ECMI Modelling Week, July 17–24, 2016, Sofia, Bulgaria

Group 9

S-Box Analysis

Bernardo Amaral

University of Coimbra, Portugal
bernardoamaral6@gmail.com

Stanislav Darachev

Sofia University, Bulgaria
darachev@abv.bg

Maximilian Ludvigsson

Chalmers University of Technology, Sweden
maxiludvigsson@gmail.com

Ronak Molazem

University of Joseph Fourier, Grenoble, France
rmolazem17@gmail.com

Alejandro Molero Casanova

Autonomous University of Barcelona, Spain
94.molero.a@gmail.com

Instructor: **Stela Zhelezova**

Institute of Mathematics and Informatics, BAS, Bulgaria
stela@math.bas.bg

Abstract. The present report is the result of a week-long workshop within the 30th ECMI Modeling Week organized by Faculty of Mathematics and Informatics, Sofia University ”St. Kliment Ohridski” in cooperation with the European Consortium for Mathematics in Industry (ECMI) and Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, July 17 - 24, 2016, Sofia, Bulgaria.

Our task was to implement a tool for evaluation of some cryptographic properties of an S-box using different computational techniques. We use two different algorithms to obtain considered characteristics and correspondingly we make two different implementations.

9.1 Introduction

The block ciphers are a particular class of cryptographic algorithms. They are fundamental to present cryptography and they are the most widely used cryptographic primitive. Shannon [9] introduced the ideas of confusion and diffusion for practical cipher design. The basic operations of substitution and permutation are of special interest to achieve these ideas. Typically, substitution is achieved with a suitably designed look-up table, substitution box, or simply S-box. So S-boxes introduce non-linearity on word level. What is it that makes one cryptographic algorithm better than another? The S-boxes involved in the algorithm are of major importance. They have to be carefully chosen to posses specific security properties.

The level of security achieved by cryptographic algorithms based on S-boxes is measured by the quality of combinatorial properties within the relevant Boolean functions. The selection of S-boxes with strong cryptographic properties reduces the effectiveness of advanced cryptanalytic attacks, including linear cryptanalysis and differential cryptanalysis. The idea of linear cryptanalysis [7] is to find an approximation of the relationship between the plain text the cipher text and the key, i.e. presenting a linear dependence involving the tree parties and rating of probability this dependence is true. The goal of this attack is discovering of part of the key as big as possible. The resistance of S-box to the linear cryptanalysis is close related to the coefficients of the WHT of all non-zero linear combination of the relevant component functions. They are used to calculate the Nonlinearity of an S-box. The bigger is the Nonlinearity the unsuccessful is the attack.

Any Boolean function of low algebraic order can be described and evaluated in terms of a relatively low number of coefficients in its ANF even if the number of input variables is large. Therefore the higher is the algebraic degree of an S-box, the difficult is to approximate it with vectorial Boolean function with low degree which is the low order approximation attack [8].

The differential cryptanalysis [2] is based on analysis of the effect of particular differences in plaintext pairs on the differences of the resultant ciphertext pairs. These differences are used to assign probabilities to the possible keys in the cipher and to locate the most probable key. If the DDT is flatter then the probabilities for the distinct differences are similar and no information can be extracted. The smallest is the differential uniformity (the maximal value in the DDT) the lowest is the possibility of successive differential cryptanalysis.

The Global Avalanche Characteristic (GAC) [16] contributed to the strength of the S-box against various attacks including differential and linear ones. It is measured by two indicators. The smallest are their values the better is the GAC of an S-box.

There are some publicly available tools performing evaluation of different S-box criteria. The `Boolfun package` in [6] and the `BooleanFunctions` module in `Sage` [10] have functionality for cryptographic analysis of Boolean functions. The `Sbox` module in Sage has many options but from the cryptographic point of view it can calculate only the difference distribution table and the linear approximation matrix of an S-box while `SET` (S-box Evaluation Tool) [11] can evaluate a wide set of cryptographic properties of Boolean functions and S-boxes. It deals with the properties we consider and some other as well. Between them are the Branch number and Transparency order of an S-box.

In the newest tool [14] authors propose as performance indexes of S-box Nonlinearity, Balansedness, Strict avalanche criterion and other 3 probabilistic criteria. The Strict avalanche criterion is less useful [16] than the Global Avalanche Criterion which we use.

