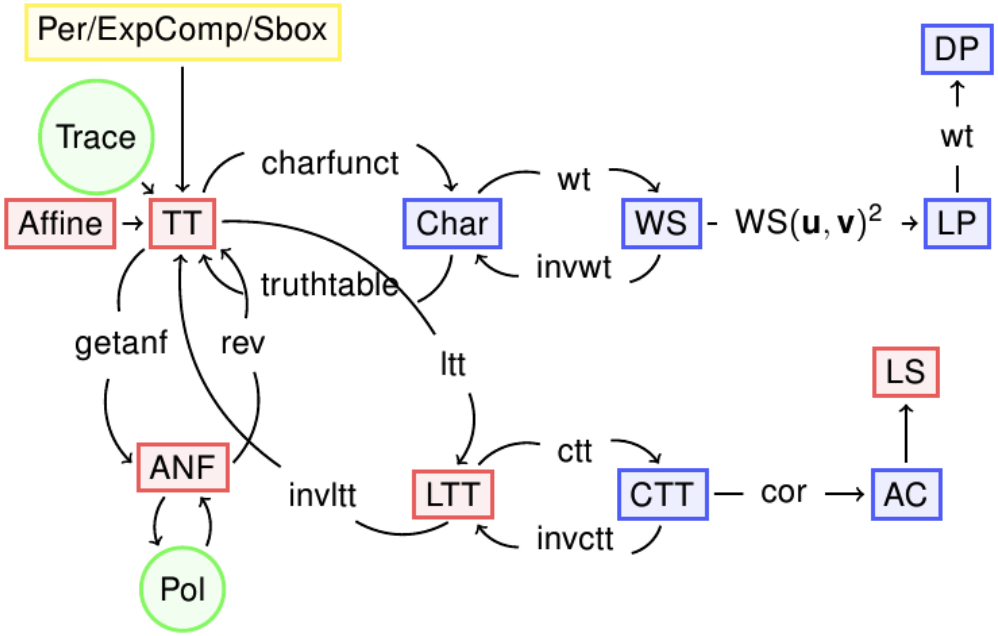
Representations included in this chapter are the Truth Table (TT), the polynomials in Algebraic Normal Form (Pol) and ANF Table (ANF), the Image (Char), Component functions Truth Table(LTT), Sequence vectors of Component functions CTT, the Trace Representation (Trace) and Affine function Representation. 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 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.

3.1 Truth Table

3.1.1 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.

Definition 3.1.1. 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:

$$F(\boldsymbol{\alpha_i}) \in \mathrm{V_m} \quad \forall i \in \{0, \ldots, 2^n - 1\} \tag{3.1}$$

The matrix with 2^n rows and m columns will be referred as the *Truth Table* of F and will be generally written as TT_F:

$$\mathsf{TT}_F = \begin{bmatrix} f_1(\boldsymbol{\alpha_0}) & \dots & f_m(\boldsymbol{\alpha_0}) \\ f_1(\boldsymbol{\alpha_1}) & \dots & f_m(\boldsymbol{\alpha_1}) \\ \dots & \dots & \dots \\ f_1(\boldsymbol{\alpha_{2^n-1}}) & \dots & f_m(\boldsymbol{\alpha_{2^n-1}}) \end{bmatrix} \tag{3.2}$$

each $\boldsymbol{\alpha_i} = (x_1, \dots, x_n) \in \mathrm{V_n} \quad i \in \{1, \dots, 2^n - 1\}$ 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 $\boldsymbol{\alpha_0} < \boldsymbol{\alpha_1} < \dots < \boldsymbol{\alpha_{2^n-1}}$.

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 $\mathsf{TT}_{f_i} \quad i \in \{1, \dots, m\}$) and it holds that:

$$\begin{array}{rcl} \gamma : \mathcal{F}_{n,m} & \rightarrow & \mathsf{M}_{2^n \times m}(\mathrm{GF}(2)) \\ F & \rightarrow & \mathsf{TT}_F \end{array} \tag{3.3}$$

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., $\mathsf{TT}_f = (f(0), f(1), f(2), \dots, f(2^n - 1))$. 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:

$$d(F, G) = \sum_{\mathbf{x} \in \mathrm{V_n}} d\left(F(\mathbf{x}), G(\mathbf{x})\right) \tag{3.4}$$

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: LTT and CTT.

We will denote by 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 CTT of F the matrix whose columns are the sequence vectors of the 2^m component functions of F *.

3.1.2 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)
```

*Sometimes it is called the Polarity Truth Table.

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 3.1.1. 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;
```

```
   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]
[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]
]
```

3.2 Trace Representation

3.2.1 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 follows:

$$tr(\mathbf{x}) = \sum_{i=0}^{2^n-1} x^i \tag{3.5}$$

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:

Definition 3.2.1. 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 $2^n - 1$:

$$F(\mathbf{x}) = \sum_{i=0}^{2^n-1} \delta_i x^i,\ \delta_i \in \mathrm{GF}(2^{\mathrm{n}}) \tag{3.6}$$

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* [21] is efficient when the degree of the univariate polynomial representation of the S-box over $\mathrm{GF}(2^n)$ 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.

3.2.2 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 3.2.1. The following program provides the Trace representation over $\mathrm{GF}(2^n)$of a Vector Boolean function with Truth Table in a file with extension ".tt". $\mathrm{GF}(2^n)$ 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, 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 the Rijndael S-box S_{RD} (described in Figure 2.6), provided in a file with extension ".tt", and the corresponding GF2X representation of g : [110110001], provided in a file with extension ".irr". The output of the program would be a GF2EX which represents polynomials over GF2E; hence, it can be used, for example, for arithmetic in GF(2^n):

$$05 \cdot x^{254} + 09 \cdot x^{253} + f9 \cdot x^{251} + 25 \cdot x^{247} + f4 \cdot x^{239} + 01 \cdot x^{223} + b5 \cdot x^{191} + 8f \cdot x^{127} + 63 \quad (3.7)$$

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

3.3 Polynomials in ANF

3.3.1 Description

Definition 3.3.1. 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 kth-order product terms ($0 < k \leq n$) of the variables in the form:

$$\begin{aligned} f(x_1, \ldots, x_n) \quad &= a_0 + a_1 x_1 + \cdots + a_n x_n + a_{12} x_1 x_2 + \cdots + a_{n-1,n} x_{n-1} x_n + \cdots \\ &+ a_{12\ldots n} x_1 x_2 \ldots x_n = \textstyle\sum_{\mathsf{I} \in \mathsf{P}(N)} a_{\mathsf{I}} \left(\prod_{i \in \mathsf{I}} x_i \right) = \sum_{\mathsf{I} \in \mathsf{P}(N)} a_{\mathsf{I}} x^{\mathsf{I}}, \quad a_{\mathsf{I}} \in \mathrm{GF}(2) \end{aligned} \tag{3.8}$$

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).

3.3.2 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 3.3.1. 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:

$$\begin{aligned}
f_1(NibbleSub) &= 1 + x_4 + x_2 + x_2x_3 + x_2x_3x_4 + x_1 + x_1x_2 + x_1x_2x_3 \\
f_2(NibbleSub) &= 1 + x_3x_4 + x_2 + x_2x_4 + x_1 + x_1x_3 + x_1x_3x_4 \\
f_3(NibbleSub) &= 1 + x_4 + x_3 + x_3x_4 + x_2x_4 + x_2x_3 + x_1x_4 + x_1x_3 + x_1x_2 + x_1x_2x_4 + x_1x_2x_3 \\
f_4(NibbleSub) &= x_3 + x_2x_4 + x_1 + x_1x_4 + x_1x_3x_4
\end{aligned} \tag{3.9}$$

3.4 ANF Table

3.4.1 Description

Definition 3.4.1. *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:

$$\mathsf{ANF}_F{}^i = \mathsf{ANF}_{f_i} \quad i \in \{1, \ldots, m\} \tag{3.10}$$

where $\mathsf{ANF}_F{}^i$ is the i-th column of 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.

