Computing Partial Walsh Transform from the Algebraic Normal Form of a Boolean Function

Kishan Chand Gupta
Centre for Applied Cryptographic Research,
Department of Combinatorics and Optimization,
University of Waterloo,
200 University Avenue West, Waterloo,
Ontario, Canada N2L 3G1.
E-mail: kgupta@math.uwaterloo.ca

Palash Sarkar
Applied Statistics Unit,
Indian Statistical Institute,
203, B T Road, Kolkata 700 108,
India.
E-mail: palash@isical.ac.in

Abstract

We study the relationship between the Walsh transform and the algebraic normal form (ANF) of a Boolean function. In the first part of the paper, we obtain a formula for the Walsh transform at a certain point in terms of parameters derived from the algebraic normal form. We use previous results by Carlet and Guillot to obtain an explicit expression for the Walsh transform at a point in terms of parameters derived from the ANF. The second part of the paper is devoted to simplify this formula and develop an algorithm to evaluate it. Our algorithm can be applied in situations where it is practically impossible to use the fast Walsh transform algorithm. Experimental results show that under certain conditions it is possible to execute our algorithm to evaluate the Walsh transform (at a small set of points) of functions on a few scores of variables having a few hundred terms in the algebraic normal form.

Keywords : Boolean function, Algebraic Normal Form, Walsh Transform.

1 Introduction

Boolean functions are used for a wide variety of applications in engineering and computer science. An m-variable Boolean function $g(x)$ is a map $g : \{0,1\}^m \rightarrow \{0,1\}$. One of the most useful tools for the study of Boolean functions is the Walsh transform, which is essentially the Fourier transform applied to the function $(-1)^{g(x)}$. The Walsh transform measures the correlations between an m-variable Boolean function and all the m-variable linear functions. These correlations uniquely determine the function and hence it is possible to work entirely with the Walsh transform. In fact, many properties of Boolean functions are most easily stated in terms of Walsh transform.

For practical applications it is often useful to be able to compute the Walsh transform of a Boolean function. It turns out that there is an excellent algorithm to do so, namely, the fast Walsh transform [1]. For an m-variable function the fast Walsh transform takes time $O(m2^m)$ and hence can be used for functions of around 30 variables. The fast Walsh transform is computed from the description of the function itself. More precisely, the fast Walsh transform takes as input the bit string $f(\sigma_0) \dots f(\sigma_{2^m-1})$ of length 2^m, where for $0 \leq i \leq 2^m - 1$, σ_i is the m-bit binary representation of i.

There is another way to uniquely represent a Boolean function, namely by its algebraic normal form, which expresses a Boolean function as a multivariate polynomial over F_2, the finite field of two elements. The number of nonzero terms in the polynomial can be 2^m in the worst case. However, many interesting classes of Boolean functions do have compact algebraic normal form representation. (For example, an $m = 2k$-variable bent function can have as few as k many terms in their ANF.)

Carlet and Guillot [3] study an alternative representation of Boolean functions – the numerical normal form (NNF). In Theorem 1 of [3], they obtain a formula for computing the NNF from the ANF representation and in Equation (6) of [3], they obtain a formula for computing the Walsh transform from the NNF representation. Using these two results, it is possible to obtain a formula for computing the Walsh transform at a point in terms of parameters derived from the ANF representation.

Our main contribution in this paper is to obtain an algorithm for computing the Walsh transform from the above mentioned formula. Our algorithm can be used to compute the Walsh transform in cases where it is not possible to use the fast Walsh transform. For example, it is possible in certain cases to run our algorithm for 50 to 100 variable functions having a few hundred terms in their algebraic normal form. For such functions it is possible to compute the Walsh transform for a small set of points v. Note that it is practically impossible to compute the Walsh transform of an m-variable function at all points in F_2^m if m is around 50 or more. Our algorithm provides a method to probe the spectral domain of large variable functions. This will provide some useful information about the function such as the size of its support and an estimate of its nonlinearity. Note that for small number of variables, the fast Walsh transform is faster than our algorithm. Hence we do not provide a substitute for the fast Walsh transform; rather we provide a tool to analyse a Boolean function in situations where the fast Walsh transform cannot be used.

Boolean functions are studied extensively from different perspectives – coding theory [7, 6], circuit complexity [2, 8] and cryptography [4, 9] are some examples. In all these areas, the Walsh transform is the main tool in the analysis of Boolean functions. However, to the best of our knowledge, the only previously known algorithm for computing the Walsh transform is the fast Walsh transform [1]. Hopefully the present work will motivate researchers to study the algorithmic issues of the Walsh transform more deeply.

