Eighth International Workshop on Optimal Codes and Related Topics
July 10-14, 2017, Sofia, Bulgaria *pp. 17-24*

Fast implementation of the ANF transform

Valentin Bakoev v.bakoev@uni-vt.bg
"St. Cyril and St. Methodius" University of Veliko Turnovo, Bulgaria

Abstract. The algebraic normal forms (ANFs) of Boolean functions are used in computing the algebraic degree of S-boxes, which is one of the most important cryptographic criteria. It should be computed by fast algorithms so that more S-boxes are generated and the best of them are selected. Here we continue our previous work for fast computing the ANFs of Boolean functions. We represent and investigate the full version of bitwise implementation of the ANF Transform. The obtained algorithm has a time-complexity $\Theta((9n-2).2^{n-7})$ and $\Theta(2^{n-6})$ space complexity. The experimental results show that it is more than 20 times faster in comparison to the well-known byte-wise ANF Transform.

1 Introduction

The algebraic degree of an S-box is one of its most important cryptographic parameters. Its computing includes obtaining of the Algebraic Normal Forms (ANF) of the coordinate Boolean functions, which form the S-box. During the generation of S-boxes it should be done as fast as possible (as well as computing the other cryptographic criteria), in order to obtain more S-boxes and to select the best of them.

The algebraic degree of a Boolean function is defined by one special representation – in the area of algebra and cryptology it is known as ANF. It is not well-known that this representation is considered in the literature from other viewpoints and has many generalizations, for example:

- The classical (logical) one – this representation is known as Zhegalkin polynomial. The famous Zhegalkin theorem states that for every Boolean function there exists an unique polynomial form of it over the set of Boolean functions $\{x.y, x \oplus y, \tilde{1}\}$ ($\tilde{1}$ is the constant 1) [7, 10].
- Circuit theory, switching functions etc. – such representation is known as: modulo-2 sum-of-products, Reed-Muller-canonical expansion, positive polarity Reed-Muller form, etc. [6, 9].

The computing of these representations for a given Boolean function can be done in different ways [2, 3, 9]. The fastest of them are based on multiplication of special transformation matrix and the true table vector of Boolean function. This matrix is the same in the mentioned scientific areas and so the obtained algorithms and representations are equivalent. However, in dependence of the

area of consideration, they have different names: ANF Transform (ANFT), (fast) Möbius (or Moebius) Transform, Zhegalkin Transform, Positive polarity Reed-Muller Transform (PPRMT), etc. Our experience convinced us how important these facts are in searching papers by keywords.

The algorithm for performing ANFT in its typical form (i.e. operating on bytes) is well-known and is given in many sources. However its bitwise implementation is considered quite rarely. In [4, 5] the authors stress to the necessity of optimization issues and techniques, and consider some of them. They give two different programs (written in C language) that perform the ANFT, working with bitwise representation of Boolean functions in 32-bits computer words. However, in both sources the role of shifts and masks is not explained, the correctness and complexity of the programs are not given. It is not clear how to rewrite them to run on 64-bits (or more bits) computer words.

In 1998 we developed an algorithm for computing the Zhegalkin transform [8]. In 2008 we proposed other version of this algorithm (called PPRMT), which is derived in a different way [1]. We also commented its bitwise representation and implementation, that can improve its time and space complexity. However, this work did not attract the attention of cryptographers and a reason for this might have been its name – PPRMT.

This report is a continuation of [1]. Here we represent our full version of the bitwise ANFT with comments about its implementations and correctness. We obtain a general time complexity $\Theta((9n-2).2^{n-7})$ by using a bitwise representation of Boolean functions in 64-bits computer words. Some experimental results are also given.

2 Basic notions and preliminary results

Here we give the usual for cryptography terms and notations following [2, 3].

We consider the field of two elements $\mathbb{F}_2 = \{0, 1\}$ with two operations: sum modulo 2 (XOR), denoted by $x \oplus y$ and multiplication modulo 2 (AND), denoted by $x.y$ or simply xy, for $x, y \in \mathbb{F}_2$. The n-dimensional vector space over $\mathbb{F}_2$ is $\mathbb{F}_2^n$, it includes all 2^n binary vectors. For an arbitrary $a = (a_1, a_2, \ldots, a_n) \in \mathbb{F}_2^n$, $\bar{a}$ denotes the natural number $\bar{a} = \sum_{i=1}^{n} a_i.2^{n-i}$. The binary representation of $\bar{a}$ $(0 \le \bar{a} \le 2^n - 1)$ in n bits determines the coordinates of a. The correspondence between a and $\bar{a}$ is a bijection. For $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n) \in \mathbb{F}_2^n$, we say that "*a precedes lexicographically b*" and denote it by $a \prec b$ if $\exists\, k, 0 \le k < n$, such that $a_i = b_i$, for $i \le k$, and $a_{k+1} < b_{k+1}$. The relation "$\prec$", defined over $\mathbb{F}_2^n$, gives an unique *lexicographic* (*standard*) *order* of the vectors of $\mathbb{F}_2^n$ in the sequence $a_0 = (0, 0, \ldots, 0) \prec a_1 = (0, 0, \ldots, 0, 1) \prec \cdots \prec a_{2^n-1} = (1, 1, \ldots, 1)$. So $\bar{a}_0 = 0 < \bar{a}_1 = 1 < \cdots < \bar{a}_{2^n-1} = 2^n - 1$ and the natural number $i = \bar{a}_i$ is called a *serial number* of the vector a_i.