3.4.2 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 3.4.1. 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) {
      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]
]
```

3.5 Image

3.5.1 Description

Definition 3.5.1. 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:

$$\theta_F(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{y} = F(\mathbf{x}) \\ 0 & \text{if } \mathbf{y} \neq F(\mathbf{x}) \end{cases} \tag{3.11}$$

Definition 3.5.2. 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:

$$\mathsf{Img(F)} = \begin{bmatrix} \theta_F(\boldsymbol{\alpha_0},\boldsymbol{\alpha_0}) & \dots & \theta_F(\boldsymbol{\alpha_0},\boldsymbol{\alpha_{2^m-1}}) \\ \theta_F(\boldsymbol{\alpha_1},\boldsymbol{\alpha_0}) & \dots & \theta_F(\boldsymbol{\alpha_1},\boldsymbol{\alpha_{2^m-1}}) \\ \dots & \dots & \dots \\ \theta_F(\boldsymbol{\alpha_{2^n-1}},\boldsymbol{\alpha_0}) & \dots & \theta_F(\boldsymbol{\alpha_{2^n-1}},\boldsymbol{\alpha_{2^m-1}}) \end{bmatrix} \tag{3.12}$$

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

Lemma 3.5.1. *By equation 1.5, it is clear that all the rows of the matrix* Img(F) *have one element equal to one and the rest is zero, that is* $\forall\, i \in \{1, \ldots, 2^n\}$*:*

$$\begin{array}{rcl} \mathsf{Img(F)}_i & = & \begin{bmatrix} a_{i1} & \ldots & a_{i2^m} \end{bmatrix} \\ \textit{where } (\exists! j \in \{1, \ldots, 2^m\}\ a_{ij} = 1) & \wedge & (a_{ik} = 0\ \forall\, k \neq j \in \{1, \ldots, 2^m\}) \end{array} \tag{3.13}$$

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 $truthtable$.

3.5.2 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 3.5.1. 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 0 1 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 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 3.1 in which the rows and columns are indexed with the corresponding vector:

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

3.5.3 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.

Theorem 3.5.2. *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:*

$$\mathbf{x}\mathsf{H_n} = \mathbf{y}, \forall\, \mathbf{x}, \mathbf{y} \in \mathbb{R}^{2^n} \tag{3.14}$$

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

Figure 3.1: *Image representations of NibbleSub.*

Corollary. *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 $\mathsf{H_n}$.*

$$\xi_f = a_{\boldsymbol{\alpha_0}} \xi_{l_{\boldsymbol{\alpha_0}}} + \cdots + a_{\boldsymbol{\alpha_{2^n-1}}} \xi_{l_{\boldsymbol{\alpha_{2^n-1}}}} \tag{3.15}$$

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

Definition 3.5.3. 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:

$$\mathcal{W}_f(\mathbf{u}) = \hat{\chi}_f(\mathbf{u}) = \mathcal{W}\{\xi_f\}(\mathbf{u}) = \langle \xi_f, \xi_{l_{\mathbf{u}}} \rangle = \sum_{\mathbf{x} \in \mathrm{V_n}} \chi_f(\mathbf{x}) \chi_{\mathbf{u}}(\mathbf{x}) \tag{3.16}$$

or, as it is most often written as:

$$\mathcal{W}_f(\mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} (-1)^{f(\mathbf{x}) + \mathbf{u} \cdot \mathbf{x}} \tag{3.17}$$

As a result, the Walsh Transform of $f \in \mathcal{F}_n$ at $\mathbf{u}$ is the coefficient of the sequence of f (ξ_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 ξ_f and $\mathcal{W}_f$ form a Transform pair and write:

$$\xi_f \overset{W}{\longleftrightarrow} \mathcal{W}_f \quad (\xi_f \text{ corresponds to } \mathcal{W}_f) \tag{3.18}$$

Definition 3.5.4. 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:

$$\mathsf{WS}(f) = \left[\hat{\chi}_f(\boldsymbol{\alpha_0}) \quad \dots \quad \hat{\chi}_f(\mathbf{u}) \quad \dots \quad \hat{\chi}_f(\boldsymbol{\alpha_{2^n-1}})\right]^T \tag{3.19}$$

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.

Definition 3.5.5. 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:

$$\mathsf{WS}(F) = \begin{bmatrix} \hat{\theta}_F(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_0}) & \dots & \hat{\theta}_F(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_{2^m-1}}) \\ \hat{\theta}_F(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_0}) & \dots & \hat{\theta}_F(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_{2^m-1}}) \\ \dots & \dots & \dots \\ \hat{\theta}_F(\boldsymbol{\alpha_{2^n-1}}, \boldsymbol{\alpha_0}) & \dots & \hat{\theta}_F(\boldsymbol{\alpha_{2^n-1}}, \boldsymbol{\alpha_{2^m-1}}) \end{bmatrix} \tag{3.20}$$

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 $l_{\mathbf{v}} \in \mathcal{L}_m$.

3.5.4 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 3.5.2. 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]
]
```

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

$$NibbleSub = (f_1(NibbleSub), f_2(NibbleSub), f_3(NibbleSub), f_4(NibbleSub)) \tag{3.21}$$

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 $(1,0,0,0)$.

3.6 Linear Profile and Linear Cryptanalysis

3.6.1 Description

A complete enumeration of all linear approximations of the S-box is given in the *Linear Profile* [†], 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 [8] $2^{-n} \leq lp(F) \leq 1$ 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.

3.6.2 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 literature, an equivalent matrix called Linear Approximation Table is used as well.

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 3.6.1. 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]
[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 3.2 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	1111
0000	256	0	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	0
0010	0	0	16	16	0	0	16	16	0	0	16	16	0	0	144	16
0011	0	0	0	0	0	0	0	0	16	144	16	16	16	16	16	16
0100	0	16	0	16	16	64	16	0	0	16	0	16	16	64	16	0
0101	0	16	16	0	16	0	64	16	16	0	64	16	0	16	16	0
0110	0	16	16	64	16	0	0	16	0	16	16	64	16	0	0	16
0111	0	16	0	16	16	64	16	0	16	0	16	0	64	16	0	16
1000	0	0	0	0	0	0	0	0	16	16	16	16	16	16	16	144
1001	0	0	16	16	0	0	16	16	64	0	16	16	0	64	16	16
1010	0	64	16	16	64	0	16	16	16	16	0	0	16	16	0	0
1011	0	64	0	64	64	0	64	0	0	0	0	0	0	0	0	0
1100	0	16	64	16	16	0	16	0	16	0	16	64	0	16	0	16
1101	0	16	16	0	16	64	0	16	64	16	16	0	16	0	0	16
1110	0	16	16	0	16	64	0	16	16	0	0	16	64	16	16	0
1111	0	16	64	16	16	0	16	0	0	16	64	16	16	0	16	0

Figure 3.2: *Linear Profile of NibbleSub.*

3.7 Differential Profile and Differential Cryptanalysis

3.7.1 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* [‡], 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.

Definition 3.7.1. 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:

$$D_F(\mathbf{u}, \mathbf{v}) = \{\mathbf{x} \in \mathrm{V_n} \mid \Delta_{\mathbf{u}} F(\mathbf{x}) = \mathbf{v}\} \tag{3.22}$$

Definition 3.7.2. 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 [29] introduced the concept of *differential uniformity* as a measure of the resistance to differential crytanalysis as follows:

Definition 3.7.3. 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 $\mathbf{v} \in \mathrm{V_m}$:

$$\#\{\mathbf{x} \in \mathrm{V_n} \mid F(\mathbf{x} + \mathbf{u}) + F(\mathbf{x}) = \mathbf{v}\} \leq du(F) \tag{3.23}$$

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,

$$du(F) = \max_{(\mathbf{u},\mathbf{v}) \neq (\mathbf{0},\mathbf{0})} \#D_F(\mathbf{u}, \mathbf{v}) = \max_{(\mathbf{u},\mathbf{v}) \neq (\mathbf{0},\mathbf{0})} \#\{\mathbf{x} \in \mathrm{V_n} \mid F(\mathbf{x}) + F(\mathbf{x} + \mathbf{u}) = \mathbf{v}\} \tag{3.24}$$

Definition 3.7.4. Let define the function $\delta_F : \mathrm{V_n} \times \mathrm{V_m} \to \mathbb{Q}$ as follows:

$$\delta_F(\mathbf{u}, \mathbf{v}) = \frac{1}{2^n} \#D_F(\mathbf{u}, \mathbf{v}) \tag{3.25}$$

Definition 3.7.5. 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:

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

[‡]In the literature, an equivalent matrix called Difference Distribution Table is used as well.