2 Preliminaries

Let $F_2 = GF(2)$. An m-variable Boolean function $g(x)$ is a map $g : F_2^m \to F_2$. We consider the domain of an m-variable Boolean function to be the vector space $(F_2^m, \oplus)$ over F_2, where $\oplus$ is used to denote the addition operator over both F_2 and the vector space F_2^m. The symbol $+$ is used to denote addition over integers. The inner product of two vectors $u, v \in F_2^m$ will be denoted by $\langle u, v\rangle$. The weight of an m-bit vector u is the number of ones in u and will be denoted by $wt(u)$. The *support* of a Boolean function g is denoted by $Sup(g)$ and is defined to be $Sup(g) = \{x : g(x) = 1\}$. The weight of g is denoted by $wt(g)$ and is defined to be $wt(g) = |Sup(g)|$. The *Walsh Transform* of a Boolean function $g(x)$ is an integer valued function $W_g : \{0,1\}^m \to [-2^m, 2^m]$ defined by (see [7, page 414])

$$W_g(u) = \sum_{w \in F_2^m} (-1)^{g(w)\oplus\langle u,w\rangle}. \qquad (1)$$

For even m, an m-variable function g is called *bent* if $W_g(u) = \pm 2^{\frac{m}{2}}$, for all $u \in F_2^m$. A parameter of fundamental importance in cryptography is the *nonlinearity* of a Boolean function. This quantity measures the distance of a Boolean function from the set of all affine functions. It is sometime convenient to express nonlinearity in terms of the spectrum of a Boolean function. The nonlinearity $nl(g)$ of an m-variable Boolean function g, can be written as

$$nl(g) = 2^{m-1} - \frac{1}{2}\max_{u \in F_2^m} |W_g(u)|.$$

An m-variable Boolean function $g(x_1, x_2, \cdots, x_m)$ can be uniquely written as

$$g(x_1, x_2, \ldots, x_m) = \bigoplus_{(\alpha_1,\alpha_2,\cdots,\alpha_m)\in F_2^m} A_g(\alpha_1, \alpha_2, \cdots, \alpha_m) x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_m^{\alpha_m} \tag{2}$$

where $A_g(x_1, x_2, \cdots, x_m)$ is a Boolean function. This representation is called the *algebraic normal form (ANF)* of g. We will call the Boolean function A_g the ANF of g.

To illustrate different definitions, terms and and notations we take a small example. Example : $g(x_1, x_2, x_3) = x_1x_2 \oplus x_2x_3 \oplus x_1 \oplus x_2$**, and** $Sup(A_g) = \{(1, 1, 0), (0, 1, 1), (1, 0, 0), (0, 1, 0)\}$**.**

3 Walsh Transform and ANF

The first task is to obtain an explicit formula for the Walsh transform from the ANF. This is done by combining the remark following Theorem 1 and Equation (6) of [3]. The following definition is from [3].

Definition 3.1 *Let g be a m variable Boolean function (and more generally of complex valued function on F_2^m). We call Numerical Normal Form of g, the following expression of g as a polynomial:*

$$g(x_1, \cdots, x_m) \quad = \quad \textstyle\sum_{u\in F_2^m} \lambda_u (\prod_{i=1}^{n} x_i^{u_i}) \quad = \quad \sum_{u\in F_2^m} \lambda_u x^u, \lambda_u \in \mathbf{C}. \tag{3}$$

Now we define certain terms. For $\alpha = (\alpha_1, \cdots, \alpha_m)$ and $u = (u_1, \cdots, u_m)$ we define $u \le \alpha$ if $u_i \le \alpha_i$ for $1 \le i \le m$. *In the rest of the paper we will denote* $(wt(u) \bmod 2)$ *by* $wt_2(u)$. Dirac function δ_v is defined as $\delta_v = 1$ if $v = 0$ else $\delta_v = 0$. Let $Sup(A_g) = \{\alpha^{(1)}, \ldots, \alpha^{(p)}\}$. Given $\emptyset \ne S \subseteq [p] = \{1, 2, \cdots, p\}$, let $S = \{i_1, \cdots, i_r\}$. Let

$$\left.\begin{array}{rcl} \sigma(S) & = & \bigvee_{i\in S} \alpha^{(i)} \quad = \quad \alpha^{(i_1)} \vee \cdots \vee \alpha^{(i_r)}); \\ \mu(S) & = & wt(\sigma(S)). \end{array}\right\} \tag{4}$$

Here $\vee$ represents the bitwise logical OR of two binary strings of the same length. The following two results are from [3].

Theorem 3.1 *[3, Theorem 1 (remark)] Let λ_u be the coefficient of Numerical Normal Form of the Boolean function g. Then*

$$\lambda_u \quad = \sum_{\emptyset\ne S\subseteq[p];\sigma(S)=u} (-2)^{|S|-1}. \tag{5}$$

Theorem 3.2 *[3, Equation 6] Let λ_u be the coefficient of Numerical Normal Form of m variable Boolean function g. Then*

$$W_g(v) \quad = \quad 2^m(\delta_v) + (-1)^{wt_2(v)+1} \sum_{u\in F_2^m : v\le u} 2^{m-wt_2(u)+1} \lambda_u. \tag{6}$$