Our goal in this project will be to develop a toolbox of our own to evaluate the security level of an S-box. In the following sections, we will define more precisely the properties mentioned before, and we will see how to implement them in two different programming languages (Matlab and C++) in order to develop this toolbox. We will then discuss the pros and cons of each implementation and compare them.

9.2 Definitions

A Boolean function (BF) is a discrete function $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ so that maps n binary inputs to one binary output. Boolean functions can be represented using several different forms. The basic representation is by Binary Truth Table (TT). The TT is a list of the output for each of the 2^n possible inputs to that Boolean function.

S-boxes can be considered as vectorial Boolean functions $S : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^m$, therefore the S-box cryptographic properties can be derived from their involved Boolean functions. Thus, from now on, the definitions will be given for single Boolean functions, and in the end, we will extend these definitions to S-boxes.

The Hamming weight of a Boolean function is defined as the number of 1s in the TT: $wt(f) = \sum f(x)$.

The Hamming distance between two functions $f(x)$ and $g(x)$ is defined as the number of TT positions in which they differ $d(f, g) = wt(f \oplus g)$.

The Algebraic Normal Form (ANF) describe a Boolean function in terms of a unique `XOR` sum of `AND` products of the input variables:

$$f(x) = a_0 \oplus a_1.x_1 \oplus a_2.x_2 \oplus \ldots \oplus a_{1,2}.x_1.x_2 \oplus \ldots \oplus a_{1,2,n}.x_1.x_2.\ldots.x_n$$

From a function ANF one can obtain its Algebraic Degree (deg_f) which is defined as the degree of the largest product term that exists in the ANF [12]. The degree of a product term refers to the number of variables it includes. Algebraic Degree of BF is a measure of its complexity.

The Walsh-Hadamard Transform (WHT) uniquely determines a Boolean function in terms of its correlation with all linear functions. It is a real-valued function defined for all $\omega \in \mathbb{F}_n^2$ as:

$$W_f(\omega) = \sum (-1)^{f(x) \oplus x.\omega}$$

We notice that this function, given ω, simply counts how many times our boolean function f "looks like" the linear function $\omega.x$. In this sense, it is also some sort of Fourier transformation. Its values are bounded $-2^n \leq W_f(\omega) \leq 2^n$. The nonlinearity of the Boolean function can be measured using the maximum absolute value of its WHT [13]:

$$N_f = 2^{n-1} - \frac{1}{2} max_\omega |W_f(\omega)|$$

The smaller the minimum distance to any affine function, the greater the nonlinearity.

Imbalance is defined as the Hamming distance to a balanced function. An imbalanced function is correlated to a constant function thus the balancedness is a fundamental cryptographic properties of BF. A function is call balanced if the half of the function values are equal to one, and it is easy to see that this is the same as asking that $W_f(0) = 0$.

The auto-correlation function (AC) of the Boolean function gives information on its differential properties. It is real-valued function defined for all $\omega \in \mathbb{F}_n^2$ as:

$$r_f(\omega) = \sum (-1)^{f(x) \oplus f(x \oplus \omega)}$$

For every Boolean function $r(0) = 2^n$ and for the other possible inputs the maximal value of the autocorrelation is less or equal to 2^n.

In [16] the global avalanche characteristics (GAC) is defined. It consists of the next two indicators:

- Absolute indicator: $|AC_f| = max|\{r(\omega)|\}$, $\omega \neq 0$;
- Sum-Of-Square indicator: $\sigma_f = \sum (r(\omega))^2$, $\omega \neq 0$.

Now, in order to obtain the equivalent cryptographic properties of an S-box $S : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^m$ we need to consider all non-zero linear combinations F, $|F| = 2^m - 1$ of its component functions. We consider the next properties that are widely accepted as important for cryptographically strong S-boxes:

- Algebraic degree: $deg_F = min\{deg_{f_j}\}, j = 1, \dots, 2^m - 1$.
- Nonlinearity: $N_F = min\{N_{f_j}\}, j = 1, \dots, 2^m - 1$.
- Balancedness: an S-box is balanced if each of its component BF and their linear combinations are balanced.
- Maximum Absolute Indicator: $|AC_F| = max_f|\{r_{f_j}(\omega)|\}$, $\omega = 1, \dots, 2^n$, $j = 1, \dots, 2^m - 1$.
- Maximum Sum-of-squares: $\sigma_F = max_f\{\sum (r_{f_j}(\omega)^2\}$, $\omega = 1, \dots, 2^n$, $j = 1, \dots, 2^m - 1$.