Definition 3.7.6. The maximum value of $\delta_F(\mathbf{u}, \mathbf{v})$ is called the *differential potential* of F:

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

Corollary. *The differential uniformity of $F \in \mathcal{F}_{n,m}$ and its differential potential are related as follows:*

$$dp(F) = \frac{1}{2^n} du(F) \tag{3.26}$$

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 $dp(F) = 2^{-n} du(F)$.

3.7.2 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 3.7.1. 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;

   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 3.3 represents the Differential Profile of $NibbleSub$ and emphasizes in blue the elements which achieve the highest value.

3.8 Autocorrelation Spectrum

3.8.1 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.

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

Figure 3.3: *Differential Profile of NibbleSub.*

Definition 3.8.1. The *directional derivative* of $f \in \mathcal{F}_n$ in the direction of $\mathbf{u} \in \mathrm{V_n}$ is defined as:

$$\Delta_{\mathbf{u}} f(\mathbf{x}) = f(\mathbf{x}+\mathbf{u}) + f(\mathbf{x}), \quad \mathbf{x} \in \mathrm{V_n} \tag{3.27}$$

Similarly, the *directional derivative* of the sequence of a Boolean function ξ_f in the direction of $\mathbf{u} \in \mathrm{V_n}$ is defined as:

$$\Delta_{\mathbf{u}} \chi_f(\mathbf{x}) = \chi_f(\mathbf{x}+\mathbf{u}) \cdot \chi_f(\mathbf{x}), \quad \mathbf{x} \in \mathrm{V_n} \tag{3.28}$$

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:

$$r_f(\mathbf{u}) = \sum_{\mathbf{x} \in \mathrm{V_n}} \chi_f(\mathbf{x}) \chi_f(\mathbf{x}+\mathbf{u}) \tag{3.29}$$

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

1. For every Boolean function $r_f(\mathbf{0}) = 2^n$, since $(\chi_f(\mathbf{x}))^2 = 1 \; \forall \, \mathbf{x} \in \mathrm{V_n}$.
2. 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.

Definition 3.8.2. *Autocorrelation Spectrum*, denoted by $\mathrm{R}(F) \in \mathrm{M}_{2^n \times 2^m}(\mathbb{Z})$, obtained by Equation 1.7. 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.

3.8.2 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 [23], [9]. 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:

Definition 3.8.3. 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 [9] and Evertse [17]. 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.

Definition 3.8.4. $F \in \mathcal{F}_{n,m}$ 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 $c \in \mathrm{GF}(2) \ \ \forall \mathbf{x} \in \mathrm{V_n}$.

Definition 3.8.5. $F \in \mathcal{F}_{n,m}$ 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 [17]. 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.

3.8.3 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 3.8.1. 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;
   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 3.4 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	1111
0000	16	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	8
0010	16	−8	0	−8	−8	0	0	8	8	−8	0	0	−8	8	−8	8
0011	16	0	0	0	0	0	0	−16	−8	8	0	0	0	0	−8	8
0100	16	0	−8	0	0	−16	0	8	0	8	−8	−8	−8	0	8	8
0101	16	0	0	−8	0	0	0	−8	0	−8	8	−8	0	−8	8	8
0110	16	−8	0	0	−8	0	−8	8	0	−8	0	0	8	0	−8	8
0111	16	0	−8	0	0	0	0	−8	0	8	0	0	0	−8	−8	8
1000	16	−8	−8	0	−8	0	0	8	−8	8	0	0	0	0	8	−8
1001	16	0	0	8	0	0	0	−8	0	−8	0	0	−8	0	8	−8
1010	16	8	0	0	8	0	8	8	−8	−8	0	−8	0	0	−8	−16
1011	16	0	−8	−8	0	16	−8	−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	−8
1101	16	0	0	0	0	0	8	−8	0	−16	0	0	0	0	8	−8
1110	16	8	0	8	8	0	0	8	0	−8	−8	0	0	−8	−16	−8
1111	16	0	8	0	0	−16	0	−8	0	8	8	8	−8	0	−8	−8

Figure 3.4: *Linear structures of NibbleSub.*

3.9 Affine Function and Affine Equivalence

3.9.1 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.

Definition 3.9.1. A Boolean linear function is defined as the sum of a subset of the input variables, denoted

$$l_{\mathbf{u}}(\mathbf{x}) = u_1x_1 + u_2x_2 + \cdots + u_nx_n \tag{3.30}$$

where $\mathbf{u} = (u_1, \ldots, u_n) \in \mathrm{V_n}$.

Definition 3.9.2. The set of Boolean affine functions are the linear functions and their complements, denoted

$$l_{\mathbf{u},b}(\mathbf{x}) = l_{\mathbf{u}}(\mathbf{x}) + b \tag{3.31}$$

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.

Definition 3.9.3. 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 $\mathsf{A} \in \mathsf{M}_{n\times n}(\mathrm{GF}(2))$, $\mathbf{b}, \mathbf{c} \in \mathrm{V_n}$ and $d \in \mathrm{GF}(2)$ and it is an affine transformation. The functions f and g satisfying the previous relation 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 the Absolute Values of the 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 [34]:

$$\hat{\chi}_g(\mathbf{u}) = (-1)^{\mathbf{c}\cdot\mathsf{A}^{-1}\mathbf{b}}(-1)^{\mathbf{u}\cdot\mathsf{A}^{-1}\mathbf{b}}\hat{\chi}_f\left(\left(\mathsf{A}^{-1}\right)\mathbf{u} + \left(\mathsf{A}^{-1}\right)\mathbf{c}\right) \tag{3.32}$$

Thus nonlinearity is also invariant under affine transformation.

Frequency Distribution of the Absolute Values of the 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 [34]:

$$r_g(\mathbf{u}) = (-1)^{\mathbf{u}\cdot\mathbf{c}} r_f(\mathsf{A}\mathbf{u}) \tag{3.33}$$

Thus absolute indicator is also invariant under affine transformation.

3.9.2 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 3.9.1. 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 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]
[4 -4 -4 4]
]
Frequency distribution of the absolute values of the Autocorrelation Spectrum:
(4,4)
(4,4)
(4,4)
```

3.10 Cycle Structure, Fixed Points and Negated Fixed Points

3.10.1 Description

Definition 3.10.1. 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:

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 6 & 4 & 7 & 5 & 9 & 1 & 8 & 2 \end{bmatrix} \tag{3.34}$$

can be written

$$F = (1347)(269)(5)(8) \tag{3.35}$$

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:

- 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.
- 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.
- 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 invert 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.

3.10.2 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 3.10.1. 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:
```

```
[]

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

which means:

Table 3.1: 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 $NibbleSub[(1,0,0,0)] = (0,1,1,1)$.

3.11 Permutation Vector

3.11.1 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.

3.11.2 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 3.11.1. 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"

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]
```

3.12 DES Representations

3.12.1 Description

The VBF library accepts to types of representations of DES [16] 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 lists 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.

3.12.2 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)
```

S_1																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0yyyy1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
1yyyy0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
1yyyy1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
0yyyy1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
1yyyy0	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
1yyyy1	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
0yyyy1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
1yyyy0	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1yyyy1	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
0yyyy1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
1yyyy0	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
1yyyy1	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Figure 3.5: $S1, S2, S3, S4$ *DES S-boxes.*

S_5																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
0yyyy1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
1yyyy0	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
1yyyy1	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
0yyyy1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
1yyyy0	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
1yyyy1	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
0yyyy1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1yyyy0	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
1yyyy1	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8																
	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
0yyyy1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
1yyyy0	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
1yyyy1	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 3.6: $S5, S6, S7, S8$ *DES S-boxes.*

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 3.12.1. 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 ]
[ 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 [§]:

```
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]
[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]
```

[§]Only a few values of S1 Truth Table is printed for space reasons.