We define certain terms which will be required in the proof of Theorem 3.3. Given $\emptyset \ne S \subseteq [p] = \{1, 2, \cdots, p\}$ and $v \in F_2^m$, we define $\Delta(S, v)$ as follows. Let $v = (v_1, \cdots, v_m)$ and $S = \{i_1, \cdots, i_r\}$. Let $\phi(S, v)$ be a Boolean formula which is false (0) if and only if $v \le \sigma(S)$. Define

$$\left.\begin{array}{rcll} \Delta(S, v) & = & 0 & \text{if } \phi(S, v) = 1 \\ & = & \frac{1}{2^{\mu(S)-|S|}} & \text{otherwise.} \end{array}\right\} \tag{7}$$

If $\Delta(S,v) > 0$, then the value of $\Delta(S,v)$ is independent of v and depends only on S. We define

$$C_v = (-1)^{wt_2(v)} \sum_{\emptyset \neq S \subseteq [p]} (-1)^{|S|} \Delta(S,v). \tag{8}$$

Recall that $\phi(S,v) = 0$ if and only if $v \leq \sigma(S)$. So if we look at Equation 7, $\Delta(S,v) = \frac{1}{2^{\mu(S)-|S|}}$ whenever $v \leq \sigma(S)$. Now it is not difficult to verify

$$\sum_{\emptyset \neq S \subseteq [p]} (-1)^{|S|} \Delta(S,v) = \sum_{u \in F_2^m : v \leq u} \sum_{\emptyset \neq S \subseteq [p]; \sigma(S) = u} \frac{(-1)^{|S|}}{2^{\mu(S)-|S|}}. \tag{9}$$

The following result relates Walsh transform to the ANF.

Theorem 3.3 *If g is an m-variable Boolean function then*

$$W_g(v) = 2^m (C_v + \delta_v) \tag{10}$$

where C_v is as defined by Equation 8 and $\delta_v = 1$ if $v = 0$ else $\delta_v = 0$.

Proof : From Equation 6 we have

$$W_g(v) = 2^m(\delta_v) + (-1)^{wt_2(v)+1} \sum_{u \in F_2^m : v \leq u} 2^{m - wt_2(u)+1} \lambda_u$$

Putting the value of λ_u from Equation 5 we have

$$W_g(v) - 2^m(\delta_v) = (-1)^{wt_2(v)+1} \sum_{u \in F_2^m : v \leq u} 2^{m - wt_2(u)+1} \sum_{\emptyset \neq S \subseteq [p]; \sigma(S) = u} (-2)^{|S|-1}.$$

Therefore

$$\begin{aligned}
\frac{W_g(v) - 2^m(\delta_v)}{2^m(-1)^{wt_2(v)}} &= \sum_{u \in F_2^m : v \leq u} \sum_{\emptyset \neq S \subseteq [p]; \sigma(S) = u} 2^{-wt_2(u)} (-2)^{|S|} \\
&= \sum_{u \in F_2^m : v \leq u} \sum_{\emptyset \neq S \subseteq [p]; \sigma(S) = u} \frac{(-1)^{|S|}}{2^{wt_2(u)-|S|}} \\
&= \sum_{u \in F_2^m : v \leq u} \sum_{\emptyset \neq S \subseteq [p]; \sigma(S) = u} \frac{(-1)^{|S|}}{2^{\mu(S)-|S|}} \text{ (See Equation 4)} \\
&= \sum_{\emptyset \neq S \subseteq [p]} (-1)^{|S|} \Delta(S,v) \text{ (See Equation 9)} \\
&= \frac{C_v}{(-1)^{wt_2(v)}} \text{ (See Equation 8).}
\end{aligned}$$

■

Theorem 3.3 provides a method for computing $W_g(v)$ in terms of C_v. However, to compute C_v requires an algorithm to consider all possible subsets of $[p] = \{1, \ldots, p\}$. If p is even moderately large, this yields an impractical algorithm. Our next task is to show that it is actually not required to consider all the subsets of $[p]$. In this section we present one such special case and in the next section we present a general algorithm to compute C_v without generating all the subsets of $[p]$.

Theorem 3.4 *Let $Sup(A_g) = \{\alpha_1, \cdots, \alpha_p\}$ and $t > 0$ be such that for any $\{i_1, \ldots, i_{t+1}\} \subseteq \{1, \ldots, p\}$, we have $wt(\alpha_{i_1} \vee \cdots \vee \alpha_{i_{t+1}}) = m$. Then*

$$C_v = (-1)^{wt_2(v)} \left(\sum_{j=t+1}^{p} \frac{(-1)^j}{2^{m-j}} \binom{p}{j} + \sum_{\emptyset \neq S \subseteq [p], |S| \leq t} (-1)^{|S|} \triangle (S, v) \right)$$

Proof : From Equation (8) we have

$$\begin{aligned} C_v &= (-1)^{wt_2(v)} \sum_{\emptyset \neq S \subseteq [p]} (-1)^{|S|} \triangle (S, v) \\ &= (-1)^{wt_2(v)} \left(\sum_{\emptyset \neq S \subseteq [p], |S| \geq t+1} (-1)^{|S|} \triangle (S, v) + \sum_{\emptyset \neq S \subseteq [p], |S| \leq t} (-1)^{|S|} \triangle (S, v) \right). \end{aligned}$$

