COMPUTING CRYPTOGRAPHIC PROPERTIES OF BOOLEAN FUNCTIONS FROM THE ALGEBRAIC NORMAL FORM REPRESENTATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÇAĞDAŞ ÇALIK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

FEBRUARY 2013

Approval of the thesis:

COMPUTING CRYPTOGRAPHIC PROPERTIES OF BOOLEAN FUNCTIONS FROM THE ALGEBRAIC NORMAL FORM REPRESENTATION

submitted by **ÇAĞDAŞ ÇALIK** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Department of Cryptography, Middle East Technical University** by,

Prof. Dr. Bülent Karasözen
Director, Graduate School of **Applied Mathematics** _____

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** _____

Assoc. Prof. Dr. Ali Doğanaksoy
Supervisor, **Department of Mathematics, METU** _____

Examining Committee Members:

Prof. Dr. Ersan Akyıldız
Department of Mathematics, METU _____

Assoc. Prof. Dr. Ali Doğanaksoy
Department of Mathematics, METU _____

Prof. Dr. Ferruh Özbudak
Department of Mathematics, METU _____

Asst. Prof. Dr. Ali Aydın Selçuk
Department of Computer Engineering, Bilkent University _____

Dr. Orhun Kara
TÜBİTAK BİLGEM _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ÇAĞDAŞ ÇALIK

Signature :

ABSTRACT

COMPUTING CRYPTOGRAPHIC PROPERTIES OF BOOLEAN FUNCTIONS FROM THE ALGEBRAIC NORMAL FORM REPRESENTATION

Çalık, Çağdaş

Ph.D., Department of Cryptography

Supervisor : Assoc. Prof. Dr. Ali Doğanaksoy

February 2013, 54 pages

Boolean functions play an important role in the design and analysis of symmetric-key cryptosystems, as well as having applications in other fields such as coding theory. Boolean functions acting on large number of inputs introduces the problem of computing the cryptographic properties. Traditional methods of computing these properties involve transformations which require computation and memory resources exponential in the number of input variables. When the number of inputs is large, Boolean functions are usually defined by the algebraic normal form (ANF) representation. In this thesis, methods for computing the weight and nonlinearity of Boolean functions from the ANF representation are investigated. The relation between the ANF coefficients and the weight of a Boolean function was introduced by Carlet and Guillot. This expression allows the weight to be computed in $O(2^p)$ operations for a Boolean function containing p monomials in its ANF. In this work, a more efficient algorithm for computing the weight is proposed, which eliminates the unnecessary calculations in the weight expression. By generalizing the weight expression, a formulation of the distances to the set of linear functions is obtained. Using this formulation, the problem of computing the nonlinearity of a Boolean function from its ANF is reduced to an associated binary integer programming problem. This approach allows the computation of nonlinearity for Boolean functions with high number of input variables and consisting of small number of monomials in a reasonable time.

Keywords : Boolean functions, algebraic normal form, weight, nonlinearity

ÖZ

BOOLE FONKSİYONLARININ KRİPTOGRAFİK ÖZELLİKLERİNİN CEBİRSEL NORMAL BİÇİM GÖSTERİMİNDEN HESAPLANMASI

Çalık, Çağdaş

Doktora, Kriptografi Bölümü

Tez Yöneticisi : Doç. Dr. Ali Doğanaksoy

Şubat 2013, 54 sayfa

Boole fonksiyonları simetrik anahtarlı kriptosistemlerin tasarım ve analizinde önemli rol oynamanın yanı sıra kodlama teorisi gibi alanlarda uygulamaları olan bir araştırma alanıdır. Girdi sayısı fazla olan Boole fonksiyonları, kriptografik özelliklerin hesaplanması problemini beraberinde getirir. Bu özellikleri hesaplamanın geleneksel yolu, hesaplama ve hafıza kaynakları girdi sayısına üstel olarak bağlı olan dönüşümler gerektirir. Yüksek girdi sayılı Boole fonksiyonları genellikle cebirsel normal biçim (ANF) gösterimi ile ifade edilir. Bu tezde, ANF gösterimi verilen bir Boole fonksiyonunun ağırlığını ve doğrusallıktan sapma miktarını hesaplayan yöntemler araştırılmıştır. Bir Boole fonksiyonunun ANF katsayıları ve ağırlığı arasındaki ilişki Carlet ve Guillot tarafından gösterilmiştir. Bu ifade, ANF gösteriminde p adet terim olan bir Boole fonksiyonunun ağırlığının $O(2^p)$ işlemde hesaplanabilmesine olanak sağlamıştır. Bu çalışmada, ağırlık ifadesindeki gereksiz işlemlerden kaçınan daha verimli bir algoritma önerilmiştir. Ağırlık ifadesi genelleştirilerek, doğrusal fonksiyonlara uzaklığın bir formülü elde edilmiştir. Bu formül sayesinde bir Boole fonksiyonunun doğrusallıktan sapma miktarını ANF gösteriminden bulma problemi, ilgili bir ikili tamsayı programlama problemine indirgenmiştir. Bu yaklaşımla, yüksek girdi sayılı ve az sayıda terim içeren Boole fonksiyonlarının doğrusallıktan sapma miktarı makul sürelerde hesaplanabilmektedir.

Anahtar Kelimeler : Boole fonksiyonları, cebirsel normal biçim, ağırlık, doğrusallıktan sapma miktarı

ACKNOWLEDGMENTS

I want to thank my supervisor Assoc. Prof. Dr. Ali Doğanaksoy for his guidance during the preparation of this thesis. I also want to express my gratitude to the thesis committee, academic and administrative staff of the Institute of Applied Mathematics, and all the friends.

I am grateful to my wife and my family for their support.

The work in this thesis is partially supported by TÜBİTAK under project no. 109T672.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Theory of Boolean functions is an important research area having applications in various fields such as cryptology, coding theory, digital circuit theory, to name a few. As noted in [10], *"'..., Boolean functions are the simplest interesting multivariate functions."'*, a consequence of the fact that they assign 0 or 1 to each input. Despite the conceptual simplicity, what makes Boolean functions research challenging is how quickly the number of functions grow as the number of inputs increase, rendering it impossible to make an exhaustive search on the entire space. With today's computation power, it is not possible to conduct an exhaustive search on Boolean functions of 6 variables or more . This limitation forces the researchers to devise intelligent methods with as little computation resources as possible. A great amount of research in Boolean functions has been devoted to proving the existence/non-existence of Boolean functions with specified properties, and to construct or enumerate Boolean functions satisfying desired criteria. The existing generic algorithms can be used efficiently for an arbitrary Boolean function with a small number of inputs, but they become infeasible as the number of inputs increase, due to the exponential computational complexity required by these algorithms. A relatively less studied aspect is computing the cryptographic properties of Boolean functions acting on large number of input variables (say $n > 40$).

A function that maps n-bits of input to m-bits of output can be considered as a collection of m Boolean functions. These $n \times m$ mappings are called vectorial Boolean functions. A symmetric-key cryptosystem can be thought of as a vectorial Boolean function which involves a large number of variables ranging from 64-bits to 512-bits. Such functions are supposed to attain certain desired properties of confusion and diffusion stated by Shannon [35], but when the number of input variables is high, it is not even possible to express them in the form of available representations. Instead, the research is focused on the primitive structures that are fundamental building blocks of these systems. The aim in analyzing these building blocks is to evaluate the cryptographic strength of the system and in turn try to come up with a claimed security bound, which determines a particular cryptanalytic attack's complexity to break the system.

Most common type of building blocks used in cryptosystems are S-boxes (substitution boxes). These components provide the nonlinearity that is needed for Shannon's confusion principle, i.e., to guarantee a complex relation between the input and output bits of the system. S-boxes are a major component of most of the block ciphers and hash functions, c.f. the S-box of the block cipher standard AES [13]. Because of its relatively high hardware cost, S-boxes were not a preferred component for stream ciphers in the past. However, with the advances

in technology, there are now several stream ciphers incorporating S-boxes in their designs [1, 7, 21, 24]. Stream ciphers also utilize Boolean functions as feedback functions of feedback shift registers, or filtering and combining functions to generate output from these systems.

Whatever type the building block is, there are various criteria with which a Boolean function's strength is assessed. First and foremost, in order not to be statistically distinguishable, most of the Boolean functions used in cryptosystems must be balanced, i.e., the function's output contains an equal number of zeros and ones. In order to resist against algebraic attacks, a Boolean function should have a high algebraic degree and high algebraic immunity [4, 9, 29]. High orders of correlation immunity and propagation characteristics are also desired properties [36, 33, 32]. A must have property is high nonlinearity [30], the quantity of being far away from the set of affine functions. Lack of high nonlinearity allows linear approximations of the system, and this has led to one of the earliest and powerful type of cryptanalysis on modern block ciphers [27]. Because of its importance, construction of Boolean functions having high nonlinearity and at the same time having other desired properties is a widely studied topic. While the highest achievable nonlinearity for Boolean functions with odd number of inputs is not known in general, for even number of variables the functions reaching this limit are known; they are introduced by Rothaus and called bent functions [34]. Several methods for constructing bent functions exist [28, 14, 16], yet an explicit enumeration is not known. The number of bent functions are currently known up to 8 variables [31, 17, 25]. A drawback of bent functions is their unbalancedness. On the other hand, they are good candidates to be used as a starting point to construct highly nonlinear balanced Boolean functions.

There is no ideal Boolean function possessing all the previously mentioned properties. It is shown that some of these properties are in conflict with each other, and one has to make a trade-off when choosing Boolean functions for practical use, such as correlation immunity and the algebraic degree of a function. Another example is nonlinearity and balancedness.

In coding theory, Boolean functions have strong connections with Reed-Muller codes. These codes correspond to the set of Boolean functions with algebraic degrees less than or equal to a value that is a parameter of the code. Decoding problem in r^{th} order Reed-Muller codes corresponds finding the r^{th} order nonlinearity of Boolean functions. The concept of nonlinearity is also related to the covering radius of codes.

Boolean functions can be represented in various forms; truth table, algebraic normal form (ANF), numerical normal form, Walsh spectrum and trace representations are the common ones. Truth table is a complete listing of the function's outputs. Algebraic normal form is a multivariate polynomial of input variables, that is, the function output is expressed as the sum of terms, which are products of input variables. One advantage of ANF over the truth table representation in some cases is that it allows to specify the function in a more compact way, requiring less amount of memory. Some construction methods may produce functions based on the ANF. Numerical normal form is obtained by converting the field arithmetic in the ANF to integer arithmetic, hence, an expression of the Boolean functions over the ring of integers is considered [5]. This representation has led to the characterization of functions with high nonlinearity and resiliency [19, 6, 3]. Walsh spectrum is a list of coefficients, which specify the correlation of the function with all linear functions. Most of the cryptographic properties of Boolean functions can be calculated directly from the Walsh spectrum, e.g., weight, nonlinearity, correlation immunity. Therefore, a natural approach to construct Boolean functions with

desired properties is by specifying their Walsh spectrum. However, an arbitrarily generated sequence of 2^n integers need not be the Walsh spectrum of a Boolean function.

In this thesis, Boolean functions are treated as cryptographical objects. Efficient methods for computing the weight and nonlinearity of Boolean functions from their ANF representation are investigated. What is meant by being efficient is that the computation of the properties in consideration for a special set of Boolean functions that would otherwise be infeasible with the traditional methods of computing these quantities. More explicitly, these are the functions acting on high number of input variables and having low number of monomials in their ANF representation. The main contributions of this study are two algorithms, one for computing the weight, and the other for computing the nonlinearity of a Boolean function from its ANF, without constructing its truth table or Walsh spectrum. These algorithms may be useful especially when the number of input variables is so large that the best known algorithms for computing these quantities (Fast Möbius transform and Fast Walsh transform) become impractical, due to their exponential computational complexity of $O(n2^n)$ operations. An example cryptosystem utilizing these types of Boolean functions is stream cipher Grain and its updated version Grain-128, making use of nonlinear feedback functions of length 80- and 128-bits, respectively [22, 23].

The outline of the thesis is as follows: In Chapter 2, formal definitions and necessary notation regarding Boolean functions are given. Chapter 3 is devoted to the algorithm that computes the weight of a Boolean function from its ANF. In Chapter 4, the idea used in Chapter 3 is generalized to express the distance of a Boolean function to a linear function in terms of its ANF coefficients, and it is explained how this might be used to construct an algorithm for computing the nonlinearity. Chapter 5 gives the conclusion.

CHAPTER 2

BOOLEAN FUNCTIONS

2.1 Introduction

This chapther is a brief summary of Boolean functions restricted to the scope of the thesis. Necessary definitions will be given and the notation that will be used throught the thesis is going to be settled. Representation methods such as *truth table*, *algebraic normal form*, *Walsh spectrum* and *numerical normal form* will be explained. Properties of Boolean functions related to cryptography such as balancedness, nonlinearity, correlation immunity, etc. will be defined. A detailed discussion on these topics and more can be found in [2, 11, 26].

2.2 Preliminaries

Let $\mathbb{F}_2 = \{0, 1\}$ be the finite field with two elements and $\mathbb{F}_2^n$ be the n-dimensional vector space over $\mathbb{F}_2$. Addition and multiplication operators in $\mathbb{F}_2$ are denoted by '$\oplus$' and '.', corresponding to usual addition and multiplication modulo 2. When there is no ambiguity, '.' symbol for multiplication may be omitted and the operation is shown by juxtaposition. Among the set of 16 possible binary operators that can be defined, Table 2.1 lists the common ones that will be used. Field addition and multiplication are also referred to as binary `xor` and `and` operators, respectively. `and` operator is also denoted by $\wedge$. Another binary operator that is worth mentioning is `or` operator denoted by $\vee$, outputs 1 when at least one of its inputs is 1. The unary operator `not` stands for the complement of a bit and is shown by $\neg$. Table 2.1 lists the outputs of these operators. These operators also apply to the elements of the vector space $\mathbb{F}_2^n$ where the operations are carried out componentwise.

Table 2.1: Common operators in $\mathbb{F}_2$.

x	y	$x \oplus y$	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	0	1	1	0

Inner product of vectors is a mapping $\mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2$ defined as $\langle x, y \rangle = x_1 y_1 \oplus \cdots \oplus x_n y_n$. Support of a vector $x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ is $supp(x) = \{i \mid x_i \neq 0\}$, which is the index set of non-zero

components. Indices may start from 0 or 1, which will be clear from the context. Hamming weight $wt(x)$ of a vector is the size of its support, i.e., $wt(x) = |supp(x)|$.

Vectors in $\mathbb{F}_2^n$ can be identified with integers modulo 2^n by associating $\overline{x} = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ with the integer $x = \sum_{i=1}^{n} x_i 2^{n-i}$. This mapping between integers and vectors both helps us to define an ordering on $\mathbb{F}_2^n$ called the lexicographic ordering, and provides a simpler notation by using the corresponding integers for vectors, such as $\overline{0}$ denoting the vector consisting of all zeros.