```
[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]
...
```

3.13 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:

- 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)

- 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)

- 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)

- 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).

- A matrix representing the linear combinations of Truth Table coordinate functions:

  void LTT(NTL::mat_GF2& X, VBF& a).

- A matrix representing character form of Truth Table coordinate functions:

  void CTT(NTL::mat_GF2& X, VBF& a).

3.14 Summary

Table 3.2 lists the member functions related to methods of vector Boolean functions initialization. Table 3.3 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)*.

Table 3.2: Representation of VBF.

SYNTAX	DESCRIPTION
void puttt(const NTL::mat_GF2& T)	$\mathsf{TT}_F = \mathsf{T}$
void putHexTT(istream & s)	VBF which has an hexadecimal representation of its Truth Table defined by s
void putBinTT(istream & s)	VBF which has a binary representation of its Truth Table defined by s
void putDecTT(const NTL::vec_long& d,const long& m)	VBF which has an 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 putaffine(const NTL::mat_GF2& A,const NTL::vec_GF2& b)	$F(\mathbf{x}) = \mathbf{x}\mathsf{A} + \mathbf{b}$
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

Table 3.3: Chacterizations of VBF.

SYNTAX	DESCRIPTION
void TT(NTL::mat_GF2& X, VBF& F)	$\mathsf{X} = \mathsf{TT}_F$
void getHexTT(ostream& s)	s is the hexadecimal representation of the Truth Table of F
void getBinTT(ostream& s)	s is the binary 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)	$\mathsf{X} = \mathsf{Img(F)}$
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)	$\mathbf{v}$ 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)	$\mathbf{v}$ is the permutation vector defined by F

Chapter 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.

The figure 4.1 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.

In this chapter we apply VBF library methods to find out cryptographic criteria of several cryptographic algorithms. Refer to `http://vbflibrary.tk` for an extensive description of cryptographic criteria of modern cryptographic algorithms apart from those described in this chapter.

4.1 Algebraic Degree

4.1.1 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

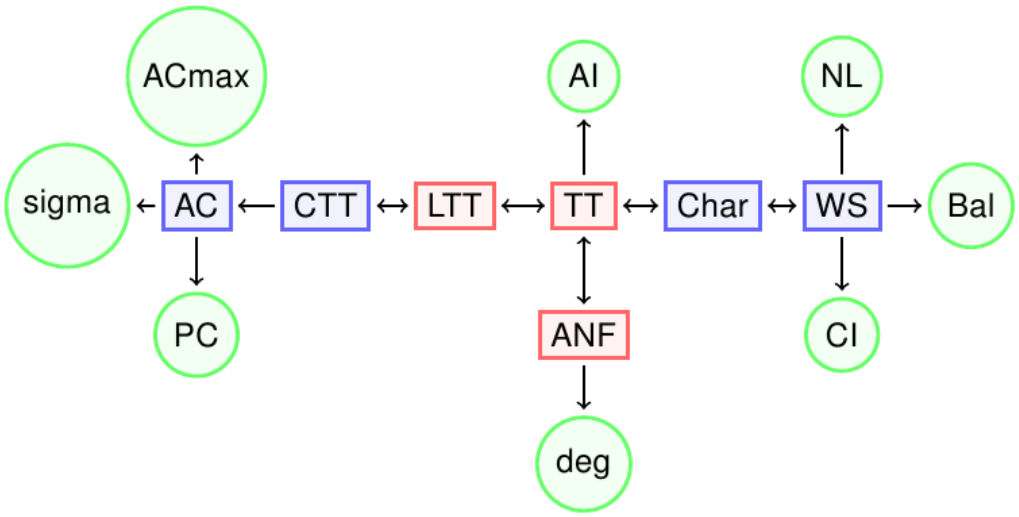


Figure 4.1: *Relationships among representations and criteria of a Vector Boolean function.*

indicator of the function's algebraic complexity. The higher the degree of a function, the greater is its algebraic complexity. *Higher order differential attack* [22] exploits the fact that the algebraic degree of the S-box is low.

Definition 4.1.1. *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 [30], namely:

$$deg(F) = \min_{g}\{deg(g) \mid g = \sum_{j=1}^{m} v_j f_j,\ \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m}\} \tag{4.1}$$

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.

4.1.2 Library

The method used to obtain this criterion is the following:

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

Example 4.1.1. The following program provides the algebraic degree of a Vector Boolean function given its Truth Table.

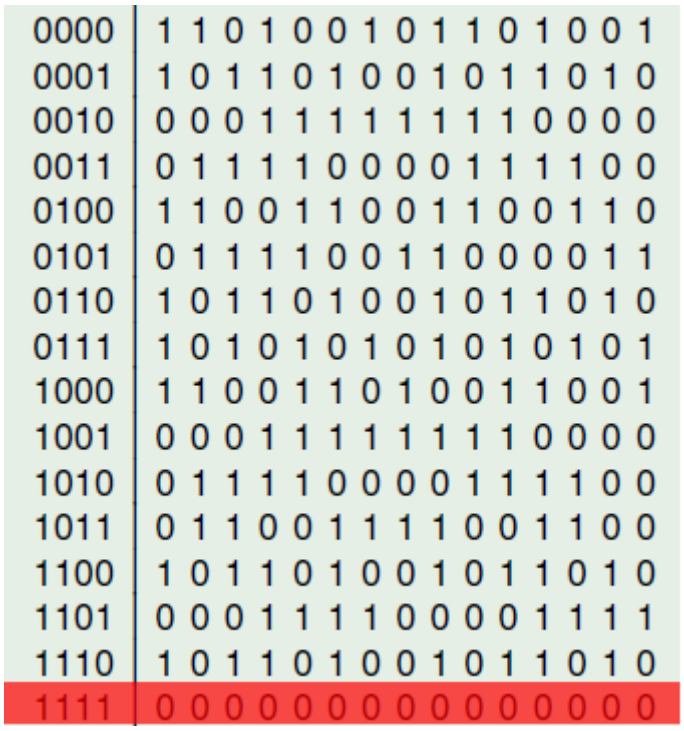
```
#include <iostream>
#include <fstream>
#include "VBF.h"

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 figure 4.2 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.

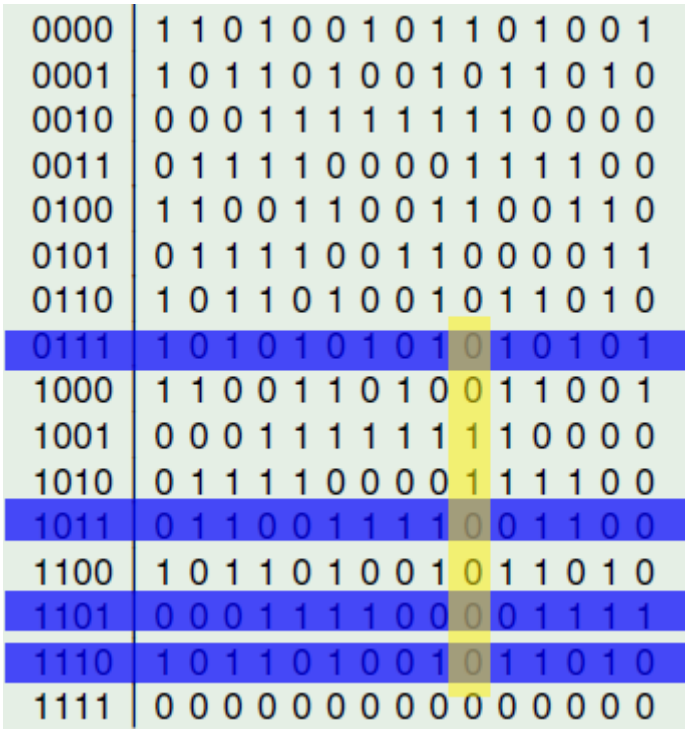
The figure 4.3 represents the ANF table of NibbleSub nonzero component functions and emphasizes in blue the ANF terms of degree 3. As we can see there are no terms of degree 3 in one of the component functions of NibbleSub, which is marked in yellow.

The figure 4.4 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

Figure 4.2: *Algebraic Degree of NibbleSub: Degree 4.*



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

Figure 4.3: *Algebraic Degree of NibbleSub: Degree 3.*

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

Figure 4.4: *Algebraic Degree of NibbleSub: Degree 2.*

4.2 Nonlinearity

4.2.1 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, [24] for block ciphers and [4] for stream ciphers).