A *Boolean function* of n variables (denoted usually by $x_1, x_2, \ldots, x_n$) is a mapping $f : \mathbb{F}_2^n \to \mathbb{F}_2$, i.e. f maps any binary input $x = (x_1, x_2, \ldots, x_n) \in \mathbb{F}_2^n$

to a single binary output $y = f(x) \in \mathbb{F}_2$. Any Boolean function f can be represented in an unique way by the vector of its functional values, called a *True Table* vector: $TT(f) = (f_0, f_1, \dots f_{2^n-1})$, where $f_i = f(a_i)$ and a_i is the i-th vector in the lexicographic order of $\mathbb{F}_2^n$, for $i = 0, 1, \dots, 2^n - 1$. When the $TT(f)$ is considered as vector-column, it is denoted by $[f]$. The set of all Boolean functions of n variables is denoted by $\mathcal{F}_n$ and its size is $|\mathcal{F}_n| = 2^{2^n}$.

Other unique representation of any Boolean function $f \in \mathcal{F}_n$ is by *Algebraic Normal Form* (ANF), which is a multivariate polynomial

$$f(x_1, x_2, \dots, x_n) = \bigoplus_{u \in \mathbb{F}_2^n} a_{\bar{u}}\, x^u\,. \tag{1}$$

Here $u = (u_1, u_2, \dots, u_n) \in \mathbb{F}_2^n$, $a_{\bar{u}} \in \{0, 1\}$, and x^u means the monomial $x_1^{u_1} x_2^{u_2} \dots x_n^{u_n} = \prod_{i=1}^{n} x_i^{u_i}$, where $x_i^0 = 1$ and $x_i^1 = x_i$, for $i = 1, 2, \dots n$.

When $f \in \mathcal{F}_n$ and the TT(f) (with 2^n values) is given, the values of the coefficients $\bar{a}_0, \bar{a}_1, \dots, \bar{a}_{2^n-1}$ can be computed by fast algorithm, usually called *ANF Transform*. This algorithm is derived in different ways, but its versions are similar to each other. The vector $\bar{a} = (\bar{a}_0, \bar{a}_1, \dots, \bar{a}_{2^n-1})$, obtained after ANFT is denoted by A_f, or by $[A_f]$ when it is considered as a vector-column.

In [1, 8] we developed an algorithm for fast computing of ANFT in two different ways, based on the equalities with the transformation matrix, as in [6]. So, if $f \in \mathcal{F}_n$ and $TT(f) = (f_0, f_1, \dots f_{2^n-1})$, then:

$$[A_f] = M_n.\,[f]\,, \quad \text{and} \quad [f] = M_n^{-1}.\,[A_f] \quad \text{over } \mathbb{F}_2.$$

The matrix M_n is defined recursively, as well as by Kronecker product:

$$M_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad M_n = \begin{pmatrix} M_{n-1} & O_{n-1} \\ M_{n-1} & M_{n-1} \end{pmatrix}, \text{ or } M_n = M_1 \otimes M_{n-1} = \bigotimes_{i=1}^{n} M_1,$$

where M_{n-1} is the corresponding transformation matrix of dimension $2^{n-1} \times 2^{n-1}$, and O_{n-1} is a $2^{n-1} \times 2^{n-1}$ zero matrix. Furthermore $M_n = M_n^{-1}$ over $\mathbb{F}_2$ and hence the forward and the inverse ANFT are performed in the same way – so we consider only the forward ANFT. The pseudocode of our algorithm (referred here as Algorithm 1) is given below. The elements of the array f are bytes and they are numbered starting from 0. Every value of the TT(f) is stored in a separate byte as a Boolean value true or false. The result A_f is obtained in the same array. The precise estimation of the time complexity of Algorithm 1 is $\Theta(n.2^{n-1})$, whereas many authors give an estimation $O(n.2^n)$ for analogous algorithms.

3 Fast implementation of the ANFT

We note that an implementation of ANFT, based on the bitwise representation of Boolean function, is possible because of:

Algorithm 1 ANF_Transform (f, n)

```
1: blocksize ← 1
2: for step = 1 n do
3:    source ← 0 {initial position of the first source block}
4:    while source < 2^n do
5:       target ← source + blocksize {initial position of the first target block}
6:       for i = 0 blocksize − 1 do
7:          f[target + i] ← f[target + i] XOR f[source + i]
8:       end for
9:       source ← source + 2 * blocksize {beginning of the next source block}
10:   end while
11:   blocksize ← 2 * blocksize
12: end for
13: return f
```

(1) f and $A_f \in \mathbb{F}_2$.