Another ordering defined on vectors is partial ordering. For $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ in $\mathbb{F}_2^n$, $x \preceq y$ if $x_i \leq y_i$ for $1 \leq i \leq n$, and in this case x is said to be a *sub-vector* of y. In this case, y is called a *super-vector* of x (or y covers x), denoted by $y \succeq x$. In other words, $x \preceq y$ if and only if $supp(x) \subseteq supp(y)$. The set of all sub-vectors of a vector $x \in \mathbb{F}_2^n$ is called the sub-space of x, denoted by $\underline{S}(x)$, and the set of all super-vectors of a vector $x \in \mathbb{F}_2^n$ is called the super-space of x, denoted by $\overline{S}(x)$.

$$\underline{S}(x) = \{y \in \mathbb{F}_2^n \mid y \preceq x\} \tag{2.1}$$
$$\overline{S}(x) = \{y \in \mathbb{F}_2^n \mid x \preceq y\} \tag{2.2}$$

Definition of a sub-vector implies that all sub-vectors of a vector $x = (x_1, \ldots, x_n)$ have the value of 0 at indices where $x_i = 0$, and can take on any value at the remaining positions. Similarly, all super-vectors of x have the value of 1 where $x_i = 1$, and the remaing positions can be set freely. These observations lead us to the following results about the cardinality of these sets:

$$|\underline{S}(x)| = 2^{wt(x)} \tag{2.3}$$
$$|\overline{S}(x)| = 2^{n-wt(x)} \tag{2.4}$$

The following two relations can be obtained from the definitions of super-vectors and sub-vectors:

$$\overline{S}(x) \cap \underline{S}(x) = \{x\} \tag{2.5}$$
$$\overline{S}(\overline{0}) = \underline{S}(\overline{1}) = \mathbb{F}_2^n \tag{2.6}$$

2.3 Truth Table and Algebraic Normal Form

An n-variable Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ specifies a mapping from the n-dimensional vector space to $\mathbb{F}_2$. $\mathcal{B}_n$ will denote the set of n-variable Boolean functions. Since there are two output choices for each n-bit input vector, total number of n-variable Boolean functions is $|\mathcal{B}_n| = 2^{2^n}$. A common way of representing a Boolean function is by supplying a list of output values for each n-bit input vector, called the *truth table* of the function. Formally, this is shown by $T_f = (f(\overline{0}), f(\overline{1}), \ldots, f(\overline{2^n - 1}))$. The *support* (resp. *weight*) of $f \in \mathcal{B}_n$ is the support (resp. weight) of its truth table, that is, the weight of the function is $wt(f) = |supp(f)| = |\{x \in \mathbb{F}_2^n \mid f(x) = 1\}|$. f is called *balanced* if its truth table contains an equal number of zeros

and ones, and called *unbalanced* otherwise. Namely, if f is a balanced Boolean function then $wt(f) = 2^{n-1}$. Note that the representation of an n-variable Boolean function by its truth table requires $O(2^n)$ units of memory, which may be inefficient for some applications, or may not be feasible at all. Addition (resp. product) of two Boolean functions is defined as the sum (resp. product) of their truth tables. Table 2.2 shows an example of addition and product of two 3-variable Boolean functions. The Hamming distance between two Boolean functions is defined as the number of truth table entries they differ by, that is, $d(f, g) = |\{x \in \mathbb{F}_2^n \mid f(x) \neq g(x)\}|$. The result of a binary addition operation indicates whether the input operands are the same or not, hence the distance between two functions corresponds to $d(f, g) = wt(f \oplus g)$. This quantity also specifies the number of changes that needs to be done to convert the truth table of a function to the other's.

Table 2.2: Truth tables of two 3-variable Boolean functions and their sum and product.

x_1	x_2	x_3	T_f	T_g	$T_{f\oplus g}$	$T_{f\cdot g}$
0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
1	0	0	0	0	0	0
1	0	1	1	1	0	0
1	1	0	0	1	1	1
1	1	1	1	0	1	0

A Boolean function can also be represented as a multivariate polynomial called the *algebraic normal form (ANF)* such that

$$f(x_1, \ldots, x_n) = \bigoplus_{u \in \mathbb{F}_2^n} a_u x^u \tag{2.7}$$

where $x^u = x_1^{u_1} x_2^{u_2} \ldots x_n^{u_n}$ is a *monomial* composed of the variables for which $u_i = 1$ and $a_u \in \mathbb{F}_2$ is called the ANF coefficient of x^u. Degree of the monomial x^u is $deg(x^u) = wt(u)$, corresponding to the number of variables appearing in the product. The highest degree monomial with the non-zero ANF coefficient is defined to be the degree of the function. The ANF coefficient a_u for $u \in \mathbb{F}_2^n$ will sometimes be used with an integer subscript a_i with $i \in \{0, \cdots, 2^n - 1\}$ where $\bar{i} = u$. For example, all Boolean functions on 3-variables can be represented in the following form:

$$f(x_1, x_2, x_3) = a_0 \oplus a_4 x_1 \oplus a_2 x_2 \oplus a_1 x_3 \oplus a_6 x_1 x_2 \oplus a_5 x_1 x_3 \oplus a_3 x_2 x_3 \oplus a_7 x_1 x_2 x_3,$$

with $a_i \in \mathbb{F}_2$ being the ANF coefficients. Note that the numbering of input variables in this representation is from left to right, which means that x_1 is associated with the most significant bit of a vector. For a general dimension n, the ANF coefficient of x_1 will be $a_{2^{n-1}}$ and the ANF coefficient of x_n will be a_1.

Considering the vectors T_f and A_f of length 2^n for truth table and ANF representations, each vector specifies a unique Boolean function. Since there are both 2^n truth table entries and ANF coefficients, there is a $1-1$ mapping between these two representations. Otherwise,

there should be more than one ANF representation for some of the Boolean functions and no corresponding ANF representation for some of the Boolean functions, which is not the case.

Another ANF representation by specifying the variables involved in a monomial with the set notation is as follows:

$$f(x) = \bigoplus_{I \in \{1,\cdots,n\}} a_I x^I \tag{2.8}$$

Here, $a_I \in \mathbb{F}_2$ are the ANF coefficients and $x^I = \prod_{i \in I} x_i$ are the corresponding monomials. The set notation has useful features when relations between the monomials are considered. The following definitions regarding the set notation of monomials will be necessary for the next chapters.

Definition 2.3.1 *The product of two monomials is defined as $x^I \cup x^J = x^{I \cup J}$, which is a monomial composed of input variables appearing in either x^I or x^J or both.*

Definition 2.3.2 *The difference of a monomial with respect to another monomial is defined as $x^I \setminus x^J = x^{I \setminus J}$, which is a monomial composed of input variables appearing only in x^I.*

Example 2.3.3 *Let $I = \{1, 2\}$ and $J = \{2, 3\}$ and the corresponding monomials are $x^I = x_1 x_2$ and $x^J = x_2 x_3$. The product of these monomials is $x^I \cup x^J = x_1 x_2 x_3$. The difference of there monomials with respect to each other are $x^I \setminus x^J = x_1$ and $x^J \setminus x^I = x_3$.*

Definition 2.3.4 *A monomial x^I covers x^J if $J \subseteq I$, i.e., all variables appearing in x^J also appear in x^I. In this case, x^J is called a sub-monomial of x^I, denoted by $x^J \subseteq x^I$.*

Proposition 2.3.5 *For two monomials x^I and x^J, the following are always satisfied.*

$$x^I \cup x^J = (x^I \setminus x^J) \cup x^J, \tag{2.9}$$

$$x^I \cup x^J = x^I, \quad \text{if } x^I \text{ covers } x^J. \tag{2.10}$$

Proof. Combining Definitions 2.3.1, 2.3.2 and 2.3.4, the results follow from basic set operations. ■

Regardless of which of the above ANF notations is used, ANF representation tells us that a Boolean function can be represented as a sum of monomials which are products of its input variables. Although there are 2^n ANF coefficients and a Boolean function can be specified with a vector $A_f = (a_0, \cdots, a_{2^n-1})$ that lists all of its ANF coefficients, it is more efficient merely to specify the monomials with non-zero ANF coefficients when $supp(A_f)$ is relatively small. This is also necessary when n is so large that a complete list of the ANF coefficients is impossible to define.

The value of $f \in B_n$ at a particular point $x \in \mathbb{F}_2^n$ can be obtained from the ANF coefficients as follows:

$$f(x) = \bigoplus_{u \preceq x} a_u. \tag{2.11}$$

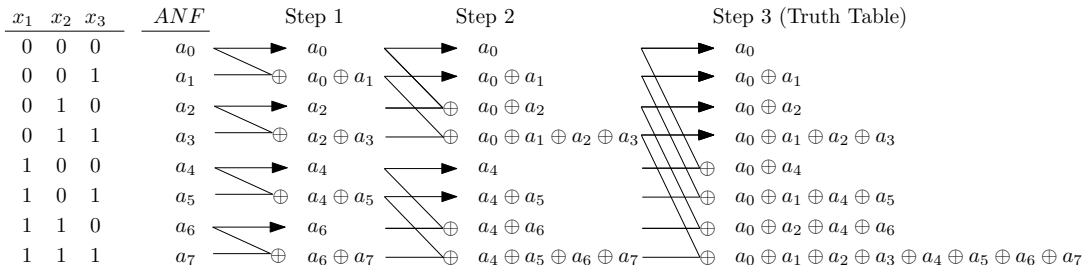

Figure 2.1: Fast Möbius transform on 3-variable Boolean functions.

That is to say, only the monomials with coefficients a_u satisfying $u \preceq x$ contribute to the value of f at point x, because for $y = (y_1, \dots y_n) \in \mathbb{F}_2^n$, the product $x^y = x^{y_1} \dots x^{y_n}$ will attain the value zero for the input x when $supp(y) \setminus supp(x) \neq \emptyset$.

One way to construct the truth table of a Boolean function from its ANF representation would be to use Equation 2.11 and calculate each truth table entry one by one. However, there is a more efficient method called the *Fast Möbius transform* which transforms the table of ANF coefficients to the truth table. For an n-variable Boolean function, the algorithm consists of n steps and at the i^{th} step, the table of length 2^n is considered as 2^{n-i} consecutive blocks of length 2^i. Each block is processed in the following manner: first (upper) half of the bits in each block are copied as is to the next step and the second (lower) half of the bits are added to the first half of the block, producing the second (lower) half for the next iteration. After the processing of all blocks are completed, the block length is doubled and the same operation is repeated where the last block processed will be the single block of length 2^n. Since there are n steps and at each step all 2^n elements are processed, this algorithm has a computational complexity of $O(n2^n)$ operations. Figure 2.1 shows how *Fast Möbius transform* works for 3-variable Boolean functions. At the end of the n^{th} step, the table at hand is the truth table of the Boolean function. This transformation is an involution, meaning that, if one starts with the truth table of the function, the output will be the list of ANF coefficients.

2.4 Walsh Spectrum

Boolean functions of the form $l_{w,c} = \langle w, x \rangle \oplus c$, where $w \in \mathbb{F}_2^n$ and $c \in \mathbb{F}_2$ are called affine functions. The set of all affine functions are denoted by $\mathcal{A}_n$. The subset of affine functions when $c = 0$ are called linear functions. The set of all linear functions are denoted by $\mathcal{L}_n$. All affine functions are balanced except for $f(x) = 0$ and $f(x) = 1$.

Discrete Fourier transform of a Boolean function is defined as follows:

$$F_f(w) = \sum_{x \in \mathbb{F}_2^n} f(x)(-1)^{\langle w,x \rangle}. \tag{2.12}$$

When considering the distances between Boolean functions, it is more convenient to work on truth table values which are converted from $(0, 1)$ to $(1, -1)$. The sign function $\hat{f}$ of a Boolean function $f \in \mathcal{B}_n$ is defined as $\hat{f}(x) = (-1)^{f(x)} = 1 - 2f(x)$. The truth table $T_{\hat{f}} =$

$(\hat{f}(\overline{0}), \cdots, \hat{f}(\overline{2^n-1}))$ consisting of the values $(1,-1)$ is called the *polarity truth table* of f. An important tool for measuring the distance of a Boolean function to the set of affine functions is the *Walsh-Hadamard transform*, which is the Discrete Fourier transform of the sign function:

$$W_f(w) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus \langle w,x \rangle}. \tag{2.13}$$

$W_f(w)$ is caleed the Walsh coefficient of f at point w. The value of $W_f(w)$ indicates how much the truth table of f is correlated with the linear function l_w. $W_f(w)$ always takes on even values in the range $[-2^n, 2^n]$. The maximum value of $W_f(w) = 2^n$ is attained when the truth table of f is in full agreement with the linear function l_w and a value of -2^n is attained when f is the complement of the linear function l_w. The evenness of Walsh coefficients can be easily deduced from the fact that if two values are added from the set $\{-1, 1\}$, the sum will be in the set $\{-2, 0, 2\}$. So, an odd number can never be produced when the summation in Equation 2.13 is formed.

The ordered list of all Walsh coefficients $W_f = (W_f(\overline{0}), \ldots, W_f(\overline{2^n-1})$ is called the Walsh spectrum of f. Each Walsh coefficient is related to the distance between f and the linear function l_w in the following way:

$$W_f(w) = 2^n - 2d(f,g) \tag{2.14}$$

$$d(f,g) = 2^{n-1} - \frac{W_f(w)}{2} \tag{2.15}$$

Since $wt(f) = d(f,g)$, the Walsh coefficient at $w = (0, \ldots, 0)$ is related to the weight of the function. More specifically, $wt(f) = 2^{n-1} - \frac{W_f(\overline{0})}{2}$. Hence, whether a Boolean function is balanced or not can be determined by looking at the value of $W_f(\overline{0})$, which should be zero in the case of balancedness.

Polarity truth tables of linear functions can be formed by the so called Sylvester-Hadamard matrices. Sylvester-Hadamard matrix of order n is an $2^n \times 2^n$ matrix defined by the following formula:

$$H_0 = [1] \tag{2.16}$$

$$H_i = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes H_{i-1} \text{ for } i \geq 1 \tag{2.17}$$

where $\otimes$ refers to the Kronecker product of matrices. These matrices satisfy

$$H_n H_n = 2^n I_{2^n}$$

where I_{2^n} is the identity matrix of size $2^n \times 2^n$.

Each row (or column) of H_n corresponds to the polarity truth table of a linear function. Therefore, the Walsh spectrum of a Boolean function f can be obtained by multiplying the polarity truth table of f with the Sylvester-Hadamard matrix:

$$W_f = T_{\hat{f}} H_n \tag{2.18}$$

Below is the Sylvester-Hadamard matrix of order 3:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

A more efficient technique for computing the Walsh spectrum of a Boolean function is *Fast Walsh transform*. Fast Walsh transform is an n step algorithm that works similar to the Fast Möbius transform, except that the binary addition operation is replaced with integer addition and subtraction. At the i^{th} step, the table of entries are considered as blocks of length 2^i and for each block, the upper half of the block elements are added to the lower half of the block elements, producing the new upper half. The new lower half of blocks is obtained by subtracting the lower half of elements from the upper half elements. This process is applied until the single block of length 2^n is processed. The resulting table constitutes the Walsh spectrum of the Boolean function. This algorithm has a computational complexity of $O(n2^n)$ operations. Figure 2.2 shows the Fast Walsh transformation for 3-variable Boolean functions.

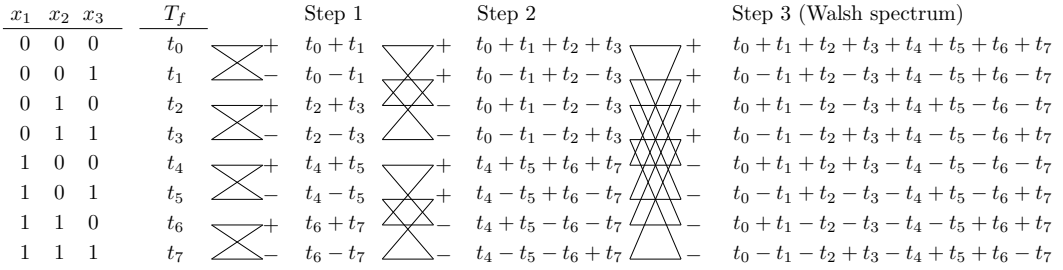

Figure 2.2: Fast Walsh transform on 3-variable Boolean functions.

Given the Walsh spectrum of a Boolean function, truth table can be obtained with the inverse Walsh transform:

$$\hat{f}(x) = \frac{1}{2^n} \sum_{w \in \mathbb{F}_2^n} W_f(w)(-1)^{\langle w,x \rangle}. \tag{2.19}$$

The inverse Walsh transform states that the value of a Boolean function at $x = (0, \ldots, 0)$ determines the sum of its Walsh coefficients:

$$2^n \hat{f}(\overline{0}) = \sum_{w \in \mathbb{F}_2^n} W_f(w) \tag{2.20}$$

In other words, the value of $f(0)$ being 0 or 1 results in the sum of Walsh coefficients being 2^n or -2^n, respectively.