Definition 4.2.1. 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 [32]:

$$NL(f) = d(f, \mathsf{A}_n) \tag{4.2}$$

Definition 4.2.2. 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 [30]:

$$NL(F) = \min_{\mathbf{v} \neq \mathbf{0}} NL(\mathbf{v} \cdot F) \quad \mathbf{v} = (v_1, \ldots, v_m) \in \mathrm{V_m} \tag{4.3}$$

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

Theorem 4.2.1. *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:*

$$NL(F) = 2^{n-1} - \frac{1}{2} \overset{*}{\max} \left(\mathsf{WS}(F)(\mathbf{u}, \mathbf{v})\right) \tag{4.4}$$

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

$$NL(f) = 2^{n-1} - \frac{1}{2} \max_{\mathbf{u} \in \mathrm{V_n} \neq \mathbf{0}} |\hat{\chi}_f(\mathbf{u})| \tag{4.5}$$

Definition 4.2.3. The *spectral radius* of a Boolean function $f \in \mathcal{F}_n$ is $r(f) = \max_{\mathbf{u} \in \mathrm{V_n} \neq \mathbf{0}} |\hat{\chi}_f(\mathbf{u})|$.

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 [25]. 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.

4.2.2 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 4.2.1. 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
   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 figure 4.5 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	−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

Figure 4.5: *Nonlinearity of NibbleSub.*

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

Example 4.2.2. 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>
#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.

4.3 r-th Order Nonlinearity

4.3.1 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 $r(r < n)$.

Definition 4.3.1. 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 [7] for details).

$$NL_r(F) = \min_{\mathbf{v}\neq\mathbf{0}\in \mathrm{V_m}} NL_r(\mathbf{v}\cdot F) = \min_{\mathbf{v}\neq\mathbf{0}\in \mathrm{V_m}} \min_{f\in\mathcal{F}_n} d(f, \mathbf{v}\cdot F) \tag{4.6}$$

Computing rth-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.

4.3.2 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 4.3.1. 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.

4.4 Balancedness

4.4.1 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).

Definition 4.4.1. $F \in \mathcal{F}_{n,m}$ 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:

$$\hat{\theta}_F(\mathbf{0}, \mathbf{v}) = 0, \ \forall \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m} \tag{4.7}$$

Theorem 4.4.1. *[33] $f \in \mathcal{F}_n$ is balanced if and only if the Walsh coefficient at* $\mathbf{0}$ *is zero:*

$$f \text{ is balanced} \iff \hat{\chi}_f(\mathbf{0}) = 0 \tag{4.8}$$

Theorem 4.4.2. *[33] $F \in \mathcal{F}_{n,m}$ 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:*

$$F \text{ is balanced} \iff \hat{\theta}_F(\mathbf{0}, \mathbf{v}) = 0, \ \forall \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m} \tag{4.9}$$

Definition 4.4.2. The imbalance of a Boolean function is defined to be

$$I(f) = |wt(f) - 2^{n-1}| = 2^{n-1}|\mathrm{C}(f, 0)| \tag{4.10}$$

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 (see section 3.6). In particular, a large imbalance may enable the function to be easily approximated by a constant function.

4.4.2 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 4.4.1. 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;
      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 figure 4.6 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	−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

Figure 4.6: *Balancedness of NibbleSub.*

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

4.5 Correlation Immunity

4.5.1 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 [37].

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 [37] 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.

Definition 4.5.1. 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.

Definition 4.5.2. [40] $f \in \mathcal{F}_n$ 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.

Definition 4.5.3. [10] $F \in \mathcal{F}_{n,m}$ 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.

Theorem 4.5.1. *[40] 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:*

$$f \text{ is a } t\text{-CI function} \Leftrightarrow \hat{\chi}_f(\mathbf{u}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq t \tag{4.11}$$

f can also be denoted as $(n,1,t)$-CI function.

Theorem 4.5.2. *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:*

$$F \text{ is a } t\text{-CI function} \ \Leftrightarrow\ \hat{\theta}_F(\mathbf{u},\mathbf{v}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq t,\ \forall\, \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m} \tag{4.12}$$

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.

4.5.2 Library

The method used to obtain this criterion is the following:

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

Example 4.5.1. 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;

   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 figure 4.7 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

Figure 4.7: *Correlation immunity of* $(1+x_1)(1+x_2)(1+x_3)(1+x_4)+x_1x_2x_3x_4$.

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.

4.6 Algebraic Immunity

4.6.1 Description

A new kind of attacks, called *algebraic attacks*, has been introduced [11], [13], [19]. 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.

Definition 4.6.1. [11], [12], [19], [26] Denote the Boolean function obtained by the product of the Truth Tables of two Boolean functions $f, g \in \mathcal{F}_n$ by $f \cdot g$ *. 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 $\mathbf{1}+f$.

*Note that this product is different from the dot product between two vectors $\mathbf{x}, \mathbf{y}$

A function f should not be used if f or $1 + f$ has a low degree annihilator. If this happens, algebraic attacks [14] can be executed.

Definition 4.6.2. 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.

4.6.2 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 4.6.1. 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;
   }
   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
```

4.7 Global Avalanche Criterion

4.7.1 Description

The Global avalanche criterion (GAC) was introduced in [41] to measure the overall avalanche characteristics of a Boolean function.

Definition 4.7.1. [41] 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:

$$AC_{max}(F) = \max(|\mathsf{AC}(F)(\mathbf{u}, \mathbf{v})|) \; \forall \, \mathbf{u} \neq \mathbf{0} \in \mathrm{V_n}, \; \forall \, \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m} \tag{4.13}$$

2. The *sum-of-squares indicator*, denoted by σ, is the second moment of the autocorrelation coefficients:

$$\sigma(F) = \sum_{(\mathbf{u},\mathbf{v}) \in \mathrm{V_n} \times \mathrm{V_m}} \mathsf{AC}(F)(\mathbf{u}, \mathbf{v})^2 = \frac{1}{2^n} \sum_{(\mathbf{u},\mathbf{v}) \in \mathrm{V_n} \times \mathrm{V_m}} \mathsf{WS}(F)(\mathbf{u}, \mathbf{v})^4 \tag{4.14}$$

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

4.7.2 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 4.7.1. The following program provides the absolute indicator and the sum-of-squares 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-squares 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-squares indicator that can be achieved by
   a Vector Boolean function with the same dimensions is "
   << sigmamax(F) << endl;
  cout << "The minimum sum-of-squares 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-squares 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-squares indicator that can be achieved by a
Vector Boolean function with the same dimensions is 4096
The minimum sum-of-squares indicator that can be achieved by a
Vector Boolean function with the same dimensions is 256
```

The figure 4.8 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	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

Figure 4.8: *Absolute indicator of NibbleSub.*

The figure 4.9 represents the Autocorrelation Spectrum of *NibbleSub* and emphasizes in blue the columns (component functions) in which the maximum sum-of-squares 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	−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
	640	640	640	640	1024	640	1408	640	1408	640	640	640	640	1408	1408

Figure 4.9: *Sum-of-squares indicator of NibbleSub.*

4.8 Linearity Distance

4.8.1 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 [18]). 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.

Definition 4.8.1. 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 [27]:

$$LD(f) = d(f, \mathsf{LS}_n) = \min_{\mathsf{S} \in \mathsf{LS}_n} d(f, \mathsf{S}) \tag{4.15}$$

where:

$$\mathsf{LS}_n = \{f \in \mathcal{F}_n \mid f \text{ has a linear structure } \neq \mathbf{0}\} \tag{4.16}$$

Theorem 4.8.1. *[6] 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:*

$$LD(F) = \min_{\mathbf{v}\neq\mathbf{0}\in \mathrm{V_m}} LD(\mathbf{v}\cdot F) = 2^{n-2} - \frac{1}{4}\cdot AC_{max}(F) \tag{4.17}$$

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 [3] can be seen as an extension of the ideas of attacks based on the presence of linear structures [28]. 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 $NL(F) \geq LD(F)$.

4.8.2 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 4.8.1. 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 result is congruent with the results in example of subsection 3.8.2. We showed that 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.