Under the given condition if $|S| \geq t + 1$ then using Equation 7 we have $\mu(S) = m$. Hence

$$\begin{aligned} C_v &= (-1)^{wt_2(v)} \left(\sum_{\emptyset \neq S \subseteq [p], |S| \geq t+1} (-1)^{|S|} \frac{1}{2^{m-r}} + \sum_{\emptyset \neq S \subseteq [p], |S| \leq t} (-1)^{|S|} \triangle (S, v) \right) \\ &= (-1)^{wt_2(v)} \left(\sum_{j=t+1}^{p} (-1)^j \frac{1}{2^{m-j}} \binom{p}{j} + \sum_{\emptyset \neq S \subseteq [p], |S| \leq t} (-1)^{|S|} \triangle (S, v) \right) \end{aligned}$$

This completes the proof. ■

Under the condition of Theorem 3.4, to evaluate C_v we only have to consider all non empty subsets of $[p]$ of cardinality at most t. If t is reasonably small, this is much better than considering all non empty subsets of $[p]$. In the next section, we develop this idea to obtain an algorithm to compute C_v.

4 Algorithm

In general we are interested in computing the Walsh transform of g at all the points $u \in F_2^m$. However, if m is relatively large (say around 50), then it will be practically impossible to compute the Walsh transform at all the 2^m points. In such a situation, it will be of interest to compute the Walsh transform at a particular point or for a small set of points. The fast Walsh transform takes time $O(m2^m)$ and computes the Walsh transform at all the 2^m points. In fact, to the best of our knowledge, there is no known algorithm which can compute the Walsh transform at a particular point in time less than 2^m.

Our approach is to design an algorithm that consists of two parts. In the first part, the algorithm does a certain amount of preprocessing and prepares a list. In the second part, the algorithm takes as input a particular $v \in F_2^m$ and computes C_v. (Using Theorem 3.3 this also gives us $W_g(v)$). Once the preprocessing is complete, the second part can be run for different v without running the first part. This makes it efficient to compute $W_g(v)$ for a set of v.

We start by defining certain parameters. For $\emptyset \neq S \subseteq [p]$ recall from equation (4) that $\sigma(S) = \bigvee_{i \in S} \alpha^{(i)}$ and $\mu(S) = wt(\sigma(S))$. For $j = 0, \cdots, m$ define

$$B_j \stackrel{\triangle}{=} \sum_{\mu(S)=j} (-2)^{|S|} \text{ and } B \stackrel{\triangle}{=} \sum_{j=0}^{m-1} \sum_{\mu(S)=j} (-1)^{|S|} \Delta(S, v).$$

Note that if $S = \{i\}$ and $\alpha^{(i)} = (0, \ldots, 0)$, then $\mu(S) = 0$. The vector $(0, \ldots, 0) \in Sup(A_g)$ implies that the constant term in the ANF of g is equal to 1. The values of $B_0, \ldots, B_m$ are independent of v and only the value of B depends on v.

Theorem 4.1 *For any $v \in F_2^m$, we have*

$$C_v \;=\; (-1)^{wt_2(v)}\left[\frac{(-1)^p - 1 - B_0 - \cdots - B_{m-1}}{2^m} + B\right]. \tag{11}$$

Proof : Define

$$A \stackrel{\triangle}{=} \sum_{\emptyset \neq S \subseteq [p]} (-2)^{|S|} = \sum_{j=1}^{p} (-2)^j \binom{p}{j} = -1 + (-1)^p.$$

Then $A = \sum_{j=0}^{m} B_j$. Consequently, $B_m = A - B_0 - \cdots - B_{m-1}$. To see this note that $\sum_{j=0}^{m} B_j = \sum_{j=0}^{m} \sum_{\mu(S)=j} (-2)^{|S|} = \sum_{\emptyset \neq S \subseteq [p]} (-2)^{|S|} = A$.

From Equation (8) we have

$$\begin{aligned} C_v &= (-1)^{wt_2(v)} \sum_{\emptyset \neq S \subseteq [p]} (-1)^{|S|} \Delta(S, v) \\ &= (-1)^{wt_2(v)} \left[\sum_{j=0}^{m} \sum_{\mu(S)=j} (-1)^{|S|} \Delta(S, v)\right] \\ &= (-1)^{wt_2(v)} \left[\sum_{\mu(S)=m} (-1)^{|S|} \Delta(S, v) + \sum_{j=0}^{m-1} \sum_{\mu(S)=j} (-1)^{|S|} \Delta(S, v)\right] \\ &= (-1)^{wt_2(v)} \left[\frac{B_m}{2^m} + B\right] \\ &= (-1)^{wt_2(v)} \left[\frac{A - B_0 - \cdots - B_{m-1}}{2^m} + B\right] \end{aligned}$$

This completes the proof. ∎

Our algorithm is based on Equation (11). The intuition behind the algorithm is the following. For most sets S, the value of $\mu(S)$ will be equal to m and will be accounted for by B_m. Thus if we can avoid computing B_m directly, then we will be saving a lot of computation. We will compute $B_0, \ldots, B_{m-1}$ and then use Equation (11) to compute the value of C_v. However, the value of B has to be computed. We next describe how this is done.