Nonlinearity of a Boolean function f is the minimum distance between f and the set of affine functions:

$$N_f = \min_{g \in A_n} d(f, g) \tag{2.21}$$

Since each Walsh coefficient specifies the distance of a Boolean function to a particular linear function, nonlinearity can be calculated as a function of the Walsh spectrum:

$$N_f = 2^{n-1} - \frac{1}{2} \max_{w \in \mathbb{F}_2^n} |W_f(w)| \tag{2.22}$$

Remark 2.4.1 *Nonlinearity of Boolean functions is invariant under the transformation $g(x) = f(Ax \oplus b) \oplus l_w \oplus c$, where A is an $n \times n$ invertible binary matrix, $b, w \in \mathbb{F}_2^n$ and $c \in \mathbb{F}_2$ [11].*

Walsh spectrum satisfies the following relation called the Parseval's identity:

$$\sum_{x \in \mathbb{F}_2^n} W_f(w)^2 = 2^{2n} \tag{2.23}$$

Using Parseval's identity, one obtains that the maximum absolute value of Walsh coefficients can be minimized when all coefficients have the same magnitude, i.e.,

$$\sum_{w \in \mathbb{F}_2^n} W_f(w)^2 = 2^{2n} \tag{2.24}$$

$$2^n W_f(w)^2 = 2^{2n} \tag{2.25}$$

$$|W_f(w)| = \pm 2^{n/2} \tag{2.26}$$

The set of Boolean functions with this property are called *bent* functions. Bent functions achieve highest possible nonlinearity and only exist for even n.

A Boolean function is called correlation immune of order m if $W_f(w) = 0$ for all w with $1 \leq wt(w) \leq m$, and called called m-resillient if it is also balanced.

Walsh spectrum characterizes many cryptographic properties of Boolean functions. Therefore, an approach in constructing Boolean functions satisfying certain cryptographic properties is to construct a Walsh spectrum satisfying these properties. However, not every sequence of arbitrarily assigned Walsh coefficients correspond to a Walsh spectrum of a Boolean function. One way of testing whether a given Walsh spectrum is valid is to check whether the sum of the squares of its entries is equal to 2^{2n}. Also, the sum of the Walsh coefficients must be $\pm 2^n$. These two conditions are necessary but not sufficient, so if one wants to be certain, he/she must compute the value of the Boolean function by the inverse Walsh transform and see that this transformation yields 0 or 1 at all the points. There is another method appearing in [15], which states that

$$\sum_{w \in \mathbb{F}_2^n} W_f(w) W_f(w \oplus a) = 2^{2n} \delta(a), \forall a \in \mathbb{F}_2^n$$

must be satisfied where $\delta(a) = 1$ when $a = \overline{0}$ and 0 otherwise.

2.5 Numerical Normal Form

The last representation method for Boolean functions that will be discussed in this chapter is the *numerical normal form (NNF)*, introduced in [5]. This representation is obtained by carrying the arithmetic of ANF representation on $\mathbb{F}_2$ to the arithmetic over the integers. This allows us to represent a Boolean function in the following form:

$$f(x) = \sum_{u \in \mathbb{F}_2^n} \lambda_u x^u \tag{2.27}$$

where $x^u = \prod x_i^{u_i}$ is a monomial and $\lambda_u \in \mathbb{Z}$ is the NNF coefficient of the corresponding monomial. When NNF coefficients are taken modulo 2, they correspond to the ANF coefficients, that is:

$$a_u = \lambda_u \pmod 2.$$

Transformation from ANF to NNF can be accomplished by making use of the following conversion between binary and integer arithmetic:

$$a \oplus b = a + b - 2ab.$$

Example 2.5.1 *Let the ANF of a Boolean function be given as* $f(x_1, x_2, x_3) = x_1 \oplus x_3 \oplus x_2x_3$. *The NNF of this function can be computed as:*

$$\begin{aligned} f(x_1, x_2, x_3) &= (x_1 + x_3 - 2x_1x_3) \oplus x_2x_3 \\ &= x_1 + x_3 - 2x_1x_3 - x_2x_3 + 2x_1x_2x_3 \end{aligned}$$

Note in the example that the odd NNF coefficients are also the ANF coefficients of the function. In [5], an algorithm to compute the NNF coefficients from the truth table of a function is given. This algorithm is quite similar to the Fast Möbius transform. The only difference is that there is a subtraction operation instead of the binary addition operation. Figure 2.3 shows how the NNF coefficients are obtained from the truth table.

x_1	x_2	x_3	T_f	Step 1	Step 2	Step 3 (NNF)
0	0	0	t_0	t_0	t_0	t_0
0	0	1	t_1	$t_1 - t_0$	$t_1 - t_0$	$t_1 - t_0$
0	1	0	t_2	t_2	$t_2 - t_0$	$t_2 - t_0$
0	1	1	t_3	$t_3 - t_2$	$t_3 - t_2 - t_1 + t_0$	$t_3 - t_2 - t_1 + t_0$
1	0	0	t_4	t_4	t_4	$t_4 - t_0$
1	0	1	t_5	$t_5 - t_4$	$t_5 - t_4$	$t_5 - t_4 - t_1 + t_0$
1	1	0	t_6	t_6	$t_6 - t_4$	$t_6 - t_4 - t_2 + t_0$
1	1	1	t_7	$t_7 - t_6$	$t_7 - t_6 - t_5 + t_4$	$t_7 - t_6 - t_5 + t_4 - t_3 + t_2 + t_1 - t_0$

Figure 2.3: Transformation of truth table to NNF coefficients.

CHAPTER 3

COMPUTING WEIGHT FROM ALGEBRAIC NORMAL FORM

3.1 Introduction

This chapter is based on the published work [12] in which an algorithm that computes the weight of a Boolean function from its ANF is proposed. For functions acting on large number of variables ($n > 30$) and having low number of monomials in their ANF, the algorithm is advantageous over the standard method of computing weight by constructing the truth table from ANF with *Fast Möbius Transform* which requires $O(n2^n)$ operations for an n-variable function [2]. Boolean functions having special structure allowing their weight to be calculated more efficiently exist, e.g. majority functions [18].

The relation between the ANF of a Boolean function and its weight is introduced in [5]. However, a direct evaluation of this expression requires a computational complexity of $O(2^p)$ operations if the ANF contains p monomials, which become quickly infeasible as p gets larger. This limitation induced by the monomial count can be coped to some extent by exploring the expression in detail and avoiding unnecessary computations, which forms the main contribution of this study. In a relevant work based on [5], Gupta and Sarkar proposed an algorithm to compute the Walsh coefficients of a Boolean function from its ANF [20]. The drawback of the algorithm proposed in [20] is that it requires the function to be composed of high degree monomials in order to be efficiently computable. The algorithm proposed here can also be an alternative and more efficient way of computing the Walsh coefficients of a Boolean function since the Walsh coefficient of a function f at point w can be deduced from the distance of f to the linear function $l_w = \langle w, x \rangle$ which is equivalent to the weight of $f \oplus l_w$.

3.2 The Relation Between ANF and Weight

The expression of the weight of a Boolean function in terms of its ANF coefficients is given in [5, Section 3.3 (Remark)]. Here, this formula called the *weight function* is restated. Let $a_i \in \mathbb{F}_2$ for $0 \leq i \leq 2^n - 1$ be the ANF coefficients of a Boolean function. For instance, a_0 is the coefficient of 1, a_1 is the coefficient of x_n, a_{2^n-1} is the coefficient of $x_1 \dots x_n$, and so on. For a set $K = \{i_1, \dots, i_k\} \subseteq \{0, \dots, 2^n - 1\}$, $a^K = a_{i_1} \dots a_{i_k}$ is a product of ANF coefficients. The weight function is

$$F(a_0, \ldots, a_{2^n-1}) = \sum_{I \subseteq \{0,\ldots,2^n-1\}} \lambda_I a^I \tag{3.1}$$

where

$$\lambda_I = (-2)^{|I|-1} 2^{n-wt(v_{i_1} \vee \cdots \vee v_{i_k})} \tag{3.2}$$

is called the *weight coefficient* of a^I. A Boolean function containing no monomials having weight zero imposes $\lambda_I = 0$ if $I = \emptyset$. $|I|$ is called the *order part* and $wt(v_{i_1} \vee \cdots \vee v_{i_k})$ is called the *product weight part* of the weight coefficient. In the product weight part, v_{i_j} is an n-bit vector corresponding to the binary expansion of integer i_j and specifies which variables appear for the ANF coefficient a_{i_j}.

Example 3.2.1 *For a 2-variable Boolean function*

$$f(x_1, x_2) = a_0 \oplus a_2 x_1 \oplus a_1 x_2 \oplus a_3 x_1 x_2,$$

the weight function is as follows:

$$\begin{aligned} F(a_0, a_1, a_2, a_3) \quad = \quad & 4a_0 + 2a_1 + 2a_2 + a_3 \\ & -4a_0a_1 - 4a_0a_2 - 2a_0a_3 - 2a_1a_2 - 2a_1a_3 - 2a_2a_3 \\ & +4a_0a_1a_2 + 4a_0a_1a_3 + 4a_0a_2a_3 + 4a_1a_2a_3 \\ & -8a_0a_1a_2a_3. \end{aligned}$$

The weight function consists of $2^{2^n} - 1$ terms and for a Boolean function having p monomials in its ANF, exactly $2^p - 1$ of the these terms will contribute to the sum, since in (3.1), if any of the monomials in the set I is not present in the ANF, a^I becomes zero and the term vanishes. Hence, an equivalent formulation of the weight function involving only non-zero terms can be obtained. If $f(x_1, \ldots, x_n) = x^{I_1} \oplus \cdots \oplus x^{I_p}$, then the weight of f is

$$wt(f) = \sum_{J \subseteq \{1,\ldots,p\} \setminus \emptyset} (-2)^{|J|-1} 2^{n-|I_{j_1} \cup \cdots \cup I_{j_{|J|}}|}. \tag{3.3}$$

Here, J selects all non-empty monomial combinations and the value of each term in the sum is determined by the number of monomials in J and the number of distinct variables appearing in the monomials selected by J.

Example 3.2.2 *Let $f(x_1, x_2, x_3) = x_1 \oplus x_1x_2 \oplus x_2x_3$. The weight function will be evaluated with the following input:*

$$F(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) = F(0, 0, 0, 1, 1, 0, 1, 0)$$

and the non-zero weight coefficients ($a^I \neq 0$) is given in Table 3.1. The weight of the function will be the sum of the weight coefficients, which is

$$wt(f) \quad = 2 + 4 + 2 - 2 - 2 - 4 + 4 = 4.$$

The following proposition provides a quick computation of weight coefficients if both a monomial and its sub-monomials exist in the ANF of a Boolean function.

Table 3.1: Weight coefficients of $f(x_1, x_2, x_3) = x_1 \oplus x_1x_2 \oplus x_2x_3$.

I	Order	Union Weight	λ_I
$\{3\}$	1	2	$(-2)^{1-1}2^{3-2} = 2$
$\{4\}$	1	1	$(-2)^{1-1}2^{3-1} = 4$
$\{6\}$	1	2	$(-2)^{1-1}2^{3-2} = 2$
$\{3,4\}$	2	3	$(-2)^{2-1}2^{3-3} = -2$
$\{3,6\}$	2	3	$(-2)^{2-1}2^{3-3} = -2$
$\{4,6\}$	2	2	$(-2)^{2-1}2^{3-2} = -4$
$\{3,4,6\}$	3	3	$(-2)^{3-1}2^{3-3} = 4$

Proposition 3.2.3 *Let $f(x) = x^I \oplus x^{J_1} \oplus \cdots \oplus x^{J_t}$ and the monomial x^I of degree d covers all the remaining t monomials. Then the sum of the weight coefficients involving x^I is*

$$K(n,t,d) = \sum_{M=I\cup J, J\subseteq\{J_1,\ldots,J_t\}\setminus\emptyset} \lambda_M a^M = \begin{cases} 0 & \text{if } t \text{ is even;} \\ -2^{n-d+1} & \text{if } t \text{ is odd.} \end{cases} \tag{3.4}$$

Proof. Considering the weight coefficients of the covering monomial x^I and its sub-monomials, the product weight part of any monomial combination containing x^I will have d variables. So, the sum S in the weight function involving these monomials becomes;

$$\begin{aligned} S &= \sum_{i=1}^{t} \binom{t}{i} (-2)^i 2^{n-d} \\ &= 2^{n-d}[(-2+1)^t - 1] \end{aligned}$$

which is equal to (3.4). Notice that in S, the weight coefficients consisting of i sub-monomials have the order part as $(-2)^i$ instead of $(-2)^{i-1}$ because of the inclusion of the covering monomial x^I. ■

Definition 3.2.4 *Let $f(x_1, \ldots, x_n) = x_{i_1} \ldots x_{i_d} \oplus g(x_{j_1}, \ldots, x_{j_k})$ such that $\{i_1, \ldots, i_d\} \cap \{j_1, \ldots, j_k\} = \emptyset$. Then the monomial $x_{i_1} \ldots x_{i_d}$ is called an isolated monomial, meaning that the variables it depends on does not appear in any other monomial in g.*

Now, two well known properties where Prop. 3.2.6 is a special case of Prop. 3.2.5 appearing in [26] as the *randomization lemma* is stated.

Proposition 3.2.5 *If $f(x) = x^I \oplus g(x)$ and x^I is an isolated monomial of degree d then*

$$wt(f) = 2^{n-d} + wt(g) - 2^{1-d}wt(g).$$

Proof. The result follows from the fact that if a function is written as the sum of two functions $f(x) = g(x) \oplus h(x)$ then $wt(f) = wt(g) + wt(h) - 2wt(gh)$. Since x^I and $g(x)$ have no common variables, $wt(x^I g) = 2^{-d} wt(g)$. The weight of the function consisting of a single monomial of degree d being 2^{n-d} completes the proof. ■

Proposition 3.2.6 *A Boolean function containing an isolated monomial of degree* 1 *is balanced.*

Proof. Substituting $d = 1$ in Prop. 3.2.5, one gets

$$\begin{aligned} wt(f) &= 2^{n-1} + wt(g) - 2^{1-1} wt(g), \\ &= 2^{n-1}. \end{aligned}$$

■

The most important benefit of determining the isolated monomials in a function is a consequence of Prop. 3.2.6, finding the weight by checking a balancedness condition and aborting the weight computation if the condition is met. Apart from this, it is possible that there are more than one isolated monomials all having degrees greater than 1 in a function, in which case the weight of the function can be computed by first putting the isolated monomials aside, then computing the weight of the function composed of the remaining monomials and combining these together according to Prop. 3.2.5 one by one.

3.3 The Algorithm

Suppose the ANF and weight of a Boolean function composed of k monomials is given. An investigation of how the weight of the function changes if one more monomial is added to the function will be the basis of an iterative method to compute the weight of a function.

Let $f(x) = \underbrace{x^{I_1} \oplus \cdots \oplus x^{I_k}}_{g(x)} \oplus x^{I_{k+1}}$ and weight of $g(x)$ be given. Considering the weight function (3.1) or (3.3), in order to find the weight of $f(x)$, the new weight coefficients introduced by the addition of the $(k+1)^{st}$ monomial should be calculated. There are 2^k such coefficients; the coefficient of the $(k+1)^{st}$ monomial itself plus all possible non-empty $2^k - 1$ combinations of the first k monomials together with the $(k+1)^{st}$ monomial.