4.9 Propagation Criterion

4.9.1 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 [35] and it is a generalization of the Strict Avalanche Criterion (SAC) defined by Webster and Tavares in [39].

Definition 4.9.1. $f \in \mathcal{F}_n$ 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.

Definition 4.9.2. 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.

Definition 4.9.3. Let $f \in \mathcal{F}_n$ and $l \in \{1, \ldots, n\}$, f satisfies the propagation criterion of degree l if and only if:

$$f \text{ satisfies the } PC(l) \Leftrightarrow f(\mathbf{x}) + f(\mathbf{x} + \mathbf{u}) \text{ balanced } \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq l \tag{4.18}$$

Theorem 4.9.1. *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:*

$$f \textit{ satisfies } PC(l) \iff r_f(\mathbf{u}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq l \tag{4.19}$$

Definition 4.9.4. $F \in \mathcal{F}_{n,m}$ 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:

$$r_F(\mathbf{u}, \mathbf{v}) = 0,\ \forall\, \mathbf{u} \in \mathrm{V_n},\ 1 \leq wt(\mathbf{u}) \leq l,\ \forall\, \mathbf{v} \neq \mathbf{0} \in \mathrm{V_m} \tag{4.20}$$

4.9.2 Library

The method used to obtain this criterion is the following:

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

Example 4.9.1. 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 figure 4.10 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

Figure 4.10: *Propagation Criterion of* $x_1x_2 + x_3x_4$.

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

4.10 Summary

A list of the member functions related to these criteria may be found in Table 4.1.

Table 4.1: 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 $PC(l)$

Table 4.2 lists the member functions related to bounds and other properties of above criteria.

Table 4.2: 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-squares indicator
NTL::ZZ minsigma(VBF& F)	Minimum possible sum-of-squares indicator
NTL::RR ldmax(VBF& F)	Maximum possible linearity distance

Chapter 5

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 [5]. 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 [2], DES [16] and AES [15]. 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.

5.1 Equality Testing

5.1.1 Description

Definition 5.1.1. 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.

5.1.2 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 5.1.1. 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;
   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
```

5.2 Composition Function

5.2.1 Description

Definition 5.2.1. 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 $G \circ F(\mathbf{x}) = G(F(\mathbf{x}))\ \forall\, \mathbf{x} \in \mathrm{V_n}$. See figure 5.1.

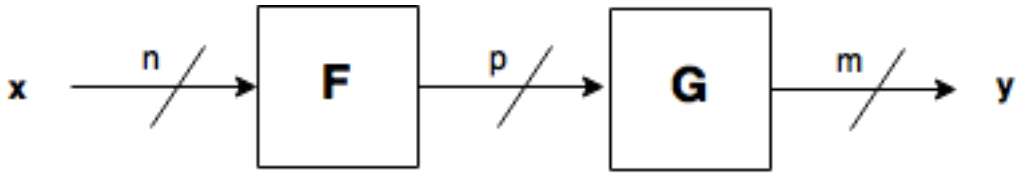


Figure 5.1: *Composition.*

5.2.2 Library

It can be obtained with the following method:

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

Example 5.2.1. 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 section "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 5.2.2. 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;

   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 [20] 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 [20] 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
```

5.3 Functional Inverse

5.3.1 Description

Definition 5.3.1. 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. See figure 5.2.

5.3.2 Library

If a Vector Boolean Function $F \in \mathcal{F}_{n,n}$ is invertible, then we can find its inverse with the following method:

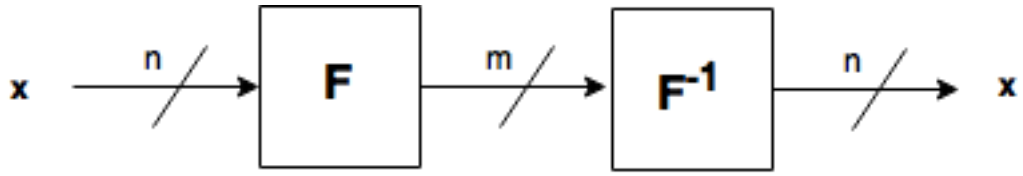


Figure 5.2: *Inverse*.

```
void inv(VBF& X, VBF& F)
```

Example 5.3.1. 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]
]
```

5.4 Sum

5.4.1 Description

Definition 5.4.1. 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: $\mathsf{T}_{F+G} = \mathsf{T}_F + \mathsf{T}_G$.

5.4.2 Library

It can be obtained with the following method:

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

Example 5.4.1. 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;
      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
ACmax(F+G)=4
LD(F+G)=3
```

These results are congruent with the properties of changing one bit of the Truth Table:

- $NL(F+G) = NL(F) - 1 = 6 - 1 = 5$.
- $AC_{max}(F+G) = AC_{max}(F) + 4 = 0 + 4 = 4$.
- $LD(F+G) = LD(F) - 1 = 4 - 1 = 3$.

5.5 Direct Sum

5.5.1 Description

Definition 5.5.1. 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 $(F_1 \oplus F_2)((\mathbf{x_1}, \mathbf{x_2})) = F_1(\mathbf{x_1}) + F_2(\mathbf{x_2})$. See figure 5.3.

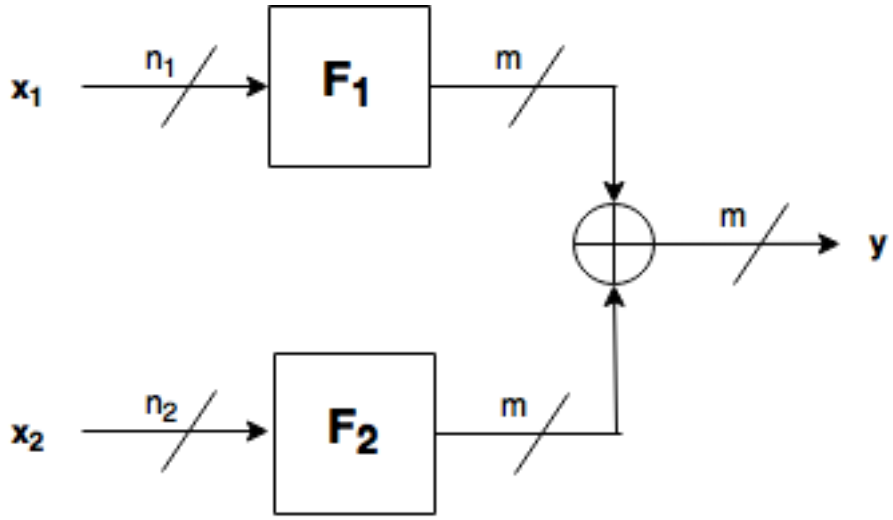


Figure 5.3: *Direct Sum.*

5.5.2 Library

The method included in VBF to perform this construction is the following:

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

Example 5.5.1. 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;
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

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 [36] and others derived by Jose Antonio Alvarez:

- $wt(F \oplus G) = 2^6 \cdot 32 + 2^6 \cdot 32 - 2 \cdot 32 \cdot 32 = 2048$.
- $deg(F \oplus G) = \max\{3, 3\} = 3$.
- F is 1-resilient, G is 2-resilient, and $F \oplus G$ is $(1 + 2 + 1)$-resilient.
- $R(F \oplus G) = 16 \cdot 32 = 512$ because F and G are Boolean functions.
- $NL(F \oplus G) = 2^{12-1} - \frac{1}{2} \cdot 512 = 1792$.
- $AC_{max}(F \oplus G) = \max\{32 \cdot 64, 64 \cdot 64\} = 4096$.
- $LD(F \oplus G) = 2^{12-2} - \frac{1}{4} \cdot 4096 = 0$.
- $\max\{3, 2\} \leq AI(F \oplus G) = 3 \leq \min\{\max\{3, 3\}, 3 + 2\}$.

5.6 Concatenation

5.6.1 Description

Definition 5.6.1. 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 $(\mathbf{x}, x_{n+1}) \to (x_{n+1} + 1) F_1(\mathbf{x}) + x_{n+1} F_2(\mathbf{x})$ where $\mathbf{x} \in \mathrm{V_n}$.