Define

$$\left.\begin{aligned} \mathcal{S}_1 &= \{S \subseteq [p] : \mu(S) < m\}; \\ \mathcal{S}_2 &= \sigma(\mathcal{S}_1) = \{\sigma(S) : S \in \mathcal{S}_1\}. \end{aligned}\right\} \tag{12}$$

Example (continued) Again we consider the Boolean function $g(x_1, x_2, x_3) = x_1x_2 \oplus x_2x_3 \oplus x_1 \oplus x_2$, so $Sup(A_g) = \{(1,1,0), (0,1,1), (1,0,0), (0,1,0)\}$. Here $p = 4$, $m = 3$. It is easy to see
$\mathcal{S}_1 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{1,3\}, \{1,4\}, \{2,4\}, \{3,4\}, \{1,3,4\}\}$,
$\mathcal{S}_2 = \{\sigma(\{1\}) = (1,1,0), \sigma(\{2\}) = (0,1,1), \sigma(\{3\}) = (1,0,0), \sigma(\{4\}) = (0,1,0)\}$ and hence $|\mathcal{S}_1| = 9$ and $|\mathcal{S}_2| = 4$. Note by definition of σ, $\sigma(\{1,3\}) = (1,1,0) \vee (1,0,0) = (1,1,0) = \sigma(\{1\})$. And this is the reason for $|\mathcal{S}_2| \leq |\mathcal{S}_1|$.

Write $\mathcal{S}_2 = \{str_1, \ldots, str_n\}$ for some $n > 0$. For $1 \leq i \leq n$ define $val_i = \sum_{\sigma(S)=str_i} (-2)^{|S|}$. In the first part of our algorithm we prepare the list $\mathcal{L} = ((str_1, val_1), \ldots, (str_n, val_n))$. Note that this part does not depend on v.

Example (continued) $\mathcal{S}_2 = \{str_1 = (1,1,0), str_2 = (0,1,1), str_3 = (1,0,0), str_4 = (0,1,0)\}$. Now $val_1 = \sum_{\sigma(S)=str_1} (-2)^{|S|} = (-2)^1 + (-2)^2 + (-2)^2 + (-2)^2 + (-2)^3 = -2$. Similarly $val_2 = 2$, $val_3 = -2$ and $val_4 = -2$

For $str, v \in F_2^m$, define $\psi(str, v) \stackrel{\triangle}{=} \overline{str} \wedge v$ where $\overline{str}$ is bitwise complement of str and $\wedge$ is the bitwise logical AND. The operation $\wedge$ is performed bitwise on str and v. Suppose $\emptyset \neq S \subseteq [p]$ such that $\sigma(S) = str$ and $v \in F_2^m$. Then $\Delta(S, v) > 0$ if and only if $\psi(str, v) = (0, \ldots, 0)$. Thus

$$B = \sum_{\psi(str_i, v)=(0,\ldots,0)} \frac{val_i}{2^{wt(str_i)}}.$$

Also for $j = 0, \ldots, m-1$,

$$B_j = \sum_{wt(str_i)=j} val_i.$$

Hence once the list $\mathcal{L}$ is prepared, it is easy to compute $B_0, \ldots, B_{m-1}$ and B. Now we describe a method for preparing the list $\mathcal{L}$.

Example (continued) We take $v = (1, 0, 0)$, then $\psi((1, 1, 0), (1, 0, 0)) = (0, 0, 1) \wedge (1, 0, 0) = (0, 0, 0)$. Similarly $\psi((0, 1, 1), (1, 0, 0)) = (1, 0, 0)$, $\psi((1, 0, 0), (1, 0, 0)) = (0, 0, 0)$ and $\psi((0, 1, 0), (1, 0, 0)) = (1, 0, 0)$. So $B = \frac{val_1}{2^{wt(str_1)}} + \frac{val_3}{2^{wt(str_3)}} = \frac{2}{4} + \frac{-2}{2} = -\frac{1}{2}$ Again from definition $B_0 = 0, B_1 = val_3 + val_4 = -2 + (-2) = -4, B_2 = val_1 + val_2 = 2 + 2 = 4$. Now $A = -1 + (-1)^p = 0$. So $C_v = (-1)^{wt_2(v)} \left[\frac{A - B_0 - B_1 - B_2}{2^m} + B\right] = -[\frac{0-0+4-4}{2^3} + \frac{1}{2}] = -\frac{1}{2}$.

First we describe a rooted directed tree $\mathcal{T}$ whose nodes are the subsets of $[p] = \{1, \ldots, p\}$. The root node of $\mathcal{T}$ is the empty set. The children of a set S are the sets $S_1, \ldots, S_k$, where for $1 \leq i \leq k$, $S_i = S \cup \{\max(S) + i\}$ and $k = p - \max(S)$. This ensures that if S' is a node in the subtree rooted at S, then $S \subset S'$.