There is one last property to define before starting with the implementations. Given an S-Box $S : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^m$, and given $\Delta \in \mathbb{F}_2^n$ and $\delta \in \mathbb{F}_2^m$, we define the differential characteristic of F with input difference Δ and output difference δ as:

$$D_\Delta^\delta(S) = \#\{x \in \mathbb{F}_2^n \mid S(x) \oplus S(x \oplus \Delta) = \delta\}.$$

The set of all differential characteristics is called differential distribution table. The maximum value Δ_S in this table is the differential uniformity parameter.

This is probably the only parameter that we define for the whole S-box at once, instead of defining it first for the Boolean functions of the S-box, and then extending the definition using the set of all linear combinations of this Boolean functions. The reason is that, what matters here is the relation between input differences and their exact output differences. The differential uniformity is also important in order to describe the strength of the S-box against differential cryptanalysis: the flatter the differential distribution table (and therefore, the lower the differential uniformity parameter), the better.

9.3 Implementations

There are several ways of implementing the relevant properties described above. The main computing problems arise when we try to compute the ANF form, the WHT, the AC function and the differential distribution table of the S-box, because it is needed to perform a huge amount of calculations. Some properties have a straightforward and fast implementation while some demands more creativity. The following will describe the implementation details.

9.3.1 Balancedness

If $W_f(0) = 0$ the Boolean function is balanced. We examine the Walsh coefficients for the zero input of S ($W_S(0)$) and if all of them are zeroes we conclude that the S-box is balanced.

9.3.2 Nonlinearity

The main functions of a S-box is to contribute nonlinearity to the encryption algorithm. One of the most common attacks on a block cipher is to approximate the S-boxes with an affine function in order to reconstruct the key (linear cryptanalysis). In order to test how resistant an S-box is against this, the Walsh-Hadamard transform (WHT) is used to define a nonlinearity measure.

Matlab Implementation

Let us consider the Walsh-Hadamard transform as defined in the introduction,

$$W_f(\omega) = \sum (-1)^{f(x) \oplus x.\omega}$$

or also

$$W_f(\omega) = \langle [\hat{\omega}], [\hat{f}] \rangle$$

where $[\hat{f}]$ and $[\hat{\omega}]$ are the polarity truth tables of our Boolean function f and the linear Boolean function $\omega.x$ and $\langle \, , \, \rangle$ denotes the scalar product. Thus, we could be able to get the whole Walsh-Hadamard spectrum (that is, a vector containing all values of $W_f(\omega)$) if we use a matrix, called the Walsh-Hadamard matrix, such that the $\omega-$th row was the polarity truth table of the linear function $\omega.x$, for each $\omega \in \mathbb{F}_2^n$.

For $n = 1$, the Walsh-Hadamard matrix, that we will denote as H_1, clearly is:

$$H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

For a general n, we notice that given $\omega = (\omega_1, ..., \omega_{2^n}) \in \mathbb{F}_2^n$, $\omega^* = (\omega_2, ..., \omega_{2^n}) \in \mathbb{F}_2^{n-1}$, then

$$[\hat{\omega}] = \big([\hat{\omega^*}], [\hat{\omega^*}]\big)$$

when $\omega_1 = 0$, and

$$[\hat{\omega}] = \big([\hat{\omega^*}], -[\hat{\omega^*}]\big)$$

when $\omega_1 = 1$. But, if H_{n-1} is the Walsh-Hadamard matrix given by all the $\omega* \in \mathbb{F}_2^{n-1}$, then its clear with the relation described that:

Lemma 1 *The Walsh-Hadamard matrix for Boolean functions over $\mathbb{F}_2^n$ can be recursively defined as:*

$$\begin{aligned} H_0 &= 1 \\ H_n &= \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix}. \end{aligned}$$

The WHT transform is therefore expressed in matrix notations as $[W_f] = H_n.[\hat{f}]$, which is something very suitable in order to do an implementation in Matlab, given its simple syntax for matrix calculation and manipulation. So, to use the program advantages we change the TT of S in polarity TT by $[\hat{f}_i] = 1 - 2.[f_i]$ and proceed with the matrix multiplication. When a vectorial Boolean function is considered the expression become $[W_F] = H_n.[\hat{F}]$, where $\hat{F}$ is the matrix with all non-zero linear combination of component S-box functions represented in polarity TT.