5.6.2 Library

The method included in VBF to perform this construction is the following:

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

Example 5.6.1. 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"

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)) {
    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:

- $wt(F|_cG) = 8 + 8 = 16$.
- $deg(F|_cG) = 3 \leq 1 + \max\{3, 3\} = 1 + 3 = 4$.
- F is 0-resilient, G is 0-resilient, and $F|_cG$ is 0-resilient.
- $NL(F|_cG) = 8 \geq 4 + 4 = 8$.
- If $AI(F) = AI(G) = 2$, then $AI(F|_cG) = 2 \leq 2 + 1$.

5.7 Concatenation of Polynomials in ANF

5.7.1 Description

Definition 5.7.1. 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 $(x_1, \ldots, x_{n_1}, x_{n_1+1}, \ldots, x_{n_1+n_2}) \to F_1(x_1, \ldots, x_{n_1}) + F_2(x_{n_1+1}, \ldots, x_{n_1+n_2})$ where $\mathbf{x} \in \mathrm{V}_{\mathrm{n_1+n_2}}$.

5.7.2 Library

The method included in VBF to perform this construction is the following:

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

Example 5.7.1. 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
```

5.8 Addition of Coordinate Functions

5.8.1 Description

Definition 5.8.1. 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 $\mathbf{v} \in \mathrm{V}_{m_1+m_2}$,$\mathbf{v_F} \in \mathrm{V}_{m_1}$ and $\mathbf{v_G} \in \mathrm{V}_{m_2}$ so that $\mathbf{v} = (\mathbf{v_F}, \mathbf{v_G})$. See figure 5.4.

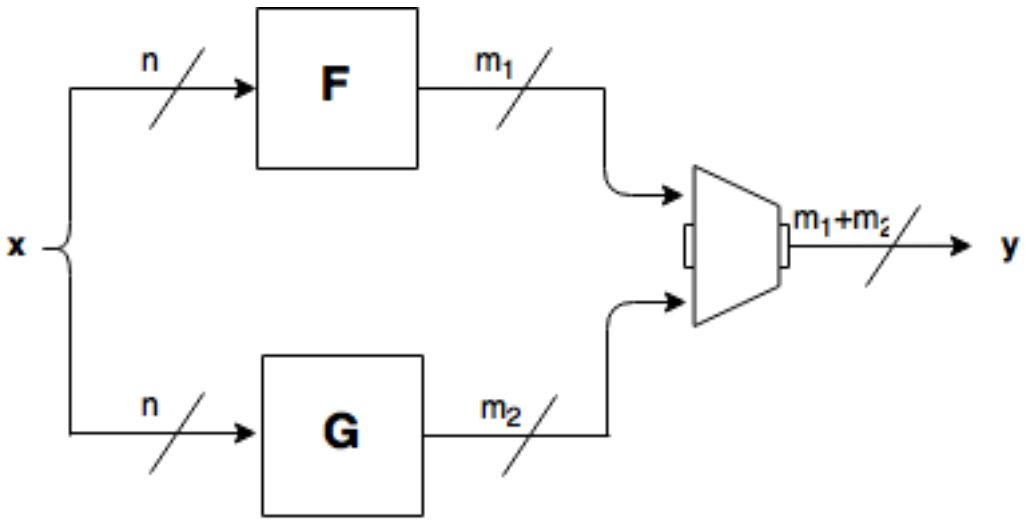


Figure 5.4: *Adding Coordinate functions.*

5.8.2 Library

This construction can be obtained with the following method:

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

Example 5.8.1. The following program provides the Truth Tables of the different intermediate constructions that allow to obtain CLEFIA S_0 8×8 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 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;

   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);
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 section "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 $\mathbf{y_1} \in \mathcal{F}_{8,4}$.

5.9 Bricklayer

5.9.1 Description

Definition 5.9.1. 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 $\mathbf{u_1} \in \mathrm{V_{n_1}}$, $\mathbf{u_2} \in \mathrm{V_{n_2}}$ and $\mathbf{u} = (\mathbf{u_1}, \mathbf{u_2})$, $\mathbf{v_1} \in \mathrm{V_{m_1}}$, $\mathbf{v_2} \in \mathrm{V_{m_2}}$ and $\mathbf{v} = (\mathbf{v_1}, \mathbf{v_2})$. See figure 5.5.

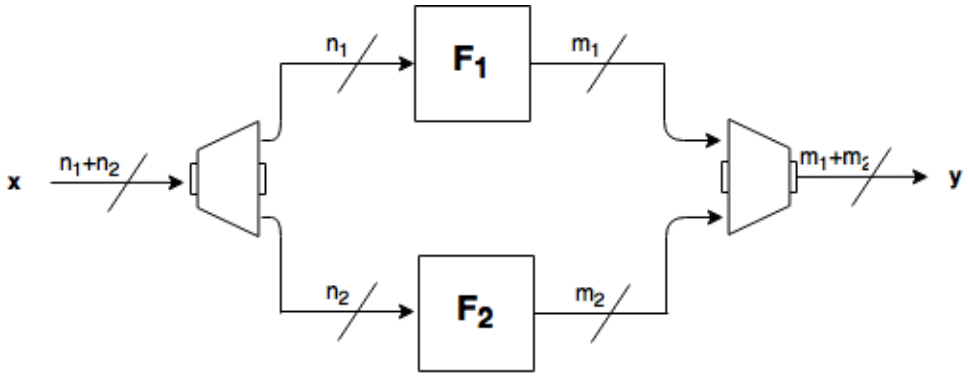


Figure 5.5: *Bricklayer*.

5.9.2 Library

It can be obtained with the following method:

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

Example 5.9.1. 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 8×8 S-box composed of smaller pseudo-randomly generated 4×4 mini S-boxes (the P-box and the Q-box) as represented in figure 5.6.

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;
```

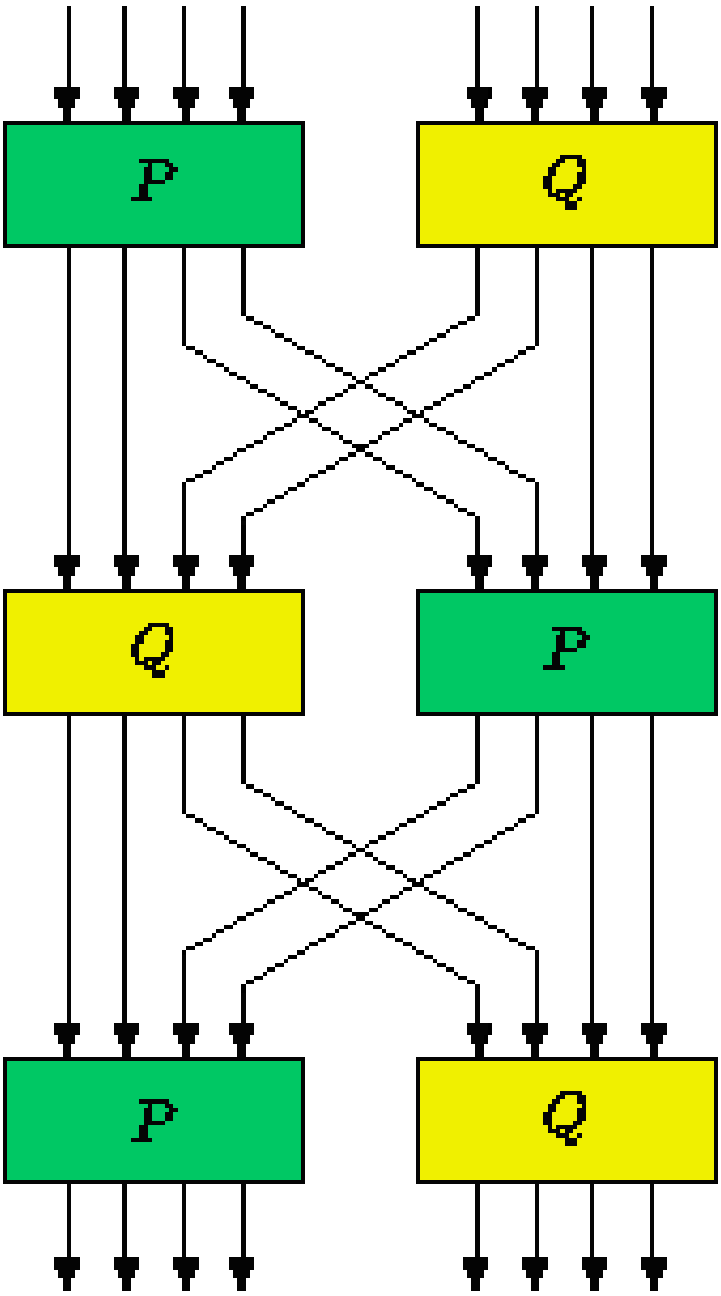


Figure 5.6: *KHAZAD S-box construction.*

```
      return 0;
   }
   inputp >> Tp;
   P.puttt(Tp);
   inputp.close();

   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".

Table 5.1: Results of spectral radius(r),NL, lp, dp, AC_{max} and LD for bricklayer of P and Q mini S-boxes.

S-box	r	NL	lp	dp	AC_{max}	LD
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 5.9.2. 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;
}

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
```

5.10 Summary

Table 5.2 lists the member functions related to the previous characterizing elements.