Our algorithm will traverse all the nodes S of $\mathcal{T}$ for which $\mu(S) < m$. (Note that $\mathcal{T}$ has 2^p nodes and if an algorithm is required to traverse all the nodes of $\mathcal{T}$, then the algorithm will be exponential in p.) From the structure of $\mathcal{T}$ we know that if $\mu(S) = m$ for some node S, then $\mu(S') = m$ for all nodes S' in the subtree rooted at S. This crucial fact makes the traversal particularly efficient. If during the traversal we reach a node S with $\mu(S) = m$, then we need not visit any of the nodes in the subtree rooted at S. This means that we are effectively *pruning* the subtree rooted at S from $\mathcal{T}$. The more we encounter this pruning effect, the more efficient is our algorithm.

While traversing $\mathcal{T}$ we prepare the list $\mathcal{L}$ in the following manner. Initially $\mathcal{L}$ is the empty list. Let $\mathsf{First}(\mathcal{L}) = \{str : (str, val) \in \mathcal{L}\}$. Suppose we have reached a node S with $\mu(S) < m$. If $\sigma(S) \notin \mathsf{First}(\mathcal{L})$, then we add $(\sigma(S), (-2)^{|S|})$ to $\mathcal{L}$. On the other hand, if $\sigma(S) \in \mathsf{First}(\mathcal{L})$, then we update val to $val + (-2)^{|S|}$.

Thus the operations on $\mathcal{L}$ are search and insert. We implement $\mathcal{L}$ using a height balanced binary tree (see [5]). Hence each search/insert operation requires time $O(\log \mathcal{L})$. One such search or insert operation is required for each S such that $\mu(S) < m$. Also the total time spent at any node which is visited is $O(m)$. Hence the total time required by the algorithm is $O(m|\mathcal{S}_1| \log(|\mathcal{S}_2|))$. We next present the algorithm for computing C_v.

Algorithm ComputeCv

Inputs :

1. $sup(A_g)$, where g is an m-variable Boolean function.
2. $v \in F_2^m$.

Output : C_v.

Part 1 : Computation of list $\mathcal{L}$.
Set $\mathcal{L}$ equal to the empty list.
Set str equal to the empty string.
$\mathsf{Traverse}(str)$.

Assume $\mathcal{L} = ((str_1, val_1), \ldots, (str_n, val_n))$ at the end of Traverse.

Part 2 : Computation of C_v.

Set $B_0 = \cdots = B_{m-1} = 0$.

For $i = 1$ to n do

if $(wt(str_i) = j))$ then $B_j = B_j + val_i$.

if $(\psi(str_i, v) = (0, \ldots, 0))$ then $B = B + \frac{val_i}{2^{wt(str_i)}}$.

End For.

$C_v = (-1)^{wt_2(v)}(\frac{1}{2^m}((-1)^p - 1 - \sum_{i=0}^{m-1} B_i) + B)$.

Return C_v.

End Algorithm ComputeCv.

The subroutine Traverse() performs a depth first search on the tree $\mathcal{T}$ described before. The details of the algorithm Traverse() are given below. In the algorithm we use the notation $\sigma(tstr)$ for a p-bit string $tstr$ to mean $\sigma(S)$ where $S = \{i : tstr_i = 1\}$.

Algorithm Traverse(str)

Input : a binary string str of length at most p.

For $i = |str|$ to $p - 1$ do

Set $j = i - |str|$.

Set $tstr = str||0^j||1$.

If $(\mu(tstr) < m)$ then

If $(\sigma(tstr) \notin \mathsf{First}(\mathcal{L})$, then

Add $(\sigma(tstr), (-2)^{wt(tstr)}$ to $\mathcal{L}$.

Else suppose $(\sigma(tstr), val)$ is present in $\mathcal{L}$.

Set $val = val + (-2)^{wt(tstr)}$.

End If.

If $(|tstr| < p)$, then Traverse($tstr$).

End If.

End For.

End Algorithm Traverse

From the above discussion we obtain the following result.

Theorem 4.2 *Let $v_1, \ldots, v_t \in F_2^m$ and g be an m-variable Boolean function. Then $\{W_g(v_i) : 1 \leq i \leq t\}$ can be computed in time $O(m(|\mathcal{S}_1|\log(|\mathcal{S}_2|) + t|\mathcal{S}_2|))$.*

Proof : Part 1 of Algorithm ComputeCv has to be executed only once for all the vectors $v_1, \ldots, v_t$. This takes time $O(|\mathcal{S}_1|\log(|\mathcal{S}_2|))$. Part 2 of Algorithm ComputeCv has to run once for each of the vectors $v_1, \ldots, v_t$. Each execution of Part 2 takes time $O(|\mathcal{S}_2|)$. This gives the time complexity of the algorithm. Correctness of the algorithm follows from the previous discussion. ■

Remark : It is important to note that for small t it is possible to have $|\mathcal{S}_1|\log(|\mathcal{S}_2|) + t|\mathcal{S}_2| \ll \min(2^p, 2^m)$. In such situations it is possible to efficiently compute the Walsh transform of g for a small set of points. Cardinality of $|\mathcal{S}_1|$ and $|\mathcal{S}_2|$ depends on ANF and in the worst case can be $O(2^p)$. To keep $|\mathcal{S}_1|$ and $|\mathcal{S}_2|$ within a controllable limit, the ANF should be a sparse multinomial and support of ANF should contain terms with very high weights (nearly m). In Section 5, we provide some experimental data to support this intuition.