For the calculation of these new coefficients, two monomial sets S_1 and S_2 are formed from the monomials of $g(x)$ such that S_1 consists of the monomials which are sub-monomials of $x^{I_{k+1}}$ and S_2 consists of the differences of the remaining monomials with respect to $x^{I_{k+1}}$.

$$S_1 = \{x^{I_i} \mid I_i \subset I_{k+1}, i \in \{1, \ldots, k\}\}, \tag{3.5}$$

$$S_2 = \{x^{I_i} \setminus x^{I_{k+1}} \mid i \in \{1, \ldots, k\}\}. \tag{3.6}$$

Note that in S_2, it is possible that a monomial appears more than once. For example, $x_1x_3 \setminus x_1x_2 = x_3$ and $x_2x_3 \setminus x_1x_2 = x_3$. In this case, the monomials appearing an even number of times will be discarded and only one instance of monomials appearing an odd number of times will be stored, which is explained by the following proposition.

Proposition 3.3.1 *If x^I and x^J are two monomials whose differences with respect to a monomial x^K are equal, then the sum of weight coefficients involving these three monomials is zero.*

Proof. Let $f(x) = x^I \oplus x^J \oplus x^K \oplus g(x)$. Since $x^I \setminus x^K = x^J \setminus x^K$, the product weight part of the monomials $x^I \cup x^K$ and $x^J \cup x^K$ will be the same. Let T denote the sum of the weight coefficients involving x^I, x^K and the monomials in g. The sum of the coefficients involving x^J, x^K and the monomials in g will also be equal to T from the equality of the product weight stated before. Finally, considering the coefficients involving both x^I and x^J and the rest of the monomials, the sum will be $-2T$, canceling the previous sums, because the order part of the weight coefficients will increase by one while the product weight part remains the same. ■

The new weight coefficients introduced by the addition of monomial $x^{I_{k+1}}$ can be grouped into three sums T_1, T_2 and T_3; the coefficients formed by the monomials in S_1 and $x^{I_{k+1}}$, S_2 and $x^{I_{k+1}}$, S_1 and S_2 and $x^{I_{k+1}}$, respectively. T_1 can be computed according to (3.4). T_2 can be computed by finding the weight of the monomials in S_2 and then combining this with the monomial $x^{I_{k+1}}$ according to Prop. 3.2.5. This is a recursive weight computation call, but to a function that is simpler and contains probably less monomials because the monomials in S_2 consist of the variables not appearing in $x^{I_{k+1}}$. The value of T_3 depends on the parity of $|S_1|$. If S_1 contains an even number of monomials then according to Prop. 3.3.1 these will be zero because the monomials in S_1 are sub-monomials of $x^{I_{k+1}}$ and they will contribute to the order part of the weight coefficient but not to the product weight part. If there is an odd number of monomials in S_1 then the sum of the even parts will be zero and the contribution will be $-2T_2$ because of the order part increasing by 1 and product weight part not changing. To sum up, the new weight coefficients will be as follows:

$$
\begin{aligned}
T_1 &= K(n, |S_1|, d), && \text{(by Prop. (3.2.3))} \\
T_2 &= -2^{1-d} wt(S_2), && \text{(by Prop. (3.2.5))} \\
T_3 &= -2T_2(|S_1| \pmod 2). && \text{(by Prop. (3.2.3))}
\end{aligned}
$$

Apart from these, there is one more weight coefficient to be considered, the weight coefficient of the newly added monomial itself, which is 2^{n-d} if the degree of the monomial is d.

Combining these observations, Algorithm 3.3.1, can be used to compute the weight of a Boolean function by processing the monomials of the function. The algorithm takes as input the monomials of a Boolean function denoted by $func$, whose weight is to be computed. $func$ consists of a list of monomials and these are processed one by one in the given order. The monomials are assumed to be sorted with respect to their degrees in ascending order, to improve the performance. Global variable n defines the number of input variables. W stores the weight of the function processed up to that point and S holds the weight contribution of the processed monomial to the function, which can be negative. Insertion and removal of monomials from monomial lists will be denoted by + and − respectively. The monomial processed currently, denoted by x^I in the algorithm will be referred to as the processed monomial. There are three monomial lists other than the function itself; $ProcessedList$ is initially empty and contains the previously processed monomials as the algorithm proceeds. $SubMonList$ and $DisjointList$ are constructed for each processed monomial from scratch. $SubMonList$ is a list of monomials covered by the processed monomial. $DisjointList$ is the difference of each monomial in $ProcessedList$ with respect to the processed monomial. Notice that the even instances of

monomials in $DisjointList$ will be removed as explained in Prop. 3.3.1. Hence, if a monomial appears k times in $DisjointList$ then $k \pmod 2$ instances of it will remain. The weight of the monomials in $DisjointList$ also have to be computed (with a recursive call) and combined with the processed monomial to be added to S. The last term added to S is the weight coefficient of the processed monomial. At the end of processing of a monomial, $W + S$ will be the updated weight of the partial function composed of the processed monomials.

An example illustrating how the algorithm runs for the 4-variable Boolean function

$$f(x_1, x_2, x_3, x_4) = x_1 \oplus x_3 \oplus x_1x_3 \oplus x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3x_4 \oplus x_1x_2x_3x_4$$

is given in Table 3.2. If the processed monomial x^I has no common input variables with the variable $VarSet$ specified by the $Isolated$ column of the table, it means that x^I is an isolated monomial (considering only the monomials up to that point) and the weight contribution S is directly calculated according to Prop. 3.2.5, not requiring the calculation of $SubMonList$ and $DisjointList$. At the end of each step, $VarSet$ will be updated to include the processed monomial's input variable indices in order to determine whether the upcoming monomials are isolated or not. Next three columns denote the three monomial lists used in the algorithm, with $DisjointList$ being simplified after the cancellation of even number of monomials. Column S denotes the contribution of the processed monomial to the function's weight and column W shows the weight of the function composed of the processed monomials so far.

Table 3.2: Execution of the algorithm for $f(x_1, x_2, x_3, x_4) = x_1 \oplus x_3 \oplus x_1x_3 \oplus x_2x_4 \oplus x_1x_2x_3 \oplus x_2x_3x_4 \oplus x_1x_2x_3x_4$.

Step	x^I	VarSet	Isolated	ProcessedList	SubMonList	DisjointList	S	W
1	x_1	$\emptyset$	Yes	$\emptyset$	-	-	8	8
2	x_3	$\{1\}$	Yes	$\{x_1\}$	-	-	0	8
3	x_1x_3	$\{1,3\}$	No	$\{x_1, x_3\}$	$\{x_1, x_3\}$	$\emptyset$	4	12
4	x_2x_4	$\{1,3\}$	Yes	$\{x_1, x_3, x_1x_3\}$	-	-	-2	10
5	$x_1x_2x_3$	$\{1,2,3,4\}$	No	$\{x_1, x_3, x_1x_3, x_2x_4\}$	$\{x_1, x_3, x_1x_3\}$	$\{x_4\}$	0	10
6	$x_2x_3x_4$	$\{1,2,3,4\}$	No	$\{x_1, x_3, x_1x_3, x_2x_4, x_1x_2x_3\}$	$\{x_3, x_2x_4\}$	$\{x_1\}$	0	10
7	$x_1x_2x_3x_4$	$\{1,2,3,4\}$	No	$\{x_1, x_3, x_1x_3, x_2x_4, x_1x_2x_3, x_2x_3x_4\}$	$\{x_1, x_3, x_1x_3, x_2x_4, x_1x_2x_3, x_2x_3x_4\}$	$\emptyset$	1	11

3.3.1 An Extension Making Use of the Isolated Monomials

Algorithm 3.3.1 can be improved by making use of Prop. 3.2.5. To accomplish this, the isolated monomials should be found and removed from the function, in order to be processed after the weight of the function composed of the remaining monomials is computed. Algorithm 3.3.2 returns the list of isolated monomials in a function and also removes these monomials from the function. The variable $VarSet$ stores the variable indices appearing in the processed monomials. If a monomial x^I does not contain variables previously appeared, i.e., $VarSet \cap I = \emptyset$, this monomial is added to the $IsolatedList$. Otherwise, the monomials in $IsolatedList$ are checked for a common variable with x^I and removed from the $IsolatedList$ if found any. At the end of this process, $IsolatedList$ contains the isolated monomials of the function. In the last step, the isolated monomials are removed from the original function and $IsolatedList$ is returned as the output of the algorithm.

The extended weight computation algorithm incorporating Algorithm 3.3.2 is given as Algorithm 3.3.3. The first step of Algorithm 3.3.3 is to move the isolated monomials of $func$ to $IsolatedList$. Then, $IsolatedList$ is checked for a monomial of degree 1, indicating the balancedness of the function. With isolated monomials being separated, weight of $func$ is computed with Algorithm 3.3.1 However, a small change in this algorithm is needed. As Algorithm 3.3.1 is a recursive algorithm calling itself, the improvement provided by Algorithm 3.3.2 can be continuously made effective if the recursive call to Algorithm 3.3.1 is replaced with Algorithm 3.3.3. This way, each attempt at computing the weight of a function starts by separating the isolated monomials.

Algorithm 3.3.1: ComputeWeight($func$)

global n : Number of input variables
local $VarSet$: A set storing appeared input variable indices
local $SubMonList, DisjointList, ProcessedList$: Monomial lists
local d : Degree of a monomial
local S : Partial sum coming from the processed monomial
local W : Weight of the function consisting of processed monomials

$W \leftarrow 0$
$VarSet \leftarrow \emptyset$
$ProcessedList \leftarrow \emptyset$

for each *monomial* $x^I \in func$
$S \leftarrow 0$
$d \leftarrow deg(x^I)$
if $VarSet \cap I = \emptyset$
then $S \leftarrow -2^{1-d}W$
else
$SubMonList \leftarrow \{Term \in ProcessedList : Term \subseteq x^I\}$
$DisjointList \leftarrow \{Term \setminus x^I : Term \in ProcessedList\}$
if $DisjointList \neq \emptyset$
then $S \leftarrow -2^{1-d}$ComputeWeight($DisjointList$)
if $|SubMonList| \equiv 1 \pmod 2$
then $S \leftarrow -S - 2^{n-d+1}$
$S \leftarrow S + 2^{n-d}$
$W \leftarrow W + S$
$VarSet \leftarrow VarSet \cup I$
$ProcessedList \leftarrow ProcessedList + x^I$

return (W)

Algorithm 3.3.2: SplitIsolatedMonomials($func$)

local $VarSet$: A set storing appeared input variable indices
local $IsolatedList$: List of isolated monomials

$VarSet \leftarrow \emptyset$
$IsolatedList \leftarrow \emptyset$

for each *monomial* $x^I \in func$
{**if** $VarSet \cap I = \emptyset$
  then $IsolatedList \leftarrow IsolatedList + x^I$
  else {**for each** *monomial* $x^J \in IsolatedList$
    if $J \cap I \neq \emptyset$
      then $IsolatedList \leftarrow IsolatedList - x^J$
$VarSet \leftarrow VarSet \cup I$

comment: Remove isolated monomials from func

for each *monomial* $x^I \in IsolatedList$
$func \leftarrow func - x^I$

return ($IsolatedList$)

Algorithm 3.3.3: ComputeWeight2($func$)

local $IsolatedList$: List of isolated monomials
local W : Weight

$IsolatedList \leftarrow$ SplitIsolatedMonomials($func$)

if $\exists\, x^I \in IsolatedList$ such that $deg(x^I) = 1$
  then return (2^{n-1})

$W \leftarrow$ ComputeWeight($func$)

for each $x^I \in IsolatedList$
$W \leftarrow W + 2^{n-deg(x^I)} - 2^{1-deg(x^I)}W$

return (W)

3.4 Implementation Results and Comparison of the Complexities

Throughout the section n and p will denote the number of input variables and the number of monomials appearing in the ANF of a Boolean function, respectively. The proposed weight

computation algorithm tries to improve upon the exhaustive calculation of 2^p weight coefficients, with the worst case complexity being 2^p. Therefore, the algorithm is clearly more efficient than the traditional method using *Fast Möbius Transform* for $p \leq n$. For $p > n$ it can still perform better as the experiments indicate. However, a precise computation of the running time complexity of the algorithm should regard not only n and p, but also the possible structural relations of the monomials. That is, two functions having the same number of input variables and the same number of monomials may have quite different time complexities. Due to this, the comparison of the time complexity with Gupta-Sarkar's algorithm has been given for particular classes of Boolean functions, namely randomly generated functions and homogeneous functions.

The algorithm has been implemented in C language and timings have been collected from a PC having Intel Core-i5 processor running at 3.2GHz. Memory requirement for each setting is negligible. Table 3.3 lists the average running times of the algorithm for randomly generated Boolean functions.

Table 3.3: Timings for random functions.

n / p	32	64	128	192	256
32	<1s	<1s	<1s	2s	6s
64	<1s	<1s	15s	1m 40s	5m 56s
128	<1s	7s	6m 19s	1h 3m	4h 52m

The time complexity of Gupta-Sarkar's algorithm is given in [20] as $nS_1 \log_2(S_2)$ where the dominant factor S_1 is defined as the number of monomial combinations whose product contains less than n input variables. The value of S_1 is related to the degree distribution of the monomials of the function and can be as high as 2^p. The expected value of S_1 can be calculated with the following formula with q being the probability of an input variable appearing in a monomial, which is equal to $\frac{1}{2}$ for random functions.

$$\sum_{k=1}^{p} (1 - (1 - (1 - q)^k)^n) \binom{n}{k} \tag{3.7}$$

By the help of the above formula, corresponding expected values of $\log_2(S_1)$ for random Boolean functions are given in Table 3.4.

Table 3.4: Expected value of $\log_2(S_1)$ for random functions.

n / p	32	64	128	192	256
32	23.66	42.43	79.84	113.05	137.35
64	24.62	43.43	80.84	114.05	138.35
128	25.55	44.43	81.84	115.05	139.35

A second experiment is conducted for homogeneous Boolean functions, that is, the functions consisting of the monomials of the same degree. The number of input variables is chosen to be $n = 64$ and two different monomials counts, namely, $p = 64$ and $p = 80$ is used. The

monomials are randomly chosen from the set of possible monomials for each degree. Table 3.5 lists the timings for degrees 1 up to 32. Degrees greater than 32 are omitted because of the execution times being negligible. The timings listed in the table reflect the average execution time of five Boolean functions for the specified degree and monomial count. The column next to the execution times in the table gives the expected values of $\log_2(S_1)$ for Gupta-Sarkar's algorithm. Expected value of S_1 for homogeneous functions can be computed from (3.7) by substituting the appropriate value of q for each degree. Based on the values of S_1 in Table 3.4 and Table 3.5, one can conclude that for the specified classes of Boolean functions, the proposed weight computation algorithm is able to compute the weights of the functions for which the complexity of Gupta-Sarkar's algorithm is impractical.

3.5 Conclusion

In this chapter, an algorithm for computing the weight of a Boolean function from its ANF representation is proposed, which can be used for functions having large number of input variables where constructing the truth table by applying the Fast Möbius transform is infeasible. The algorithm improves upon the known method of computing weight due to Carlet and Guillot which required $O(2^p)$ operations for functions containing p monomials. The proposed algorithm achieves a better performance by eliminating unnecessary calculations and this fact is supported by the empirical results. The experiments also indicate that the algorithm succeeds in computing the weights for certain function classes where the expected running times of a comparable algorithm proposed by Gupta and Sarkar is infeasible.

Table 3.5: Timings for 64-variable homogeneous functions and the corresponding expected values of $\log_2(S_1)$.