Table 5.2: Constructions over VBF.

SYNTAX	DESCRIPTION
long operator==(VBF& F, VBF& G)	Returns 1 if F and G are equal 0 otherwise
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)	$X(\mathbf{x}, \mathbf{y}) = F(\mathbf{x}) + G(\mathbf{y})$
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& G)	$X = (F, G)$
void bricklayer(VBF& X, VBF& F, VBF& G)	$X = F \vert G$

Bibliography

[1] 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. [cited at p. 2]

[2] Carlisle M. Adams and Stafford E. Tavares. 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, 1993. [cited at p. 95]

[3] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In *Advances in Cryptology CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, pages 2–21, 1990. [cited at p. 90]

[4] G. Xiao C. Ding and W. Shan. *The Stability Theory of Stream Ciphers*. Springer-Verlag, Berlin, 1991. Lecture Notes in Computer Science Volume 561. [cited at p. 73]

[5] C. Carlet. On the secondary constructions of resilient and bent functions. *Progress in Computer Science and Applied Logic*, 23:3–28, 2004. [cited at p. 95]

[6] C. Carlet. Boolean functions for cryptography and error correcting codes. Technical report, University of Paris, BP 105 - 78153, Le Chesnay Cedex, FRANCE, 2008. [cited at p. 90]

[7] Claude Carlet. 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, 2008. [cited at p. 78]

[8] F. Chabaud and S. Vaudenay. Links between differential and linear cryptanalysis. In *Advances in Cryptology EUROCRYPT' 94*, pages 356–365, 1995. [cited at p. 5, 42]

[9] David Chaum and Jan-Hendrik Evertse. Crytanalysis of des with a reduced number of rounds: Sequences of linear factors in block ciphers. In *CRYPTO*, pages 192–211, 1985. [cited at p. 51]

[10] Lusheng Chen, Fang-Wei Fu, and Victor K.W. Wei. 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, 2002. [cited at p. 82]

[11] 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. [cited at p. 84]

[12] 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. [cited at p. 84]

[13] Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In *EUROCRYPT*, pages 345–359, 2003. [cited at p. 84]

[14] Nicolas T Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Advances in Cryptology ASIACRYPT 2002*, pages 267–287. Springer, 2002. [cited at p. 85]

[15] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002. [cited at p. 95]

[16] Des. Data encryption standard. In *In FIPS PUB 46, Federal Information Processing Standards Publication*, pages 46–2, 1977. [cited at p. 62, 95]

[17] Jan-Hendrik Evertse. Linear structures in blockciphers. In *EUROCRYPT*, pages 249–266, 1987. [cited at p. 51]

[18] Jan-Hendrik Evertse. Linear structures in blockciphers. In David Chaum and WynL. Price, editors, *Advances in Cryptology EUROCRYPT 87*, volume 304 of *Lecture Notes in Computer Science*, pages 249–266. Springer Berlin Heidelberg, 1988. [cited at p. 89]

[19] J.-C. Faugère and G. Ars. An algebraic cryptanalysis of nonlinear filter generators using Gröbner bases. Technical report, INRIA 4739, 2003. [cited at p. 84]

[20] K.C. Gupta and P. Sarkar. Improved construction of nonlinear resilient s-boxes. *Information Theory, IEEE Transactions on*, 51(1):339–348, Jan 2005. [cited at p. 101]

[21] T. Jakobsen and L.R. Knudsen. The interpolation attack on block ciphers. In *Proceedings of Fast Software Encryption'97, Lecture Notes in Computer Science 1267*, pages 28–40, 1997. [cited at p. 31]

[22] Xuejia Lai. Higher order derivatives and differential cryptanalysis. In Richard E. Blahut, Jr. Costello, Daniel J., Ueli Maurer, and Thomas Mittelholzer, editors, *Communications and Cryptography*, volume 276 of *The Springer International Series in Engineering and Computer Science*, pages 227–233. Springer US, 1994. [cited at p. 70]

[23] Xuejia Lai. Additive and linear structures of cryptographic functions. In Bart Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 75–85. Springer Berlin Heidelberg, 1995. [cited at p. 51]

[24] Mitsuru Matsui. Linear cryptanalysis method for des cipher. In *EUROCRYPT*, pages 386–397, 1993. [cited at p. 73]

[25] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *Advances in Cryptology EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin Heidelberg, 1994. [cited at p. 74]

[26] Willi Meier, Enes Pasalic, and Claude Carlet. Algebraic attacks and decomposition of boolean functions. In Christian Cachin and JanL. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 474–491. Springer Berlin Heidelberg, 2004. [cited at p. 84]

[27] Willi Meier and Othmar Staffelbach. Nonlinearity criteria for cryptographic functions. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology EUROCRYPT 89*, volume 434 of *Lecture Notes in Computer Science*, pages 549–562. Springer Berlin Heidelberg, 1990. [cited at p. 89]

[28] Kaisa Nyberg. Perfect nonlinear S-boxes. In DonaldW. Davies, editor, *Advances in Cryptology EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 378–386. Springer Berlin Heidelberg, 1991. [cited at p. 90]

[29] Kaisa Nyberg. Differentially uniform mappings for cryptography. In *EUROCRYPT*, pages 55–64, 1993. [cited at p. 46]

[30] 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. [cited at p. 70, 73]

[31] 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. [cited at p. 5]

[32] J. Pieprzyk and G. Finkelstein. Towards effective nonlinear cryptosystem design. *Computers and Digital Techniques, IEE Proceedings E*, 135(6):325–335, Nov 1988. [cited at p. 73]

[33] K. Pommerening. Fourier Analysis and Boolean Maps – A Tutorial, 2005. http://www.staff.unimainz.de/pommeren/Kryptologie/Bitblock/A_Nonlin/Fourier.pdf. [cited at p. 79]

[34] B. Preneel. Analysis and design of cryptographic hash functions. ph.d. dissertation, katholieke universiteit leuven, 1993. [cited at p. 55]

[35] Bart Preneel, Werner Van Leekwijck, Luc Van Linden, René Govaerts, and Joos Vandewalle. Propagation characteristics of boolean functions. In *EUROCRYPT*, pages 161–173, 1990. [cited at p. 91]

[36] P. Sarkar and S. Maitra. Construction of nonlinear boolean functions with important cryptographic properties. In *EUROCRYPT*, pages 488–511, 2000. [cited at p. 109]

[37] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30(5):776–, 1984. [cited at p. 82]

[38] VBF: Vector Boolean Functions Library: User Manual and Analysis of Cryptanalytic Algorithms, 2015. http://vbflibrary.tk. [cited at p. i]

[39] A. F. Webster and S. E. Tavares. On the design of S-boxes. In Hugh C. Williams, editor, *Advances in Cryptology - Crypto '85*, pages 523–534, Berlin, 1986. Springer-Verlag. Lecture Notes in Computer Science Volume 218. [cited at p. 91]

[40] Guo-Zhen Xiao and James L. Massey. A spectral characterization of correlation-immune combining functions. *IEEE Transactions on Information Theory*, 34(3):569–, 1988. [cited at p. 82]

[41] 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. [cited at p. 86]