5 Experimental Results

Similar to Section 3, let $Sup(A_g) = \{\alpha^{(1)}, \ldots, \alpha^{(p)}\}$. We write $\alpha^{(i)} = (\alpha_{i,1}, \cdots, \alpha_{i,m})$ where $\alpha_{i,j} \in \{0,1\}$ for $1 \le i \le p$, $1 \le j \le m$. For $1 \le j \le m$, define $\rho_j = p - \sum_{i=1}^{p} \alpha_{i,j}$. Let $\rho_{max} = max_{j \in [m]}(\rho_j)$, $\rho_{avg} = \sum_{i=1}^{m} \frac{\rho_i}{m}$ and $w_{avg} = \sum_{i=1}^{p} \frac{wt(\alpha^{(i)})}{p}$. We use the notation $\mathcal{S}_1$ and $\mathcal{S}_2$ as defined in Equation 12.

Tables 1 and 2 consider the cases $p > m$ and $p < m$ respectively. (Here we note that the class of Boolean functions for which p is less than m is also very rich. For example this class includes an important subset of the class of bent functions.) The results in these two tables show that ρ_i's (specially ρ_{max}) are crucial to the complexity of the algorithm. If ρ_{max} is comparatively larger than the other ρ_i's then the size of $|\mathcal{S}_1|$ is exponential in ρ_{max}. Also note that in general $|\mathcal{S}_1| \ge 2^{\rho_{max}}$.

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert, 2^{\rho_{max}}$	$\rho_{max}, \rho_{avg}, w_{avg}$
$100, 250$	546	$2135470, 1048576$	$20, 15, 94$
$100, 250$	547	$3184044, 2097152$	$21, 15, 94$
$100, 250$	549	$5288601, 4194304$	$22, 15, 94$
$100, 250$	549	$9482897, 8388608$	$23, 15, 94$
$100, 250$	550	$17871500, 16777216$	$24, 15, 94$
$100, 250$	552	$34648698, 33554432$	$25, 15, 94$
$100, 250$	553	$68203122, 67108864$	$26, 15, 94$

Table 1

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert, 2^{\rho_{max}}$	$\rho_{max}, \rho_{avg}, w_{avg}$
$100, 50$	2258	$365465, 131072$	$17, 15, 70$
$100, 50$	2392	$1340716, 1048576$	$20, 15, 70$
$100, 50$	2429	$8678491, 8388608$	$23, 15, 70$
$100, 50$	2465	$67386792, 67108864$	$26, 15, 70$

Table 2

From the definition, the parameters m, p, ρ_{avg} and w_{avg} satisfy the following relationship: $m \times \rho_{avg} = p \times (m - w_{avg})$. For fixed m and p, ρ_{avg} decreases as w_{avg} increases. Similarly, for fixed m and ρ_{avg}, as p increases so does w_{avg}. In Table 3, we show the change of w_{avg} with p for $m = 100$ and $\rho_{avg} = 16$.

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert, 2^{\rho_{max}}$	$\rho_{max}, \rho_{avg}, w_{avg}$
$100, 50$	9976	$5313186, 524288$	$19, 16, 68$
$100, 100$	2934	$6482400, 524288$	$19, 16, 84$
$100, 200$	1414	$4609688, 524288$	$19, 16, 92$
$100, 400$	827	$6644436, 524288$	$19, 16, 96$

Table 3

It is possible to run the algorithm in cases where m is large but ρ_{max} is small. Table 4 provides some data which illustrates this fact.

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert, 2^{\rho_{max}}$	$\rho_{max}, \rho_{avg}, w_{avg}$
$200, 200$	725	$96213, 32768$	$15, 12, 188$
$400, 200$	2175	$207969, 16384$	$14, 12, 376$
$800, 200$	1938	$528605, 65536$	$16, 12, 752$
$3200, 100$	2290	$243188, 16384$	$14, 12, 2816$
$3200, 100$	2502	$4436706, 4194304$	$22, 12, 2816$

Table 4

In the above examples we see that $|\mathcal{S}_2|$ is small. In Table 5 we provide some examples for which $|\mathcal{S}_2|$ is large but w_{avg} is small.