C++ Implementation

Despite being straightforward to do, calculating the Walsh-Hadamard spectrum using matrix multiplication has its downsides, which are the amount of memory required (the Walsh-Hadamard matrix dimension is $2^n \times 2^n$) and the number of operations required (about $O(2^{2n})$, according to [5]). Also, trying to implement the same procedure in C++ as the one in Matlab would be highly disadvantageous, because C++ does not provide an easy way to operate with matrices. Therefore, another possibility is studied.

Given the polarity TT $[\hat{f}]$ of a Boolean function, let h_1 and h_2 be the first and the second halves of this vector. Notice that the matrix expression

$$[W_f] = H_n.[\hat{f}]$$

might be formally written as

$$[W_f] = \begin{pmatrix} H_{n-1}h_1 + H_{n-1}h_2 \\ H_{n-1}h_1 - H_{n-1}h_2 \end{pmatrix} = \begin{pmatrix} H_{n-1}(h_1 + h_2) \\ H_{n-1}(h_1 - h_2) \end{pmatrix}$$

This gives us a recursive rule, known as the *Butterfly Algorithm*[1]:

[1]The Butterfly Algorithm, in general, is an algorithm used to compute finite Fourier transformations. Given a function $f : U \longrightarrow R$ over an open set U, a *sample* of f is a function

Lemma 2 (Butterfly Algorithm) *Take the polarity TT $[\hat{f}]$ of a Boolean function f, halve it into h_1 and h_2 and consider the lists $h_1 + h_2$ and $h_1 - h_2$. Now, for each list, repeat the process. The result, after n iterations, is the Walsh-Hadamard spectrum of f.*

This algorithm is firstly mentioned with this very name on a 1969 MIT's report (check [15]). The following picture describes the basic procedure of this algorithm (notice that this is done here with the TT of the Boolean function, not the polarity TT, so the result is not the Walsh-Hadamard transformation).

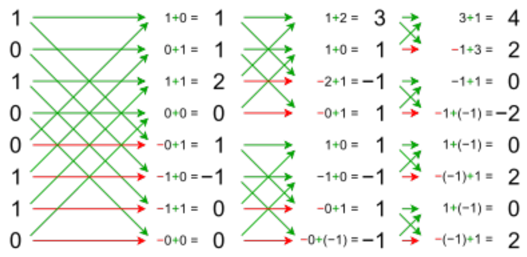

Figure 9.1: Example of Butterfly Algorithm

From the perspective of a C++ implementation, this algorithm is relatively easy to implement. But, as it has been said, it also has another advantage, compared to the matrix multiplication algorithm used in the Matlab implementation: it requires less memory (its is not needed to calculate any matrix, so nothing but the TT of the Boolean function must be stored) and less operations ($O(n2^n)$ according to [5]).

9.3.3 Algebraic Degree

In order to protect an encryption system against Low order approximation attacks, the used S-boxes need to have a high algebraic degree. That is, the Algebraic Normal Form (ANF) of the Boolean function is a polynomial of a high degree.

Matlab Implementation

The ANF of the Boolean function is easiest calculated in Matlab by utilizing that the transform is linear and thus have a matrix representation. We therefore want to find a transformation matrix A_n of size $2^n \times 2^n$ that has the property

$f_s : X \longrightarrow R$, where X is a discrete subset of U, where $f_s(x) = f(x)$. Usually, in computer science or telecommunications, given an electric signal represented by a function f, it is only possible to work with a sample of f, because of the digital electric components involved to treat the signal. Therefore, the study of the existent frequencies in the signal (which would be usually done by calculating the Fourier transformation of f) has to be performed using only the information that we have about the signal, which is the discrete function f_s. Applying the Butterfly Algorithm to f_s gives us what we call the finite Fourier transformation, which is an spectral measure of our original function f based on the information provided by f_s.

$[ANF_f] = [A_n] \cdot [f](mod\ 2)$. One can show that this matrix can be constructed by the following recursive relation:

$$A_0 = 1$$
$$A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix}$$

After the coefficients of the algebraic form is found, we check which terms are present by finding the indices of the terms with coefficient not equal to 0. After we found these indices we want to know what algebraic order the different indices correspond to. This is done by summing up the non-zero elements in the binary representation of the indices (bear in mind that the TTs for all non-zero linear combinations F are in lexicographical order).

Finally, the `max` function is used to identify the total order of the polynomial.

C++ Implementation

For the C++ implementation, we have firstly avoided to calculate the ANF of our Boolean functions, using the one calculated with Matlab to find the degree. We are going to discuss now how we find the degree given this ANF, and later, we will present an algorithm that we have finally followed to calculate the ANF with C++ and find, at the same time, the degree.