	$p = 64$		$p = 80$	
Degree	Time (sec.)	$\log_2(S_1)$	Time (sec.)	$\log_2(S_1)$
1	0	64.00	0	80.00
2	15	64.00	1605	80.00
3	614	63.99	40511	79.99
4	4613	63.99	93730	79.98
5	5540	63.97	190725	79.84
6	8625	63.87	211452	79.50
7	9551	63.63	192942	78.93
8	5903	63.25	120866	78.21
9	4840	62.74	73298	77.38
10	4109	62.14	51803	76.49
11	2902	61.47	29039	75.56
12	1863	60.75	16500	74.59
13	1395	60.00	10780	73.61
14	852	59.23	6624	72.61
15	595	58.44	4262	71.60
16	404	57.63	2483	70.58
17	251	56.82	1512	69.55
18	155	55.99	899	68.50
19	107	55.15	525	67.45
20	66	54.30	340	66.39
21	42	53.44	210	65.31
22	28	52.58	128	64.23
23	19	51.70	82	63.13
24	13	50.82	53	62.03
25	8	49.93	35	60.91
26	6	49.03	23	59.79
27	4	48.12	16	58.65
28	3	47.20	10	57.50
29	2	46.27	7	56.34
30	1	45.34	4	55.17
31	1	44.39	3	53.99
32	0	43.43	2	52.79

CHAPTER 4

COMPUTING NONLINEARITY FROM ALGEBRAIC NORMAL FORM

4.1 Introduction

In this chapter, an algorithm for computing the nonlinearity of a Boolean function from its algebraic normal form (ANF) is proposed. Among the properties associated with a Boolean function, nonlinearity is an important criterion regarding the security point of view. Nonlinearity is defined as the minimum distance of a Boolean function to the set of affine functions and the functions used in secrecy systems are expected to have high nonlinearity in order to be resistant against certain cryptanalytic attacks. This makes the nonlinearity computation a necessary task in order to prove that the claimed level of security is achieved.

Computing nonlinearity from the truth table can be done via Fast Walsh transform (FWT), which constructs the Walsh spectrum of the function and the entry with the maximum absolute value in the spectrum determines the nonlinearity. As the truth table of an n-variable Boolean function consists of 2^n entries, the cost of storing the truth table increases exponentially in n. ANF is a preferred representation either when it is impractical to store the truth table, or it is more efficient and/or secure to calculate the output of the Boolean function from an expression of the input variables.

Unless the ANF of a Boolean function belongs to a class that reveals its nonlinearity (e.g. affine functions, bent functions), the task of computing the nonlinearity from the ANF can be performed by constructing the truth table from the ANF by Fast Möbius transform and then applying FWT on it. This process has a computational complexity of $O(n2^n)$ both to transform the ANF to the truth table and to apply FWT. Clearly, when n gets larger, say $n > 40$, considering the computational and memory resources of current computers, performing these transformations becomes infeasible.

Expression of the weight of a Boolean function in terms of its ANF coefficients is introduced by Carlet and Guillot [5], which allows one to compute the weight from the ANF with a complexity of $O(2^p)$ operations, if the Boolean function consists of p monomials. Two related works utilizing this expression propose more efficient methods for Walsh coefficient computation and weight computation from the ANF [20, 12], with the latter one being the subject of Chapter 3.

In this chapter, by investigating the distance of a Boolean function to the set of affine functions

in terms of ANF coefficients, an algorithm to compute the nonlinearity for Boolean functions with large number of inputs is devised.

4.2 Distance to Linear Functions

Any integer valued function $G : \mathbb{F}_2^m \to \mathrm{Z}$ defined on binary m-tuples can be represented by

$$G(x_1, \ldots, x_m) = \sum_{I \subseteq \{1,\cdots,m\}} \lambda_I x^I \tag{4.1}$$

where $x_i \in \mathbb{F}_2$ for $1 \leq i \leq m$ and $\lambda_I \in \mathrm{Z}$ is the coefficient of the product $x^I = x_{i_1} \ldots x_{i_d}$ for $I = \{i_1, \cdots, i_d\}$. Here, $x_i \in \mathbb{F}_2$ implies $x_i^2 = x_i$, therefore all the terms x^I are distinct products of input variables and the function has 2^m terms. For the functions of interest to this study, G maps the ANF coefficients of an n-variable Boolean function to the distance to a particular linear function, hence the number of input variables is $m = 2^n$. When m is large, it is not possible to list all of the ANF coefficients of a Boolean function. Instead, the support of the ANF is supplied, which is assumed to be relatively small sized. For an input $x = (x_1, \ldots, x_m)$ and $supp(x) = \{i_1, \cdots, i_p\}$, the output of the function will be the sum

$$G(x) = \sum_{I \subseteq \{i_1,\cdots,i_p\}} \lambda_I$$

consisting of 2^p coefficients λ_I, associated with all nonzero products x^I. Using this approach, the function G can be evaluated in $O(2^p)$ operations if the complexity of computing each λ_I is negligible. In the following part of this section, it will be shown that the coefficients of the functions mapping the ANF coefficients to the weight and to the distance to a particular linear function can be computed in a very simple way. On the contrary, this is not the case for nonlinearity. This will lead to constructing a method to compute the nonlinearity without trying to compute the coefficients of the function, but by exploiting the properties of the previously mentioned functions' coefficients that could be easily computable.

The function mapping the ANF coefficients to the weight of a Boolean function was introduced in [5], which will be called the *weight function*. In order to render the remaining of the text more comprehensible, how the coefficients of this function are derived is going to be explained.

The output of a Boolean function can be calculated in terms of its ANF coefficients as follows:

$$f(x) = \bigoplus_{u \preceq x} a_u. \tag{4.2}$$

Table 4.1 shows the truth table entries of 3-variable Boolean functions in terms of ANF coefficients. When an ANF coefficient a_i contributes to the output of a Boolean function at a point, it is said that a_i appears in that truth table entry. The sum of the truth table entries gives the weight of the function. Weight can be expressed as a function over the integers by replacing the addition operation in $\mathbb{F}_2$ in each truth table entry with integer addition operation. $\mathbb{F}_2$ addition can be converted to addition over the integers by a well known formula, generalizing the fact that $a \oplus b = a + b - 2ab$:

$$\bigoplus_{i=1}^{m} a_i \quad = \quad \sum_{k=1}^{m} (-2)^{k-1} \sum_{1 \le i_1 < \cdots < i_k \le m} a_{i_1} \cdots a_{i_k}. \tag{4.3}$$

This formula states that the expression of addition over the integers consists of all combinations of products of terms, with a leading coefficient related to the number of terms in the product, such as all degree two terms having the coefficient -2 and all degree three terms having the coefficient 4, etc.

Proposition 4.2.1 *For an n-variable Boolean function, ANF coefficient a_u contributes to the output in $2^{n-wt(u)}$ points.*

Proof. From (4.2), a_u contributes to the output of the function at a point x if $u \preceq x$. This means that a_u contributes to the function output at points $\overline{S}(u)$, whose size is equivalent to $2^{n-wt(u)}$ by (2.2). ■

Proposition 4.2.2 *For each set $A = \{a_{u_1}, \cdots, a_{u_k}\}$ of ANF coefficients, there exists a coefficient a_v with the property $supp(v) = \bigcup_{1 \le i \le k} supp(u_i)$, such that the truth table entries a_v appears are exactly the same as the truth table entries all the coefficients in A appear together. Such a coefficient a_v will be called the representative coefficient of $\prod_{1 \le i \le k} a_{u_i}$.*

Proof. From (4.2), each a_{u_i} appears in truth table entries $\overline{S}(u_i)$. If y is a truth table entry such that all a_{u_i}'s appear together then $\bigcup_{1 \le i \le k} supp(u_i) \subseteq supp(y)$. Call the minimum weight vector satisfying this property v, the case where the set equality occurs. Then, by definition of a super-vector, the set $\overline{S}(v)$ also has the same property of covering the supports of u_i's, and these are the points the ANF coefficient a_v appears in the truth table. Therefore, if vector v is chosen such that $supp(v) = \bigcup_{1 \le i \le k} supp(u_i)$, a_v appears at exactly the same truth table entries as $\{a_{u_1}, \cdots, a_{u_k}\}$ appear together. ■

Combining Proposition 4.2.1 and (4.3), one obtains the weight function of an n-variable Boolean function as follows:

$$F(a_0, \ldots, a_{2^n-1}) = \sum_{I \subseteq \{0,\ldots,2^n-1\}} \lambda_I a^I \tag{4.4}$$

where $\lambda_I \in \mathrm{Z}$ is called the *weight coefficient* of the product $a^I = \prod_{i \in I} a_i$. If $I = \emptyset$, the value of λ_I is found to be zero. For a non-empty set $I = \{i_1, \cdots, i_k\}$, the value of λ_I is determined by two factors:

$$\lambda_I \quad = \quad d_I n_I, \tag{4.5}$$

$$d_I \quad = \quad (-2)^{k-1}, \tag{4.6}$$

$$n_I \quad = \quad 2^{n-wt(\overline{i_1} \vee \cdots \vee \overline{i_k})}. \tag{4.7}$$

Table 4.1: Truth Table in terms of ANF Coefficients.

x_1	x_2	x_3	ANF	Truth table
0	0	0	a_0	a_0
0	0	1	a_1	$a_0 \oplus a_1$
0	1	0	a_2	$a_0 \oplus a_2$
0	1	1	a_3	$a_0 \oplus a_1 \oplus a_2 \oplus a_3$
1	0	0	a_4	$a_0 \oplus a_4$
1	0	1	a_5	$a_0 \oplus a_1 \oplus a_4 \oplus a_5$
1	1	0	a_6	$a_0 \oplus a_2 \oplus a_4 \oplus a_6$
1	1	1	a_7	$a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7$

The value of d_I comes from (4.3) and n_I is the number of times the product a^I occurs in the truth table entries expressed in integer addition, which is related to the number of distinct input variables appearing in the monomials of contributing ANF coefficients.

Example 4.2.3 *Let $n = 3$ and consider the weight coefficient of a_1a_2 in the weight function. After the conversion of binary addition to integer addition, a_1a_2 appears in a truth table entry if and only if both of a_1 and a_2 are present at that entry. According to Proposition 4.2.2, a_3 appears as many times as a_1a_2 appears in the truth table since $\overline{1} \vee \overline{2} = \overline{3}$, and the entries they appear are exactly the same, which are the fourth and the last rows of Table 4.1. Hence, by Proposition 4.2.1, a_1a_2 appears $2^{3-wt(\overline{3})} = 2$ times and since a_1a_2 consists of two terms, each one of these terms will have the constant $(-2)^{2-1} = -2$ arising from the conversion of addition (4.3). As a result, the coefficient of a_1a_2 becomes $(-2).2 = -4$.*

4.2.1 The Linear Distance Matrix

Now, the method of obtaining the coefficients of the function which outputs the distance of a Boolean function to a linear function, in terms of ANF coefficients will be described. Essentially, the distance between a Boolean function f and a linear function l_w is equivalent to $wt(f \oplus l_w)$. So, an investigation of how the coefficients in the weight function change when the truth table of f is merged with a linear function is necessary. The weight function when computed with new coefficients will be called the *distance function*, producing the output $d(f, l_w)$. This is a generalization of the weight function defined as

$$F_w(a_0, \ldots, a_{2^n-1}) = \sum_{I \subseteq \{0,\ldots,2^n-1\}} \lambda_I^w a^I \tag{4.8}$$

where $w \in \mathbb{F}_2^n$ specifies which linear function the distance is measured, $\lambda_I^w \in \mathrm{Z}$ is the *distance coefficient* of the product a^I. When $w = \overline{0}$, (4.8) is equivalent to (4.4) and outputs the weight. The following proposition states how the coefficients λ_I^w are obtained.

Proposition 4.2.4 *Let λ_I be the weight coefficient of a^I in the weight function of an n-variable Boolean function f and let a_v be the representative ANF coefficient of a^I. The distance coefficient λ_I^w of a^I in the distance function F_w for a nonzero $w \in \mathbb{F}_2^n$ is*

$$\lambda_I^w = \begin{cases} 2^{n-1}, & \text{if } I = \emptyset, \\ \lambda_I, & \text{if } I \neq \emptyset \text{ and } w \preceq v \text{ and } wt(w) \text{ is even}, \\ -\lambda_I, & \text{if } I \neq \emptyset \text{ and } w \preceq v \text{ and } wt(w) \text{ is odd}, \\ 0, & \text{otherwise}. \end{cases}$$

Proof. Adding a nonzero linear function l_w to f complements the truth table of f at 2^{n-1} points. At these points, the new value of the function becomes $1 \oplus f(x)$, which is equivalent to $1 - f(x)$ in integer arithmetic. Considering the integer valued expression of the truth table entries in terms of ANF coefficients, this corresponds to negating the terms and producing a constant value of 1 that is independent of the ANF coefficients. This proves $\lambda_I^w = 2^{n-1}$ for $I = \emptyset$.

In order to find out the values of other coefficients, at how many points $supp(l_w)$ coincides with the truth table entries the product a^I appears must be calculated. If all (resp. half, none) of the points where a^I appears in the truth table coincide with $supp(l_w)$, then λ_I^w will be $-\lambda_I$ (resp. 0, λ_I).

Linear function l_w identified by the vector $w \in \mathbb{F}_2^n$ has $supp(w)$ terms and takes on the value 1 whenever an odd sized combination of its terms are added. Namely,

$$supp(l_w) = \{x \in \mathbb{F}_2^n \mid \#\{supp(x) \cap supp(w)\} \text{ is odd.}\}. \tag{4.9}$$

This also means that, if $x \in supp(l_w)$ then $supp(x) = I \cup J$ such that $I \subseteq supp(w)$ with $|I| \equiv 1 \pmod 2$ and $J \subseteq \{1, \ldots, n\} \setminus supp(w)$, i.e., the components which are not in the support of w can be chosen freely since they do not contribute to the output of l_w. On the other hand, if a_v is the representative ANF coefficient of the product a^I, the truth table entries where a^I appears is $\overline{S}(v)$ by (4.2).

- Assume $w \preceq v$. For a vector $x \in \mathbb{F}_2^n$ to be both in $supp(l_w)$ and $\overline{S}(v)$, $supp(w) \subseteq supp(x)$ is necessary. Otherwise, if any component $j \in supp(w)$ of x is taken to be zero, x will not be in $\overline{S}(v)$, because $j \in supp(w)$ and $w \preceq v$ implies $j \in supp(v)$, which means the j^{th} component will always be 1 in $\overline{S}(v)$. Hence, intersection occurs at the points $\overline{S}(w)$, i.e., all the terms in l_w must be chosen. If $wt(w)$ is even, the linear function l_w gets the value zero at these points and the intersection becomes the empty set, proving $\lambda_I^w = \lambda_I$. Following the same argument, if $wt(w)$ is odd and all the terms in l_w are chosen, l_w attains the value 1 at the points $\overline{S}(w)$. Since $w \preceq v$ implies $\overline{S}(v) \subseteq \overline{S}(w)$, all the points the term a_v appears in the truth table coincide with $supp(l_w)$ and the terms at these points will be negated due to complementation, leading to $\lambda_I^w = -\lambda_I$.

- Assume $w \not\preceq v$. This implies $A = supp(w) \setminus supp(v) \neq \emptyset$. Let $supp(x) = supp(w) \cap supp(v)$ for an $x \in \mathbb{F}_2^n$. Then it is easy to show that half of the vectors in $\overline{S}(x)$ have even weight and half of them have odd weight. Because for any $y_1 \in \overline{S}(x)$ with $|supp(y_1) \cap A| \equiv 1 \pmod 2$ a corresponding vector y_2 such that $|supp(y_2) \cap A| \equiv 0 \pmod 2$ can be found. A consequence of this is the output of the linear function l_w at points y_1 and y_2 are complements of each other. This means that half of the vectors in $\overline{S}(v)$ are also in

Table 4.2: Linear Distance Matrix for $n = 3$.

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
l_0	8	4	4	2	4	2	2	1
l_1	0	-4	0	-2	0	-2	0	-1
l_2	0	0	-4	-2	0	0	-2	-1
l_3	0	0	0	2	0	0	0	1
l_4	0	0	0	0	-4	-2	-2	-1
l_5	0	0	0	0	0	2	0	1
l_6	0	0	0	0	0	0	2	1
l_7	0	0	0	0	0	0	0	-1

$supp(l_w)$ and half of them are not. Therefore, in the summation of truth table entries at positions $\overline{S}(v)$, terms cancel each other making $\lambda_I^w = 0$.