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert, 2^{\rho_{max}}$	$\rho_{max}, \rho_{avg}, w_{avg}$
$50, 25$	134260	$21432167, 1048576$	$20, 16, 18$
$60, 30$	150554	$48852312, 8388608$	$23, 16, 28$
$70, 35$	92849	$61196173, 16777216$	$24, 16, 38$
$80, 40$	13403	$17968626, 33554432$	$26, 16, 48$
$90, 45$	22728	$585495610, 134217728$	$27, 16, 58$

Table 5

As long as ρ_{max} is small (say less than 25), it is possible to run the algorithm for quite large values of both m and p. Table 6 provides some evidence of this fact.

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert, 2^{\rho_{max}}$	$\rho_{max}, \rho_{avg}, w_{avg}$
$400, 800$	1578	$1442226, 32768$	$15, 14, 393$
$400, 800$	1588	$2489671, 1048576$	$20, 14, 393$

Table 6

Table 7 provides the actual running times taken by a simple C language implementation (Linux operating system on Pentium IV, 2.4 GHz CPU, 1 GB RAM).

m, p	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert$	time in seconds
$100, 50$	9976	5313186	50
$100, 100$	625	103455	2
$100, 200$	1297	71257808	830
$400, 800$	1588	2489671	1059

Table 7

Note that the following are true for the experiments performed above.

1. $|\mathcal{S}_1|$ is exponential in ρ_{max} and hence the run time of the algorithm is exponential in ρ_{max}.
2. The bounds on p are $0 \leq p \leq 2^m$. But for our algorithm to be useful the value of p can not be very large because in that case ρ_{max} can not be restricted to low values (say 25) and our algorithm will not work.
3. The value of w_{avg} should be high i.e. ANF should be very sparse multinomial and support of ANF should contain terms with very high weight.
4. In most cases, $|\mathcal{S}_2|$ is small compared to $|\mathcal{S}_1|$. Also $|\mathcal{S}_2|$ tends to grow as w_{avg} decreases.
5. In all the examples given above, the chosen functions were non-degenerate on all the variables. A Boolean function $g(x_1, \cdots, x_m)$ is said to be degenerate on variable x_i if $g(x_1, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_m) = g(x_1, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_m)$. The function g is said to be degenerate if it is degenerate on some variable, else it is said to be non-degenerate.

Finally, in summary we note that for certain types of Boolean functions on large number of variables algorithm ComputeCv can be used to compute the Walsh transform for a small set of points. This can provide some useful information about the Boolean function. For example, one can obtain the weight (or size of support) of g by computing $W_g(0)$. Also the ability to probe the spectral domain of the function at random points can provide an estimate of the nonlinearity of the function.

6 Concluding Remarks

In this paper we have developed an algorithm to compute the Walsh transform of Boolean function at a point from its algebraic normal form. This can also be extended to evaluate the Walsh transform for a set of points. The advantage of our method is that in certain situations it is possible to run our algorithm to evaluate the Walsh transform (at a small set of points) of Boolean functions with large number of variables and hence can be used in cases where the fast Walsh transform is not applicable.

An important parameter in the study of a Boolean function is its nonlinearity. To compute nonlinearity, it is necessary to compute the Walsh transform at all the points. For Boolean functions on a large number of variables (≥ 40), it is practically impossible to perform this task. In certain situations, our algorithm can be used to evaluate the Walsh transform at any particular point. In such situations, it is possible to use our algorithm in conjunction with some randomized heuristic (simulated annealing/ hill climbing/ genetic algorithms) to estimate the nonlinearity of the function. Thus we feel that the algorithm developed in this paper is a valuable technique for the study and analysis of Boolean functions.

Acknowledgements: We wish to thank Prof. J. L. Massey for reading the paper and providing several suggestions. We also wish to thank the reviewer for suggestions that led to the present simplified proof of Theorem 3.3 and removal of some complicated preliminary material.

References

[1] K. G. Beauchamp. *Applications of Walsh and Related Functions.* Academic Press, 1984.

[2] M. Bellare, D. Coppersmith, J. Hastad, M. Kiwi and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, Vol. 42, No. 6, pp. 1781–1795, November 1996.

[3] C. Carlet and P. Guillot. A new representation of Boolean functions. *Proceedings of AAECC'13*, Lecture Notes in Computer Science, 1719, pp 94-103, 1999.

[4] C. Carlet and P. Sarkar. Spectral Domain Analysis of Correlation Immune and Resilient Boolean Functions. *Finite Fields and their Applications*, Volume 8, Number 1, January 2002, Pages 120-130.

[5] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*, W. H. Freeman and Co., 1983.

[6] X.-D. Hou. Bent Functions, Partial Difference Sets, and Quasi-Frobenius Local Rings. *Designs, Codes and Cryptography* 20(3): 251-268 (2000).

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

[8] N. Linial, Y. Mansour and N. Nisan. Constant Depth Circuits, Fourier Transform, and Learnability. *Journal of the ACM*, 40(3): 607-620 (1993).

[9] P. Sarkar and S. Maitra. Cross-Correlation Analysis of Cryptographically Useful Boolean Functions and S-Boxes. *Theory of Computing Systems*, 35(1): 39-57 (2002).