Our finding of the degree, given the ANF provided by Matlab, consists of the following steps:

- The first step is to create a vector that it has the same size of the ANF form of the Boolean function, the degree vector (if a Boolean function as n binary inputs then the ANF form has a size of 2^n). This vector contains the degree of each position in the ANF form. The construction of this vector is done using this procedure: Initially the degree vector as only one element, which is the 0. After that, recursively, all the elements in the vector are going to be passed to an auxiliary vector, be summed by one and, at last, the elements of the auxiliary vector are going to be added into the degree vector until we reach the size of the ANF form. The following example shows the result of the degree vector for cases n=1,2,3.

$$n=1:\begin{bmatrix}0\\1\end{bmatrix}\qquad n=2:\begin{bmatrix}0\\1\\1\\2\end{bmatrix}\qquad n=3:\begin{bmatrix}0\\1\\1\\2\\1\\2\\2\\3\end{bmatrix}$$

- After the construction of the degree vector the program is going through the ANF form from the last element to the first one to see if there is a one or a zero, deleting the corresponding element from the ANF form and from the degree vector afterward. There is a variable which records the highest degree, *max*, which starts at zero. If there is a one, then we are going to compare the degree from that position to the one that is in the *max* variable. If *max*'s degree is less than the degree from that position, then *max* =degree. The program will compare every one of those positions and returns *max* as the final result. The following example shows what's the algebraic degree of a random ANF form:

$$\text{ANF Form}:\begin{bmatrix}1\\0\\0\\1\\1\\0\\0\\0\end{bmatrix}\qquad \text{Degree vector}:\begin{bmatrix}0\\1\\1\\2\\1\\2\\2\\3\end{bmatrix}\qquad \text{Algebraic Degree: } 2$$

Obviously, this algorithm (which takes $O(n2^n)$ to be performed) we have firstly developed has too many inconveniences; the most important one, is that we depend on the Matlab implementation, diminishing the opportunities that we could have using C++. Before moving forward, let us notice that the values of the degree vector generated are just the sum of the bits in base 2 of the position of each value.

We then have come with another algorithm, called the *Moebius transformation* (see [5]), which takes the same number of operations as the previous one, but allows us to calculate the ANF inside the C++ implementation and, by doing a little adaptation, the degree of the Boolean function at the same time. The Moebius transformation is easily derived from the matrix multiplication expression as a recurrent relation, and it goes as follows:

Lemma 3 (Moebius Transformation) *Given the TT $[f]$ of a Boolean function f over $\mathbb{F}_2^n$, $[f] = (a_0, ..., a_{2^n-1})$ we consider $[f]_1 = (a_0 \oplus a_1, a_1 \oplus a_2, ..., a_{2^n-2} \oplus a_{2^n-1})$ and we repeat the process for this and the following vectors. After n iterations, the set of all the first elements of each generated vector is the ANF list of coefficients of the Boolean function.*

In order to obtain the degree, given this algorithm, we add another step in each iteration. If the first element of the generated vector in the i-th iteration is 1, that means that the coefficient associated to the i-th monomial in the ANF is also 1, and therefore, that this monomial might contribute to the degree of the ANF. Now, the degree of the monomial is no more than the sum of the bits in base 2 of the value i. Therefore, every time we encounter a 1 in the first position of a new generated vector, we will examine in which iteration are we, sum the bits of this order of iteration, and store it as the degree of the ANF form if it is greater than the previous value of the degree stored.

9.3.4 Autocorrelation

The autocorrelation function shows how much a signal correlates with itself in different points in time. This means that it, like the Walsh transform, can gives us information about the periodicity of a signal. Ciphers with a lower autocorrelation value is thus better cryptographically. It is defined as follows,

$$r_f(\omega) = \sum_{x \in F_2^n} (-1)^{f(x) \oplus f(x \oplus \omega)}$$

Matlab Implementation

The Wiener-Khintchine theorem [1] states that the AC can be calculated directly as the inverse WHT of the power spectrum, which is the square of the polarity WHT. In matrix representation, this is given by:

$$[r_f] = 2^{-n}[H_n]([W_f])^2$$

with $(W_f)^2$ being the element wise square of the vector W_f [3]. For the all non-zero linear combination this becomes:

$$[r_F] = 2^{-n}[H_n]([W_F])^2$$

C++ Implementation

Since we can, in C++, calculate the WHT using the Butterfly Algorithm, we shall try to use it to calculate the AC along with the Wiener-Khintchine theorem.