■

Now consider all 2^n functions F_w which map the ANF coefficients of a Boolean function f to the distance $d(f, l_w)$, with the attention being on the distance coefficients of a^I with $|I| = 1$, i.e., the distance coefficients of a_i for $i \in \{0, \cdots, 2^n - 1\}$. According to (4.2.2), for any product a^I with $|I| > 1$, a representative coefficient from this set can be used. The $2^n \times 2^n$ matrix whose i^{th} row consists of such coefficients $\lambda_I^{\bar{i}}$ will be called the *Linear Distance Matrix (LDM)* of order n, denoted by M^n. In view of Proposition 4.2.4, each entry of LDM can be defined as follows:

$$M^n_{i,j} = \begin{cases} (-1)^{wt(\bar{i})} 2^{n-wt(\bar{j})}, & \text{if } \bar{i} \preceq \bar{j}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.10}$$

Table 4.2 shows the LDM of order 3. The entries of n^{th} order LDM are closely related to the Sylvester-Hadamard matrix H_n, which is defined as

$$H_0 = [1], \tag{4.11}$$

$$H_n = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes H_{n-1}, \text{ for } n \geq 1, \tag{4.12}$$

where $\otimes$ refers to the Kronecker product of matrices. Each row (or column) of H_n represents the truth table of a linear function, whose entries are transformed from $(0, 1)$ to $(1, -1)$. Table 4.3 shows H_3 where only the signs of the entries are shown. '+' and '-' denote the points the function takes on the values 0 and 1 respectively.

Proposition 4.2.5 *$M^n_{i,j}$ can be obtained by adding the entries of the i^{th} row of H_n at columns $\overline{S}(\bar{j})$:*

Proof. Since the i^{th} row of H_n represents the truth table of $l_{\bar{i}}$, the distance coefficient of a_v in the distance function can be calculated by adding the '+' and '-' values of H_n in the i^{th} row at

Table 4.3: H_3: Sylvester-Hadamard Matrix of order three.

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
l_0	+	+	+	+	+	+	+	+
l_1	+	-	+	-	+	-	+	-
l_2	+	+	-	-	+	+	-	-
l_3	+	-	-	+	+	-	-	+
l_4	+	+	+	+	-	-	-	-
l_5	+	-	+	-	-	+	-	+
l_6	+	+	-	-	-	-	+	+
l_7	+	-	-	+	-	+	+	-

columns $\overline{S}(v)$. This gives how many times the sign of a_v will be positive and negative in the integer valued expression of truth table entries when the linear function $l_{\bar{i}}$ is added to a Boolean function. This sum corresponds to the distance coefficient of a_v. ■

LDM can also be expressed with the following recursive structure:

$$M^{n,0} = [1], \tag{4.13}$$

$$M^{n,i} = \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} \otimes M^{n,i-1}, \text{ for } 1 \leq i \leq n, \tag{4.14}$$

$$M^{n,n} = M^n. \tag{4.15}$$

This recursive structure can be explained with Sylvester-Hadamard matrices. As stated in Proposition 4.2.5, the entries of M^n correspond to the sum of particular entries of H_n. When the dimension is extended from n to $n+1$, H_n grows according to (4.11). As a result of the way H_n is duplicated to produce H_{n+1}, the values of M^n are doubled for ANF coefficients that do not contain the newly introduced variable, which corresponds to the upper left quarter of M^{n+1}. The upper right quarter of M^{n+1} will be equal to M_n as this part of H_{n+1} is equal to H_n. The lower right quarter will be $-M^n$ since the entries of H_{n+1} at this part have opposite signs with H_n, and the lower left quarter will be the matrix consisting of all zeros because for each $\overline{S}(v)$, half of the entries will be positive and the other half will be negative.

The first row of the LDM contains the distance coefficients for calculating the distance to the linear function $l_0 = 0$, which also corresponds to the weight. The coefficients in this row are also equivalent to the weight coefficients of the weight function. The second and third rows contain the distance coefficients for calculating the distances to the linear functions $l_1 = x_n$ and $l_2 = x_{n-1}$ respectively, and so on. The first row of the LDM will often be an exceptional case for the rest of the discussions in this chapter because once the weight of the function is calculated in the first step of the nonlinearity computation algorithm which is going to be explained in the next section, this row will no longer be needed. Some properties of the LDM derived from (4.10) are as follows:

Remark 4.2.6 *Entries in the j^{th} column take on the values from the set $\{0, \pm 2^k\}$, where $k = n - wt(\bar{j})$.*

Except the first and the last columns of the LDM, all three values mentioned in Remark 4.2.6 appear in a column. In the first column, there is only one nonzero entry which is positive and all the other entries are zero. In the last column, there is no entry with a zero value.

Remark 4.2.7 *Nonzero entries of the j^{th} column are at positions x where $\overline{x} = \underline{S}(\overline{j})$.*

Remark 4.2.8 *Nonzero entries of the i^{th} row are at positions x where $\overline{x} = \overline{S}(\overline{i})$.*

Remark 4.2.9 *Nonzero entries of the i^{th} row are positive if $wt(\overline{i})$ is even, and negative if $wt(\overline{i})$ is odd.*

Remark 4.2.10 *Let j_1 and j_2 be two column indices in the LDM. Then the following holds:*

- *If $\overline{j_1} \preceq \overline{j_2}$ then $M_{i,j_1} \neq 0$ implies $M_{i,j_2} \neq 0$.*
- *If $supp(\overline{j_1}) \cap supp(\overline{j_2}) = \emptyset$ then at least one of M_{i,j_1} and M_{i,j_2} is zero, except for $i = 0$.*

Remark 4.2.11 *Zero entries of the j^{th} column are at positions x where $supp(\overline{x}) \cap (\{1, \cdots, n\} \setminus supp(\overline{j})) \neq \emptyset$.*

Proposition 4.2.12 *If the i^{th} row of the LDM is used to measure the distance to the linear function $l_{\overline{i}}$, negating each entry of the i^{th} row is used to measure the distance to the affine function $l'_{\overline{i}}$.*

Proof. Let the distance of a Boolean function to a nonzero linear function be expressed as

$$d(f, l_i) = 2^{n-1} + \alpha$$

where α is the sum of the distance coefficients except the constant coefficient. The value of α also corresponds to the sum of certain entries of the i^{th} row of the LDM, with each entry being multiplied with a constant depending on the Boolean function f, which will be explained in the next subsection. Regardless of this multiplication, if the distance coefficients in the i^{th} row of the LDM are negated, the new sum becomes $2^{n-1} - \alpha$, and this is equivalent to $d(f, l_i \oplus 1)$, since $d(f, l_i \oplus 1) = 2^n - d(f, l_i)$.

■

In view of Proposition 4.2.12, all entries of the LDM can be considered as absolute values. This results in computing the distance to the linear function l_w if $wt(w)$ is even and to the affine function l'_w if $wt(w)$ is odd.

4.2.2 Combining Coefficients

A Boolean function consisting of p monomials has 2^p distance coefficients associated with the nonzero products of ANF coefficients, which can be computed according to (4.5). Computing the weight of the function requires all these coefficients to be added whereas the distance to a particular linear function can be obtained by adding a subset of these coefficients plus a constant value of 2^{n-1}. Computing the distance coefficients by processing all combinations of p monomials requires a computational complexity of $O(2^p)$ operations, and this can be done at most for $p < 40$ in practice. Now, a new method will be introduced which combines the related distance coefficients, with the aim of reducing the number coefficients and avoiding the processing of all 2^p monomial combinations.

Let a^I and a^J be two terms in the distance function such that $\bigcup_{i \in I} supp(\bar{i}) = \bigcup_{j \in J} supp(\bar{j})$, that is, the input variables appearing in ANF coefficients of a^I are the same as input variables appearing in ANF coefficients of a^J. This not only makes $n_I = n_J$ according to (4.7), but also specifies that these terms appear in exactly the same truth table entries according to Proposition 4.2.2. Hence, in the distance function (4.8), for all values of w, the distance coefficients of a^I and a^J will behave the same with respect to Proposition 4.2.4. The distance coefficients of these terms can be collected under the distance coefficient of the term $a^K = a_k$ such that a_k is the representative ANF coefficient of both a^I and a^J. Since $n_I = n_J = n_K$, it is $|I|$ and $|J|$ that distinguishes the distance coefficients λ_I and λ_J from λ_K. From (4.6), it follows that $\lambda_I = (-2)^{|I|-1}\lambda_K$ and $\lambda_J = (-2)^{|J|-1}\lambda_K$. The distance coefficients of the terms which have the same representative ANF coefficients can be collected under a single distance coefficient, called the *representative distance coefficient*, that is, the distance coefficient of the term belonging to the representative ANF coefficient. When this is done, the number of times a representative distance coefficient should be added will be called the *combined coefficient*, and when multiplied with the value of the representative distance coefficient, will be called the *combined distance coefficient*. The combined coefficients of the distance function can be computed from the ANF coefficients of a Boolean function as follows:

$$C_u = \sum_{\overline{u}=\overline{u_1} \vee \cdots \vee \overline{u_k}} (-2)^{k-1} \prod_{1 \le i \le k} a_{u_i}. \tag{4.16}$$

Example 4.2.13 *Let the support of the ANF coefficients for a Boolean function be* $\{a_1, a_2, a_3\}$. *Then the distance coefficients corresponding to the nonzero products*

$$\{1, a_1, a_2, a_3, a_1a_2, a_1a_3, a_2a_3, a_1a_2a_3\}$$

are

$$\{\lambda, \lambda_1, \lambda_2, \lambda_3, \lambda_{1,2}, \lambda_{1,3}, \lambda_{2,3}, \lambda_{1,2,3}\}$$

where λ *is the constant term,* λ_1 *is the distance coefficient of* a_1, $\lambda_{1,2}$ *is the distance coefficient of* a_1a_2, *and so on. Since* $\lambda_{1,2} = \lambda_{1,3} = \lambda_{2,3} = -2\lambda_3$ *and* $\lambda_{1,2,3} = 4\lambda_3$ *from (4.6) and (4.7), these distance coefficients can be combined under the representative distance coefficient* λ_3, *producing the combined distance coefficients* $\{\lambda, \lambda_1, \lambda_2, -\lambda_3\}$. *Here, the corresponding combined coefficients are* $\{1, 1, 1, -1\}$.

Algorithm 4.2.1 calculates the combined coefficients from the ANF by processing each monomial one by one. The input to the algorithm is a list of monomials of the Boolean function

called `MonList`. The monomials are represented by $x^I = \prod_{i \in I} x_i$ where $I \subseteq \{1, \cdots, n\}$. The output of the algorithm is a list of combined coefficients called `CoefList` consisting of elements of the form $C_i x^I$ where $C_i \in \mathbb{Z}$ is the combined coefficient of the monomial x^I. When a new monomial x^I is added to the function, the product of each existing combined coefficients and the new monomial is processed and the newly produced coefficients are added to a temporary list named `NewList`. For an existing coefficient $C_j x^J$, if it is the case that $I \subseteq J$, then the product of these two coefficients will be $-2C_j x^J$, when added will negate the original coefficient. The products of terms whose n_I part as specified in (4.7) will reside in the new monomial x^I are collected under the variable `S`. These terms will be added to the combined coefficients at the end of processing that monomial. The last case is the general case where the coefficient of the term produced by the product of two monomials are added to the `NewList`. At the end of processing the combined coefficients of the previous step, all the newly produced coefficients are added to `CoefList`. Addition of items to lists is denoted by + operator. When an entry $C_i x^I$ is to be added to a coefficient list, if there already exists an element $C_j x^I$, then the coefficient $C_i x^I$ is updated as $(C_i + C_j)x^I$.

Algorithm 4.2.1: CalculateCombinedCoefficients($MonList$)

$CoefList \leftarrow \emptyset$
for each $x^I \in MonList$
$NewList \leftarrow \emptyset$
$S \leftarrow 1$
for each $C_j x^J \in CoefList$
if $I \subseteq J$
then $C_j \leftarrow -C_j$
else if $J \subset I$
then $S \leftarrow S - 2C_j$
else $NewList \leftarrow NewList + (-2C_j x^{I \cup J})$
for each $C_k x^K \in NewList$
$CoefList \leftarrow CoefList + C_k x^K$
$CoefList \leftarrow CoefList + S\, x^I$
return ($CoefList$)

The sum of all combined distance coefficients will give the weight of the function. In order to compute the distance to a nonzero linear function l_w, only a subset of these coefficients need to be added, which is determined according to whether the vector associated with the combined distance coefficient is contained in $\overline{S}(w)$. Also, since $\lambda_I^w = 2^{n-1}$ for $w \neq \overline{0}$ and $I = \emptyset$, a constant value of 2^{n-1} should be added.

Since the distance to a linear functions is related to the Walsh coefficients of a Boolean function, the sum of the combined distance coefficients can be expressed in terms of Walsh coefficients.

For $w \neq \overline{0}$, i.e., for a nonzero linear function l_w,

$$W_f(w) = 2^n - 2d(f, l_w), \tag{4.17}$$

$$W_f(w) = 2^n - 2(2^{n-1} + \alpha_w), \tag{4.18}$$

$$\alpha_w = -\frac{W_f(w)}{2} \tag{4.19}$$

where α_w is the sum of the distance coefficients, excluding the constant coefficient 2^{n-1}. From (4.19) it can be seen that if the constant term 2^{n-1} is not added, the distance function (4.8) outputs $-\frac{W_f(w)}{2}$. Because the nonlinearity of a Boolean function depends on the maximum absolute value of the Walsh spectrum (2.22), being able to compute the maximum absolute value of the sum of the distance coefficients without adding the constant term is also sufficient to find out the nonlinearity.

4.3 Computing Nonlinearity

The nonlinearity computation algorithm consists of two phases. In the first phase, combined distance coefficients are calculated according to Algorithm 4.2.1 from the given ANF coefficients. Once this phase is completed, the distance to any linear function can be obtained by adding a subset of these coefficients, i.e., by adding the coefficients corresponding to columns $\overline{S}(w)$ in the w^{th} row if the distance to the linear function l_w is to be calculated. The nonlinearity computation on the other hand, requires all the distances to the linear functions to be calculated and the one with the minimum value being identified, which cannot be done in practice if n is too high. After the combined distance coefficients for a Boolean function are calculated, the distance to any linear function can be represented of the form

$$F(b_1, \cdots, b_k) = \sum_{1 \leq i \leq k} \beta_i b_i \tag{4.20}$$

where $b_i \in \mathbb{F}_2$ and $\beta_i \in \mathrm{Z}$ is the combined distance coefficient associated with b_i. Each b_i in this function determines whether a zero or a nonzero entry is chosen from the corresponding column of the LDM. Although there are 2^k possible inputs to this function, only some of the k-bit inputs actually correspond to a distance to a linear function. By enumerating all such k-bit inputs, one obtains the set of all distinct distances. The task of computing the nonlinearity then corresponds to finding the minimum of these values. Note, however, as explained in the previous section, omitting the addition of the constant coefficient when calculating the distance to the nonzero linear functions, one gets the negative half value of the Walsh coefficient, and this constant coefficient is assumed to be excluded in (4.20). With this slight modification, it becomes the maximum absolute value of (4.20) to be found, instead of the minimum and maximum values, had the constant coefficient been added.

In the second phase of the nonlinearity computation algorithm, the maximum absolute value of the distance function (4.20) is searched, which determines the nonlinearity. This problem also corresponds to a binary integer programming problem. The set of 2^k possible k-bit input vectors will be classified as feasible or infeasible according to whether they represent a distance to a linear function or not. The feasibility checking of inputs can be performed with Algorithm 4.3.1 and Algorithm 4.3.2 which determine whether a particular zero/nonzero choice of values

in certain columns is possible in any of the rows of the LDM. By enumerating all possible distances to the set of linear functions, the minimum of these can be taken as the nonlinearity of the Boolean function.