First, we know that the inverse of the WHT matrix is the very same matrix with all its coefficients divided by 2^n; that is:

$$H_n^{-1} = 2^{-n} H_n.$$

Also, notice how we derived the Butterfly Algorithm from the WHT matrix: it was only necessary to see that we could express the WHT in terms of the H_{n-1} matrix, and the sum and difference of the first half of the polarity TT with the second half. This also stands for the inverse WHT, since the inverse matrix of the WHT matrix is essentially the same. The only thing that we have to take care of is to divide by two in every step we make. Thus, we have:

Lemma 4 (Inverse Butterfly Algorithm) *Take the WHT $[W_f]$ of a Boolean function f, halve it into h_1 and h_2 and consider the lists $(h_1 + h_2)/2$ and $(h_1 - h_2)/2$. Now, for each list, repeat the process. The result, after n iterations, is the polarity TT of f.*

So, in other to calculate the AC, and according to the Wiener-Khintchine theorem, it is sufficient to take the WHT spectrum, square it pointwise, and apply the inverse butterfly algorithm.

9.3.5 Differential uniformity

We now examine how to implement the evaluation of the differential uniformity of an S-box. Remember that what we want is a differential distribution table as flat as possible, and therefore, a differential parameter as close to zero as possible.

Matlab Implementation

In order to calculate the differential distribution table in a matrix context we utilize that the DDT are described with the following relationship:

$$[AC_S] = [DDT].[H_m],$$

the procedure for producing the Walsh-Hadamard matrix H_m is already described above and then all that is left is to solve the linear system and calculate the differential uniformity parameter.

C++ Implementation

For the C++ implementation, we are going to use some *algebraic technology* to avoid, one more time, matrix multiplication, making the program more efficient, at least, in terms of memory usage. We base our method on the explanations given in [5]. Our aim will be to encode S-boxes in the form of polynomials over a ring, that will allow us to calculate the differential distributions table just by computing polynomial products. Moreover, in this ring, we will find a way to multiply polynomials that is more efficient that the classical convolution product.

Lets talk first about this multiplication trick, before dealing with S-boxes. The ring we are considering is

$$R = \mathbb{Z}[x_1, ..., x_n] / \left(x_1^2 - 1, ..., x_n^2 - 1\right)$$

Let $p, q \in R$ be

$$p(x_1, ..., x_n) = \sum_{i=0}^{2^n-1} a_i M(i)$$

$$q(x_1, ..., x_n) = \sum_{i=0}^{2^n-1} b_i M(i)$$

where

$$M(i) = \prod_{j=1}^{n} x_j^{B_j(i)}$$

taking $B_j(i)$ as the $j-$th bit of the binary representation of i. Notice that this polynomials might represent an encoding of some discrete functions $f, g : \mathbb{F}_2^n \longrightarrow \mathbb{R}$ such that $f(i) = a_i$ and $g(i) = b_i$[2]. Also, the polynomial $p\cdot q$ represents an encoding of the discrete convolution function $f * g$. Since we are now speaking about convolution, we inspire ourselves in the following property, valid for the Fourier coefficients of two real functions of L^2:

$$\widehat{(f * g)}(n) = \widehat{f}(n)\widehat{g}(n)$$

Can we find a similar property for our context? Let us try first for the case where $n = 1$. Here,

$$p(x_1) = a_0 + a_1 x_1$$

$$q(x_1) = b_0 + a_1 x_1$$

then

$$(p\cdot q)(x_1) = (a_0 b_0 + a_1 b_1) + (a_0 b_1 + a_1 b_0) x_1$$

Let $[p] = (a_0, a_1)$ and $[q] = (b_0, b_1)$ be two lists that represent the values of the functions f and g encoded by our polynomials p and q, respectively. Then, we may write $[pq] = (a_0 b_0 + a_1 b_1, a_0 b_1 + a_1 b_0)$. Now, notice that, calculating the WHT of f and g may be done by obtaining lists $[W_p]$ and $[W_q]$ that are calculated by applying the butterfly algorithm to the lists $[p]$ and $[q]$, so that

$$[W_p] = (a_0 + a_1, a_0 - a_1)$$

$$[W_q] = (b_0 + b_1, b_0 - b_1)$$

and also, we have that

$$\begin{aligned} W_{pq} &= (a_0 b_0 + a_1 b_1 + a_0 b_1 + a_1 b_0, a_0 b_0 + a_1 b_1 - a_0 b_1 - a_1 b_0) \\ &= ((a_0 + a_1)(b_0 b_1), (a_0 - a_1)(b_0 - b_1)) \end{aligned}$$

so, we just found that

$$W_{pq} = W_p * W_q$$

where $*$ denotes here the point wise product. It is easy to see by induction that this is also true for any value of n, and thus, we have found our analogous

[2]We understand, when i is passed as an argument, the binary representation of i, which is an element of $\mathbb{F}_2^n$

property, because the WHT is just a kind of discrete Fourier transformation.

Notice that this equation suits our original goal, which was to find a fast multiplication method in this ring. Certainly, it is needed to calculate, given the lists of coefficients of the two polynomials, their WHT, then multiply both lists, and undo the WHT. Using the butterfly algorithm, this calculation is also performed in $O(n2^n)$ operations, which is, of course, faster than the $O(2^{2n})$ operations that you would need if the product of the two polynomials would be calculated directly.

Now is time to try to take profit from this property. First, let S be a S-box of n bits input and m bits output. Our first problem is that, in our previous discussion, we allowed polynomials to encode discrete functions from $\mathbb{F}_2^n$ to $\mathbb{R}$, so it seems that all we can do without really changing too many things is dealyng, not with general S-boxes, but only with boolean functions. Nevertheless, we can fix that quite easily, by considering the *characteristic function of S*, which we denote as χ_S, and it is defined as follows:

$$\begin{aligned}\chi_S : & \mathbb{F}_2^{n+m} \longrightarrow \mathbb{F}_2 \\ & (j|i) \longmapsto \begin{cases} 1 \text{ if } j = S(i) \\ 0 \text{ else} \end{cases}\end{aligned}$$

Here, $(j|i)$ represents a binary value of $m+n$ digits that is formed by a binary value j of length m followed by another value i of length n. Notice that this function takes the value 1 exactly 2^n times.

Let $\pi(x_1, ..., x_{m+n}) \in R$ be the polynomial that encodes the boolean function χ_S:

$$\pi(x_1, ..., x_{m+n}) = \sum_{k=0}^{2^{n+m}-1} \chi_S(k) M(k)$$

Since the function χ_S only values 1 exactly 2^n times, and all the other values al zero, this polynomial will only have 2^n monomials in variables $x_1, ..., x_{n+m}$. Also, where does $\chi_S(k) = 1$? By definition, when $k = (S(i)|i)$, taking $i \in \mathbb{F}_2^n$, or, in decimal representation, when $k = 2^n S(i) + i$, so that

$$\pi(x_1, ..., x_{m+n}) = \sum_{i=0}^{2^n-1} M(2^n S(i) + i)$$

And here comes what we wanted: a simple calculation can show that the coefficients of the polynomial π^2 are exactly the values of the differential distributions table of the S-box S, and since we have a method for multiplying polynomials fast in this ring, we get the following algorithm to calculate the differential characteristics:

Lemma 5 *Let S be an S-box, and consider the list $[S] = (a_1, ..., a_{2^{m+n}})$, where $a_j = 1$ if $j = 2^n S(i) + i$, and 0 in the other case. Then, the result of applying the butterfly algorithm to this list, squaring it point wise, and applying the inverse butterfly algorithm, is the differential distribution table of the S-box.*

9.4 Comparison of two implementations

n, m	C++	Matlab
3	0.0002 s	0.002 s
4	0.001 s	0.002 s
8	0.16 s	0.1 s
16	1 h 41 min	Out of memory

Table 9.1: Time required for the two implemented algorithms

We have tested our implementations with 4 example S-boxes, each of one of size 3×3, 4×4, 8×8 and 16×16. In table 9.1 we can see the time results for each implementation. For a very low n, the C++ implementation is faster which is natural considering Matlab's much larger overhead and for both $n = 4$ and $n = 8$ we don't see any big discrepancies. It is not until we reach $n = 16$ we encounter problems. We then have matrices, for the Matlab implementation, that are of the size $2^{16} \times 2^{16}$ and larger, which is more memory demanding than most Matlab installations can handle. Therefore, the program does not return any solution.

The C++ implementation doesn't have this memory problem, since everything have been implemented avoiding matrices and trying to store as less values as possible, and the program is able to perform the calculations. Nevertheless, we see that the time of computation required for dealing with all the $2^{16} - 1$ non-zero linear combinations of the Boolean functions of the S-box is outrageously big for any real application in sequential S-box generation and analysis.