A common approach in solving integer programming problems is to utilize the tree structure. The set of feasible inputs to the function (4.20) can be shown in a tree with the input variables b_i being the nodes. This tree will be called the *distance tree*. Starting with b_1 as the root node, each left (resp. right) child node of a node b_i represents the case where $b_i = 1$ (resp. $b_i = 0$). In order to enumerate feasible inputs, it is sufficient to check whether a node b_i can take on the value 0 or 1 depending on the values of the parent nodes (values of the preceding variables). This can be done efficiently by using the facts mentioned in Remarks (4.2.10), (4.2.7) and (4.2.11). Row indices r in the LDM containing a nonzero entry in the i^{th} column satisfy $\overline{r} \preceq \overline{i}$. Similarly, if a row has a zero value in the j^{th} column then $supp(\overline{r}) \cap (\{1, \cdots, n\} \setminus supp(\overline{j})) \neq \emptyset$ must be satisfied since a zero value in the j^{th} column appears only in the row indices where at least one bit is set that is not in $supp(\overline{j})$. Given two lists of columns C_0 and C_1, and another column identified by the index u, Algorithm 4.3.1 determines whether there exists a row in the LDM containing a nonzero value in column u, with the condition that the values in columns C_0 are zero and the values in columns C_1 are nonzero. Algorithm 4.3.2 performs the same task by checking whether there is a row having a zero value in column u, under the same conditions. These two algorithms allow one to enumerate all feasible inputs to (4.20) by using the associated column indices for each input variable b_i.

Algorithm 4.3.1: BRANCH1(C_0, C_1, u)

$IncludeMask \leftarrow \bigwedge_{a \in C_1} \overline{a}$

if $(\overline{u} \wedge IncludeMask) = \overline{0}$
  then return $(false)$
  else for $i \in C_0$
if $(\neg\overline{i} \wedge \overline{u} \wedge IncludeMask) = \overline{0}$
  then return $(false)$

return $(true)$

Algorithm 4.3.2: BRANCH0(C_0, C_1, u)

$IncludeMask \leftarrow \bigwedge_{a \in C_1} \overline{a}$

if $(\neg\overline{u} \wedge IncludeMask) = \overline{0}$
  then return $(false)$
  else for $i \in C_0$
if $(\neg\overline{i} \wedge IncludeMask) = \overline{0}$
  then return $(false)$

return $(true)$

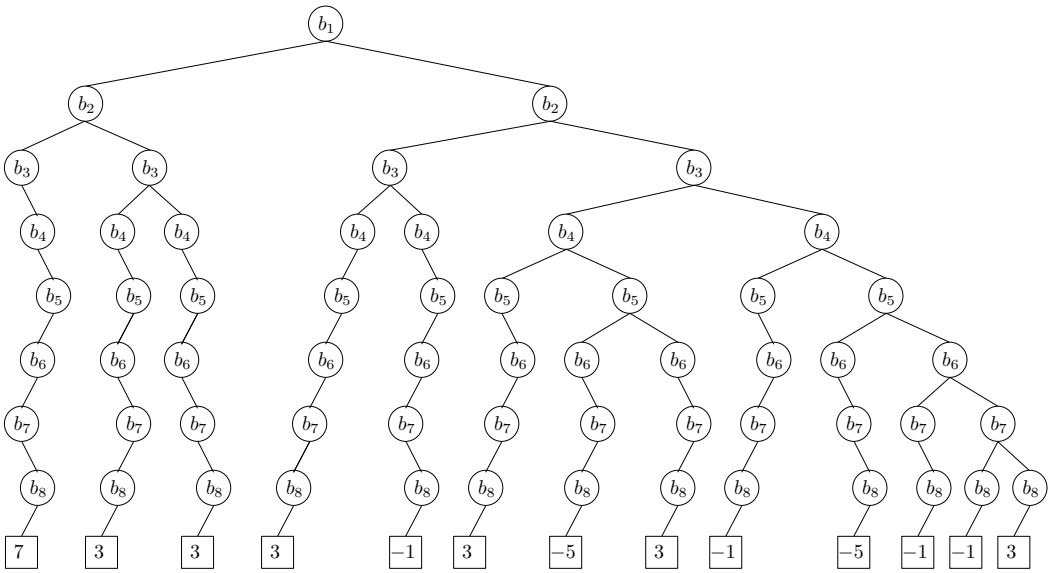

Figure 4.1: Distance tree of $F(b_1, \cdots, b_8) = 8b_1 + 8b_2 + 4b_3 + 4b_4 - 8b_5 - 4b_6 - 4b_7 + 3b_8$.

Example. Let $f(x_1, \cdots, x_5) = x_1x_5 \oplus x_4x_5 \oplus x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_2x_3x_4x_5$. Using Algorithm 4.2.1, the following set of combined distance coefficients are obtained:

$$\{\lambda_3, \lambda_{17}, \lambda_{26}, \lambda_{28}, -2\lambda_{19}, -2\lambda_{29}, -2\lambda_{30}, 3\lambda_{31}\}.$$

The associated distance function formed by these coefficients is

$$F(b_1, \cdots, b_8) = 8b_1 + 8b_2 + 4b_3 + 4b_4 - 8b_5 - 4b_6 - 4b_7 + 3b_8.$$

Note that in the LDM of order 5, b_1 is associated with column a_3, b_2 is associated with column a_{17}, and so on. Among the $2^8 = 256$ possible inputs to F, only 13 of them are found to be feasible as shown in the distance tree in Figure 4.1. In the figure, the values in the leaf nodes shown in boxes below denote the output of F when that particular combination of b_i's are chosen. Each leaf node also corresponds to a feasible input of F and is called a *path*, denoted with sequence of input bits. The value of a path is the output of F for that input. A path in the tree identifies the linear functions whose distances to the Boolean function in consideration are obtained by adding the same combined distance coefficients, also specifying the distance to these functions. As there can be more than one linear function covered by a path, different paths can have the same value. For example, there are six paths with value 3, four paths with value -1 and two paths with value -5 in the distance tree of the example function. Table 4.4 gives a more detailed information about this distance tree. The first column of the table denotes the leaf node (or path) number, the second column lists the path string, the third column gives the output of the distance function F for the corresponding path and the last column denotes which linear functions that path is associated with.

Table 4.5 shows a portion of the LDM of order 5, where only the eight columns related to the distance function of the example are shown. The distance tree constructed using Algorithm 4.3.1 and Algorithm 4.3.2 actually enumerates the distinct sums that can occur when the combined distance coefficients are added for each row of the LDM. In Table 4.5, the top most row denotes the combined coefficients C_i of the example function and the row below denotes

Table 4.4: Distance tree data of $F(b_1, \cdots, b_8) = 8b_1 + 8b_2 + 4b_3 + 4b_4 - 8b_5 - 4b_6 - 4b_7 + 3b_8$.

N	Path	F	Associated l_w
1	11001101	7	x_5
2	10101011	3	x_4
3	10001001	3	$x_4 \oplus x_5$
4	01111111	3	x_1
5	01001101	-1	$x_1 \oplus x_5$
6	00110111	3	x_2
			$x_1 \oplus x_2$
7	00101011	-5	$x_1 \oplus x_4$
8	00100011	3	$x_2 \oplus x_4$
			$x_1 \oplus x_2 \oplus x_4$
9	00010111	-1	x_3
			$x_1 \oplus x_3$
			$x_2 \oplus x_3$
			$x_1 \oplus x_2 \oplus x_3$
10	00001001	-5	$x_1 \oplus x_4 \oplus x_5$
11	00000101	-1	$x_2 \oplus x_5$
			$x_1 \oplus x_2 \oplus x_5$
			$x_3 \oplus x_5$
			$x_1 \oplus x_3 \oplus x_5$
			$x_2 \oplus x_3 \oplus x_5$
			$x_1 \oplus x_2 \oplus x_3 \oplus x_5$
12	00000011	-1	$x_3 \oplus x_4$
			$x_1 \oplus x_3 \oplus x_4$
			$x_2 \oplus x_3 \oplus x_4$
			$x_1 \oplus x_2 \oplus x_3 \oplus x_4$
13	00000001	3	$x_2 \oplus x_4 \oplus x_5$
			$x_1 \oplus x_2 \oplus x_4 \oplus x_5$
			$x_3 \oplus x_4 \oplus x_5$
			$x_1 \oplus x_3 \oplus x_4 \oplus x_5$
			$x_2 \oplus x_3 \oplus x_4 \oplus x_5$
			$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$

Table 4.5: Distance coefficients and related portion of the LDM for $f(x_1, \cdots, x_5) = x_1x_5 \oplus x_4x_5 \oplus x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_2x_3x_4x_5$.

C_i	1	1	1	1	-2	-2	-2	3	F	W	N
β_i	**8**	**8**	**4**	**4**	**-8**	**-4**	**-4**	**3**			
	a_3	a_{17}	a_{26}	a_{28}	a_{19}	a_{29}	a_{30}	a_{31}			
l_0	8	8	4	4	4	2	2	1	11	10	-
l_1	-8	-8	0	0	-4	-2	0	-1	7	14	1
l_2	-8	0	-4	0	-4	0	-2	-1	3	6	2
l_3	8	0	0	0	4	0	0	1	3	-6	3
l_4	0	0	0	-4	0	-2	-2	-1	-1	-2	9
l_5	0	0	0	0	0	2	0	1	-1	2	11
l_6	0	0	0	0	0	0	2	1	-1	2	12
l_7	0	0	0	0	0	0	0	-1	3	6	13
l_8	0	0	-4	-4	0	-2	-2	-1	3	6	6
l_9	0	0	0	0	0	2	0	1	-1	2	11
l_{10}	0	0	4	0	0	0	2	1	3	-6	8
l_{11}	0	0	0	0	0	0	0	-1	3	6	13
l_{12}	0	0	0	4	0	2	2	1	-1	2	9
l_{13}	0	0	0	0	0	-2	0	-1	-1	-2	11
l_{14}	0	0	0	0	0	0	-2	-1	-1	-2	12
l_{15}	0	0	0	0	0	0	0	1	3	-6	13
l_{16}	0	-8	-4	-4	-4	-2	-2	-1	3	6	4
l_{17}	0	8	0	0	4	2	0	1	-1	2	5
l_{18}	0	0	4	0	4	0	2	1	-5	10	7
l_{19}	0	0	0	0	-4	0	0	-1	-5	-10	10
l_{20}	0	0	0	4	0	2	2	1	-1	2	9
l_{21}	0	0	0	0	0	-2	0	-1	-1	-2	11
l_{22}	0	0	0	0	0	0	-2	-1	-1	-2	12
l_{23}	0	0	0	0	0	0	0	1	3	-6	13
l_{24}	0	0	4	4	0	2	2	1	3	-6	6
l_{25}	0	0	0	0	0	-2	0	-1	-1	-2	11
l_{26}	0	0	-4	0	0	0	-2	-1	3	6	8
l_{27}	0	0	0	0	0	0	0	1	3	-6	13
l_{28}	0	0	0	-4	0	-2	-2	-1	-1	-2	9
l_{29}	0	0	0	0	0	2	0	1	-1	2	11
l_{30}	0	0	0	0	0	0	2	1	-1	2	12
l_{31}	0	0	0	0	0	0	0	-1	3	6	13

the combined distance coefficients β_i, obtained by multiplying the C_i's with the representative distance coefficients (the positive value appearing in the associated column). The values in this row constitute the coefficients of the function F whose absolute maximum value is to be searched. The right most three columns of the table list the output values of the distance function F, the Walsh coefficients of f and which leaf node in the distance tree this row belongs, among the nodes listed in Table 4.4. Note that in Table 4.5, for the rows l_i with $wt(\bar{i})$ being odd, that is, the rows corresponding to the linear functions with odd number of terms, the value of F outputs the negative of what must actually be computed. This is because all values in the LDM are taken to be positive as a consequence of Proposition 4.2.12. The values listed in column F of Table 4.5 correspond to $-\frac{W_f(\bar{i})}{2}$ for rows l_i if $wt(\bar{i})$ is even, and $\frac{W_f(\bar{i})}{2}$ if $wt(\bar{i})$ is odd. This means that the absolute maximum value of the values produced in the distance tree can be used to find out the nonlinearity. The maximum value appearing in the distance tree is 7 which appears in the left most leaf node. Therefore, the nonlinearity is $2^{n-1} - \frac{1}{2}\max_{w\in\mathbb{F}_2^n}|W_f(w)|$, which is $16 - 7 = 9$. It must also be noted that the first row of the LDM which is used to compute the weight is not taken into account here. Since the weight of the Boolean function can obtained by adding the combined distance coefficients produced in the first phase of the nonlinearity computation algorithm, it is sufficient to check whether this value is smaller than the nonlinearity value obtained in the second phase or not. The weight of the example function is $F(1,\cdots,1) = 11$, which is larger than 9, so the nonlinearity is found to be 9. A better use of the weight computed in the first phase is to set it as the best solution and make use of the optimization techniques in the second phase for solving the integer programming problem more efficiently.

4.3.1 Branch and Bound Method

Branch and bound method is a way of efficiently solving integer programming problems by early terminating the processing of nodes if the best solution that can be obtained from a node will not attain the maximum/minimum value whatever the subsequent variable choices be [8]. This idea could be realized in this specific problem by computing one maximum and one minimum value for each variable in the optimization problem. Since the variables are processed one by one, this allows one to determine what the maximum and minimum change in the function could be, regardless of whether the input is feasible or not. The algorithm then can choose not to branch a node if it is guaranteed that neither of the solutions obtained from that node will be better than the best feasible solution already obtained. For the distance function described in (4.20) having k input bits, the maximum amount of change (both positive and negative) for each node can be computed as follows:

$$max_i = \sum_{i\le x\le k, \beta_x>0} \beta_x \tag{4.21}$$

$$min_i = \sum_{i\le x\le k, \beta_x<0} \beta_x. \tag{4.22}$$

Values max_i and min_i specify how much the function can increase and decrease at most, once the first $i-1$ variables are fixed. For instance, the function can increase the most if all of the subsequent coefficients with $\beta_x > 0$ are added and $\beta_x < 0$ are omitted. Hence, at a particular node while constructing the distance tree, by using the max_i and min_i values, it can be checked

whether the processed node can yield a larger absolute value than the best one at hand. If not, the processing of that branch of the tree is terminated at that point. If the node promises to attain a better solution, the branching continues. However, this does not guarantee that a better solution will be obtained since an increase (resp. decrease) of max_i (resp. min_i) might not be possible if these inputs are not feasible.

4.3.2 Recovering the Nearest Affine Function

Besides computing the nonlinearity, it could be as much important to identify which affine function(s) a Boolean function is closest to. This can be accomplished by using the path string of the nonlinearity algorithm described above. The path string identifies the rows in a LDM by specifying certain columns containing either zero or nonzero entries. This problem can be rephrased as follows:

Given n and two sets of indices $I = \{i_1, \ldots, i_k\}$, $E = \{e_1, \ldots, e_l\}$ where the indices are from the set $\{0, \ldots, 2^n - 1\}$. Identify the rows r in M^n satisfying $M^n_{r,j} \neq 0$ for $j \in I$ and $M^n_{r,j} = 0$ for $j \in E$.

Algorithm 4.3.3 outputs a list of linear functions identified by the vector $x \in \mathbb{F}_2^n$, given the column index sets I and E. The algorithm makes use of Remark 4.2.7 and 4.2.11 to list the rows of the LDM satisfying the given conditions. This algorithm can be executed for each leaf node of the distance tree to find out the linear functions at a specified distance to the Boolean function in question. Linear functions listed in the last column of Table 4.4 are found by executing Algorithm 4.3.3 with the column index sets I and E constructed from the corresponding path strings. For example, in the first row of the table, the path string is 11001101, which makes $I = \{3, 17, 19, 29, 31\}$ and $E = \{26, 28, 30\}$. There is only one row in the LDM whose column indices specified in I are nonzero and column indices specified in E are zero, and that row corresponds to the linear function $l_1 = x_5$. The decision of whether the Boolean function is closer to a linear function or its complement can be made based on the sign of the Walsh coefficient.