So, we conclude the following. On one hand, it is clear that the C++ implementation is more efficient in terms of memory and operations required than the Matlab implementation. We have seen this through all the previous sections, and it is demonstrated in practice when reaching the limit of big values of n and m. Although it is still too slow for practical uses, it is more opened that the Matlab implementation to adjustments that could improve its performance, because it does not work with algorithms as closed as the matrix multiplication algorithms implemented in Matlab.

On the other hand, there may still be reasons for the using Matlab though. Firstly, the implementation is much simpler and more transparent, which may be an advantage when making modifications or extending the toolbox. Secondly, there is not practical difference between being able to calculate the S-box parameters in almost two hours and not being able to do it, so, for the moment, and from a pragmatic point of view, we might say that this implementation perform as well as the C++ implementation. Also, the Matlab implementation may fit better into other tools used by researchers.

9.5 Conclusion and future work

We have implemented two versions of a toolbox that can be used for studying the properties of Boolean functions or S-boxes. The Matlab implementation is simpler, but if larger functions is to be considered the C++ should be used. Still, this implementation also have its inconveniences, because it implements a little bit more sophisticated algorithms to gain performance and safe memory, but it is not fast enough to deal properly with a huge amount of large S-boxes.

We have worked on keeping the implementations open to modification when new properties are suggested. It might be possible, for example, to find bounds for the different parameters calculated after a deeper research of the algebraic properties of the S-boxes, which could help, in some situations, to avoid unnecessary calculations. It might be also interesting to study the possibility of adapting this implementations for parallel computing, so that given an S-box, several linear combinations of its Boolean functions could be evaluated at the same time.

Bibliography

[1] Beauchamp, K.G., Applications of Walsh and related functions: With an introduction to sequence functions (Microelectronics and Signal Processing), Academic Press, London New York, 1984.

[2] Biham, E., Shamir, A., Differential Cryptanalysis of DES-like Cryptosystems,In:Menezes,A., Vanstone,S.A.(eds.) Advances in Cryptology-CRYPTO 1990, LNCS, 537, Springer, Heidelberg, 1991, 2-21.

[3] Braeken, A., Cryptographic properties of Boolean functions and S-boxes, PhD Thesis, KU Leuven, Belgum, 2006.

[4] Jacobson, N., Basic algebra I, Freeman, W.H. and Company, 1985.

[5] Joux, A., Algorithmic Cryptanalysis, Chapman and Hall/CRC Press, 2009.

[6] Lafitte, F.,The Boolfun Package: Cryptographic Properties of Boolean Functions, 2013.

[7] Matsui, M., Linear cryptanalysis method for DES cipher, Advances in Cryptology - EUROCRYPT 1993.

[8] Millan, W., Low order approximation of cipher functions, In: Dawson, Ed., Goli, J. (eds.) Cryptography: Policy and Algorithms: International Conference Brisbane, Queensland, Australia, July 3–5, 1995, Springer, Heidelberg, 1996, 144–155.

[9] Shannon, C., Communication Theory of Secrecy Systems, Bell System Technical Journal, 28(4), 1949, 656-715.

[10] Stein, W.A., et al., Sage Mathematics Software 2013, http://www.sagemath.org.

[11] Picek, S., Batina, L., Jakobovi, D., Ege, B., Golub, M., S-box, SET, Match: A Toolbox for S-box Analysis, In: Information Security Theory and Practice. Securing the Internet of Things, 8501, Lecture Notes in Computer Science, Springer, Heidelberg, 2014, 140–149.

[12] Preneel, B., Govaerts, R., Vandewalle, J., Boolean functions Satisfying Higher Order Propagation Criteria, In: Advances in Cryptology - Eurocrypt91, Proceedings, 547, Springer-Verlag, 1991, 141–152.

[13] Rothaus, O.S., On bent functions, Journal of Combinatorial Theory A, 20, 1976, 300–305.

[14] Wang,Y., Xie, Q., Wu, Y., Du, B., A Software for S-box Performance Analysis and Test, Electronic Commerce and Business Intelligence, Beijing, 2009, 125–128.

[15] Weinstein, C. J., Quantization Effects on Digital Filters, MIT Lincoln Laboratory Technical Report 468, 1969, 42.

[16] Zheng, Y., Zhang, X.-M., GAC - the criterion for Global Avalanche Characteristics of cryptographic functions, Journal for Universal Computer Science, 1 (5), 1995, 316–333.