Algorithm 4.3.3: PathToLinearFunction(n, I, E)

$IncludeMask \leftarrow \bigwedge_{i \in I} \bar{i}$
for each $x \in \underline{S}(IncludeMask)$
{
  $valid \leftarrow true$
  for each $y \in E$
  if $(\neg\bar{y} \wedge x) = \bar{0}$
    then { $valid \leftarrow$ **false**; *break* }
  if $valid =$ **true**
    then *print* x
}

4.3.3 Complexity of the Algorithm and Experimental Results

The nonlinearity computation algorithm consists of two phases each having different complexities. In the first phase, from a given set of ANF coefficients, the combined coefficients are calculated. The number of combined coefficients is equal to the number of distinct products of input monomials. Although this value can be as high as 2^p (p being the number of monomials) where each monomial combination is distinct, the actual value will vary depending on the structural relations between monomials. The expected value of this quantity is given in (3.7) in terms of n and p as

$$\sum_{k=1}^{p} (1 - (1 - (1 - q)^k)^n) \binom{n}{k}$$

where q denotes the probability of a variable appearing in a monomial. For randomly generated Boolean functions q is $\frac{1}{2}$.

For the second phase of the algorithm where the binary integer programming problem is solved, it is harder to give an explicit expression of the complexity. The difficulty of estimating the complexity is a result of the fact that it depends on the distribution of the values in the Walsh spectrum of the Boolean function.

The proposed nonlinearity computation algorithm is implemented in C language and execution times are measured for different parameters on a PC having an Intel Core2 Duo processor running at 3.0GHz. Table 4.6 gives the average running times for 60-variable Boolean functions with branch and bound method being employed. In the table, p denotes the number of monomials, k is the average number of combined distance coefficients, i.e., the number of variables of the associated integer programming problem. Next two columns denote the average running times of the first phase (calculating combined coefficients) and the total running time of the algorithm. For each number of monomials in the experiments, average timings were calculated over 10 randomly generated Boolean functions. Timings in Table 4.6 indicate that for this type of Boolean functions, the complexity of the first and second phases of the nonlinearity computation algorithm are close.

Table 4.6: Timings for $n = 60$.

p	k	Phase 1 (sec.)	Total (sec.)
30	20815	1	2
40	54842	11	16
50	123015	60	90
60	246970	373	768
70	369198	1080	2325
80	555714	2876	6341
90	909078	8392	18708
100	1189792	15615	34092

4.4 Conclusion

In this chapter, an algorithm for computing the nonlinearity of a Boolean function from its ANF coefficients is proposed. The algorithm makes use of the formulation of the distance of a Boolean function to the set of linear functions. It is shown that the problem of computing the nonlinearity corresponds to a binary integer programming problem where techniques for efficiently solving these problems such as branch and bound method can be applied to improve the performance. The algorithm allows the computation of nonlinearity for Boolean functions acting on large number of inputs where applying the Fast Walsh transform is impractical.

CHAPTER 5

CONCLUSION

Boolean functions defined on large number of inputs introduces the problem of computing the cryptographic properties of them. Computational complexities of traditional algorithms for computing these properties are so high that they become impractical. Algebraic normal form (ANF) provides a compact representation for Boolean functions if the number of input variables is high. In this thesis, methods of computing the weight and nonlinearity of Boolean functions from the ANF representation have been studied.

The expression of the weight of a Boolean function in terms of its ANF coefficients was introduced by Carlet and Guillot, which allowed one to compute the weight of a Boolean function consisting of p monomials in $O(2^p)$ operations. By eliminating the unnecessary calculations in this expression, a more efficient algorithm that is explained in Chapter 3 of this thesis is obtained.

In Chapter 4, computation of the nonlinearity of a Boolean function in terms of its ANF coefficients is investigated. Generalizing the weight expression described in Chapter 3, a formulation of the distances between a Boolean function and the set of linear functions is obtained. The Linear Distance Matrix consisting of the coefficients which are used to calculate these distances is defined. By exploiting the special structure of this matrix, the task of computing nonlinearity is reduced to solving an associated binary integer programming problem. The efficiency of solving this problem can be improved with the branch and bound method. The proposed nonlinearity computation algorithm can be used in cases where applying the Fast Walsh transform is impractical, typically when the number of input variables exceeds 40. On the other hand, the computational complexity of the algorithm enforces the number of monomials in the ANF representation being relatively small.

The extension of the nonlinearity computation algorithm to compute the higher order nonlinearities is an open problem. The most promising approach would be to extend the Linear Distance Matrix to include the distance coefficients for the quadratic functions and then solve the resulting integer programming problem. However, it is evident that the complexity of this extended version of the algorithm will be much higher than the original one because of the variety of elements in the matrix. The integer programming problem this time will not only involve binary coefficients, but also the coefficients that can take on three or more values. Nevertheless, there might be classes of Boolean functions where computing the second order nonlinearity becomes feasible if such an extension of the algorithm could be realized, that would otherwise be infeasible.

REFERENCES

[1] Biryukov, A., *A New 128-bit Key Stream Cipher LEX*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/012, 2005. http://www.ecrypt.eu.org/stream/lexp2.html

[2] Carlet, C., *Boolean Functions for Cryptography and Error Correcting Codes*, In Yves Crama and Peter L. Hammer (eds.) Boolean Models and Methods in Mathematics, Computer Science, and Engineering, pages 257–397, Cambridge University Press, 2010.

[3] Carlet, C., *On The Coset Weight Divisibility and Nonlinearity of Resilient and Correlation-immune Functions*, In T. Helleseth, P.V. Kumar, and K. Yang, editors, Sequences and their Applications, Discrete Mathematics and Theoretical Computer Science, pages 131-144. Springer London, 2002.

[4] Carlet, C., Dalai, D.K., Gupta, K.C., Maitra, S., *Algebraic Immunity for Cryptographically Significant Boolean functions: Analysis and construction*, Information Theory, IEEE Transactions on, 52(7):3105-3121, july 2006.

[5] Carlet, C., Guillot, P., *A New Representation of Boolean Functions*, In Marc Fossorier, Hideki Imai, Shu Lin, and Alain Poli, editors, Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, volume 1719 of Lecture Notes in Computer Science, pages 94-103. Springer Berlin Heidelberg, 1999.

[6] Carlet, C., Guillot, P., *Bent, Resilient Functions and The Numerical Normal Form*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 56, American Mathematical Society, Providence, RI, pages 87-96, 2001.

[7] Chen, K., Henricksen, M., Millan, W., Fuller, J., Simpson, L., Dawson, E., Lee, H., Moon, S., *Dragon: A Fast Word Based Stream Cipher*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/006, 2005. http://www.ecrypt.eu.org/stream/dragonp3.html

[8] Chinneck, C.W., *Practical Optimization: A Gentle Introduction*, online textbook, http://www.sce.carleton.ca/faculty/chinneck/po.html.

[9] Courtois, N.T., Meier, W., *Algebraic Attacks on Stream Ciphers with Linear Feedback*, In Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques, EUROCRYPT'03, pages 345-359, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] Crama, Y., Hammer, P.L., *Boolean Functions, Theory, Algorithms, and Applications*, Cambridge University Press, 2010.

[11] Cusick, T.W., Stănică, P., *Cryptographic Boolean Functions and Applications*, Academic Press, San Diego, 2009.

[12] Çalık, Ç., Doğanaksoy, A., *Computing the Weight of a Boolean Function from its Algebraic Normal Form*, In Tor Helleseth and Jonathan Jedwab, editors, Sequences and Their Applications SETA 2012, volume 7280 of Lecture Notes in Computer Science, pages 89-100. Springer Berlin Heidelberg, 2012.

[13] Daemen, J., Rijmen, V., *The Design of Rijndael: AES - The Advanced Encryption Standard*, Springer, 2002.

[14] Dillon, J.F., *Elementary Hadamard Difference Sets*, Proceedings of the Sixth South-Eastern Conference on Combinatorics, Graph Theory and Computing, Boca Raton, Florida, Congressus Numerantium No. XIV, Utilitas Math., Winnipeg, Manitoba, 1975, pages 237-249.

[15] Ding, C., Xiao, G., Shan, W., *The Stability Theory of Stream Ciphers*, In Lectures in Computer Science, volume 561, Springer-Verlag, Berlin, 1991.

[16] Dobbertin, H., *Construction of Bent Functions and Balanced Boolean Functions with High Nonlinearity*, In Bart Preneel, editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pages 61-74. Springer Berlin Heidelberg, 1995.

[17] Dobbertin, H., Leander, G., *Cryptographer's Toolkit for Construction of 8-bit Bent Functions*, IACR Cryptology ePrint Archive, 2005:89, 2005. http://eprint.iacr.org/2005/089.

[18] Filiol, E., *Designs, Intersecting Families, and Weight of Boolean Functions*, In Michael Walker, editor, Cryptography and Coding, volume 1746 of Lecture Notes in Computer Science, pages 70-80. Springer Berlin Heidelberg, 1999.

[19] Göloğlu. F., *Divisibility Results on Boolean Functions Using the Numerical Normal Form*, M.Sc. Thesis, Institute of Applied Mathematics, Middle East Technical University, 2004.

[20] Gupta, K.C., Sarkar, P., *Computing Partial Walsh Transform From the Algebraic Normal Form of a Boolean Function*, Information Theory, IEEE Transactions on, 55(3):1354-1359, march 2009.

[21] Hawkes, P., Paddon, M., Rose, G.G., de Vries, M.W., *Primitive Specification for NLSv2*, eSTREAM, ECRYPT Stream Cipher Project, Report 2006/036, 2006. http://www.ecrypt.eu.org/stream/nlsp3.html

[22] Hell, M., Johansson, T., Meier, W., *Grain - A Stream Cipher for Constrained Environments*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005. http://www.ecrypt.eu.org/stream/grainp3.html

[23] Hell, M., Johansson, T., Maximov, A., Meier, W., *A Stream Cipher Proposal: Grain-128*, In Information Theory, 2006 IEEE International Symposium on, pages 1614-1618, july 2006.

[24] Jansen, C.J.A., Helleseth, T., Kholosha, A., *Cascade Jump Controlled Sequence Generator and Pomaranch Stream Cipher*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/022, 2005. http://www.ecrypt.eu.org/stream/pomaranchp3.html

[25] Langevin, P., Leander, G., Rabizzoni, P., Véron, P., Zanotti, J-P., *The Number of Bent Functions with 8 Variables*, In Jean-Francis Michon, Pierre Valarcher, Jean-Baptiste Yunès, editors, Proceedings of First International Workshop BFCA '05, Boolean Functions: Cryptography and Applications, pages 125-135, 2006.

[26] MacWilliams, F.J., Sloane, N.J.A., *The Theory of Error Correcting Codes*, North-Holland, 1977.

[27] Matsui, M., *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.

[28] McFarland, R.L., *A Family of Difference Sets in Non-cyclic Groups*, Journal of Combinatorial Theory, Series A, 15(1):1-10, 1973.

[29] Meier, W., Pasalic, E., Carlet, C. *Algebraic Attacks and Decomposition of Boolean Functions*, In Christian Cachin and Jan L. Camenisch, editors, Advances in Cryptology - EUROCRYPT 2004, volume 3027 of Lecture Notes in Computer Science, pages 474-491. Springer Berlin Heidelberg, 2004.

[30] Meier, W., Staffelbach, O., *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.

[31] Preneel B., *Analysis and Design of Cryptographic Hash Functions* Ph.D. thesis. Katholieke Universiteit Leuven, Belgium, 1993.

[32] Preneel, B., Govaerts, R., Vandewalle, J., *Boolean Functions Satisfying Higher Order Propagation Criteria*, In Proceedings of the 10th annual international conference on Theory and application of cryptographic techniques, EUROCRYPT '91, pages 141-152, Berlin, Heidelberg, 1991. Springer-Verlag.

[33] Preneel, B., Leekwijck, W.V., Linden, L.V., Govaerts, R., Vandewalle, J., *Propagation Characteristics of Boolean Functions*, In Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology, EUROCRYPT '90, pages 161-173, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[34] Rothaus, O.S., *On Bent functions*, Journal of Combinatorial Theory, Series A, 20(3):300-305, 1976.

[35] Shannon, C.E., *Communication Theory of Secrecy Systems*, Bell System Technical Journal, Vol. 28, No. 4. (1949), pages 656-715.

[36] Siegenthaler, T., *Correlation Immunity of Nonlinear Combining Functions for Cryptographic Applications*, Information Theory, IEEE Transactions on, 30(5):776-780, sep. 1984.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Çalık, Çağdaş
Nationality: Turkish
Date and Place of Birth: 1978, Bursa
e-mail: ccalik@metu.edu.tr
Phone: +90 312 2107767
fax: +90 312 2102985

EDUCATION

Degree	Institution	Year of Graduation
M.S.	Department of Cryptography, METU	2007
B.S.	Department of Mathematics, METU	2004

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
Jan 2008-	Institute of Applied Mathematics, METU	Research Assistant

PUBLICATIONS

Ç. Çalık, A. Doğanaksoy, *Computing the Weight of a Boolean Function from its Algebraic Normal Form*, In Tor Helleseth and Jonathan Jedwab, editors, Sequences and Their Applications SETA 2012, volume 7280 of Lecture Notes in Computer Science, pages 89-100. Springer Berlin Heidelberg, 2012.

Ç. Çalık, *An Efficient Software Implementation of Fugue*, Second SHA-3 Candidate Conference, Santa Barbara, California, USA, 23-24 August 2010.

Ç. Çalık, M. S. Turan, *Message Recovery and Pseudo-Preimage Attacks on the Compression Function of Hamsi-256*, In Michel Abdalla and Paulo S.L.M. Barreto, editors, Progress in Cryptology LATINCRYPT 2010, volume 6212 of Lecture Notes in Computer Science, pages 205-221. Springer Berlin Heidelberg, 2010.

Ç. Çalık, M. S. Turan, F. Özbudak, *On Feedback Functions of Maximum Length Nonlinear*

Feedback Shift Registers, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences Vol. E93-A, No.6, ages 1226-1231, Jun. 2010.

J-P. Aumasson, Ç. Çalık, W. Meier, O. Özen, R. C.-W. Phan, K. Varıcı, *Improved Cryptanalysis of Skein*, In Mitsuru Matsui, editor, Advances in Cryptology ASIACRYPT 2009, volume 5912 of Lecture Notes in Computer Science, pages 542-559. Springer Berlin Heidelberg, 2009.

M. S. Turan, Ç. Çalık, N.B. Saran, A. Doğanaksoy, *New Distinguishers Based on Random Mappings Against Stream Ciphers*, In Solomon W. Golomb, Matthew G. Parker, Alexander Pott, and Arne Winterhof, editors, SETA, volume 5203 of Lecture Notes in Computer Science, pages 30-41. Springer, 2008.

A. Doğanaksoy, Ç. Çalık, F. Sulak, *Observations on Hellmans Cryptanalytic Time-Memory Trade-off*, 2. Ulusal Kriptoloji Sempozyumu, Ankara, 2006.

M. S. Turan, A. Doğanaksoy, Ç. Çalık, *Detailed Statistical Analysis of Synchronous Stream Ciphers*, 2. Ulusal Kriptoloji Sempozyumu, Ankara, 2006.

A. Doğanaksoy, Ç. Çalık, F. Sulak, M.S. Turan, *New Randomness Tests Using Random Walk*, 2. Ulusal Kriptoloji Sempozyumu, Ankara, 2006.

A. Doğanaksoy, Ç. Çalık, F. Sulak, M. Sönmez Turan, *Rassal Gezinti Testi*, IGS06 İstatistik Gunleri Sempozyumu, Antalya, 2006.

M. S. Turan, A. Doğanaksoy, Ç. Çalık, *Statistical Analysis of Synchronous Stream Ciphers*, SASC '06, The State of the Art of Stream Ciphers, 2006, Leuven, Belgium.