(2) The sizes of blocks, XOR-ed to other blocks in the algorithm, are powers of 2, as well as the sizes of the computer words for representation of integers. So the source and the target blocks occupy parts of a computer word or some adjacent computer words – their number is also a power of 2;

(3) All necessary operations are available as fast processor instructions – shifts, bitwise XOR, AND, etc.

The idea for a bitwise implementation, given in [1], is illustrated by the following example.

Example 1. Let us consider the function $f(x_1, x_2, x_3) = (1, 0, 1, 1, 0, 1, 1, 0)$. The steps of ANFT for f, performed by Algorithn 1, are illustrated as usually by its *butterfly* (*signal-flow*) *diagram.*

(x_1,x_2,x_3)	f	step 1	f_1	step 2	f_2	step 3	A_f
(0,0,0)	1		1		1		1
(0,0,1)	0		1		1		1
(0,1,0)	1		1		0		0
(0,1,1)	1		0		1		1
(1,0,0)	0		0		0		1
(1,0,1)	1		1		1		0
(1,1,0)	1		1		1		1
(1,1,1)	0		1		0		1

Figure 1: The butterfly diagram for $f(x_1, x_2, x_3) = (1, 0, 1, 1, 0, 1, 1, 0)$

The bitwise representation of f needs one byte, $f = 10110110$. We follow the steps of Algorithm 1 and implement them in a bitwise manner. We form a mask for each step, so that it has ones in the positions of the source bits and zeros in the rest (target) bits. For the first step we use the mask $m_1 = 10101010$ (its hexadecimal value is 0XAA). The assignment $temp = f$ AND m_1 moves the values of all source bits to the variable *temp*, so $temp = 10100010$. A right-shift on *temp* by one bit (since the variable *blocksize* in Algorithm 1 takes an initial value 1) yields to $temp = 01010001$. Thus in *temp*: (1) the source-bits are slided to the positions of the target bits and (2) the places of all source bits in *temp* are occupied by zeros. The third operation is $f = f$ XOR *temp* – thus the values from the source blocks are added modulo 2 to these in the target blocks, and the source bits are preserved. So, after the first step f gets the value $f = 11100111$ (as in Figure 1). In the second step we use the mask $m_2 = 11001100$ (i.e. 0XCC) and right-shift by two bits (the variable *blocksize* in Algorithm 1 takes a value 2 in the second step), the third operation is the same. Analogously, in the third step the mask is $m_3 = 11110000$ (i.e. 0XF0) and the right-shift is by four bits. The values and the operations of each step are given in Table 1.

Step	Variables or operators	Values or results
Input	f	10110110
1.0	m_1	10101010
1.1	$temp = f$ AND m_1	10100010
1.2	$temp = temp$ SHR 1	01010001
1.3	$f = f$ XOR $temp$	11100111
2.0	m_2	11001100
2.1	$temp = f$ AND m_2	11000100
2.2	$temp = temp$ SHR 2	00110001
2.3	$f = f$ XOR $temp$	11010110
3.0	m_3	11110000
3.1	$temp = f$ AND m_3	11010000
3.2	$temp = temp$ SHR 4	00001101
3.3	$f = f$ XOR $temp$	11011011
Output	$A_f = f$	11011011

Table 1: Step-by-step bitwise ANFT

Following Eq. (1) we obtain the ANF: $f(x_1, x_2, x_3) = 1 \oplus x_3 \oplus x_2x_3 \oplus x_1 \oplus x_1x_2 \oplus x_1x_2x_3$. So, for any $f \in \mathcal{F}_3$, the bitwise ANFT performs the same steps and we unify them in the following 3 operators in C language:

```
f^= (f & m1)>>1;   f^= (f & m2)>>2;   f^= (f & m3)>>4;
```

For the functions $f \in \mathcal{F}_4$ the bitwise ANFT can be done in a similar way. We have to double the masks (i.e. to concatenate each of them by themselves)

to fill in 2 bytes – so they should be: $m_1 = 1010101010101010$ (0XAAAA), $m_2 = 1100110011001100$ (0XCCCC) and $m_3 = 1111000011110000$ (0XF0F0). The last (fourth) step needs one more mask – it is $m_4 = 1111111100000000$ (0XFF00). Analogously, for the functions $f \in \mathcal{F}_5$ we use 4 bytes, we double the masks $m_1, \ldots, m_4$ and add a new mask m_5 (0XFFFF0000) for the last (fifth) step, and so on. For the functions $f \in \mathcal{F}_6$ the masks and the steps are given in Listing 1. As we have shown, when $f \in \mathcal{F}_n$ and $2 \leq n \leq 6$, there are n steps, each of them is performed by 4 operations. So A_f is obtained by $4.n$ operations only. Thus we have described the ***first case*** when the bitwise representation of f occupies one computer word (of 1, 2, 4 or 8 bytes).

The ***second case*** is when $f \in \mathcal{F}_n$ and $n > 6$. We use a bitwise representation of f as an array of 2^{n-6} computer words of size $2^6 = 64$ bits. So the space complexity is $\Theta(2^{n-6})$. The **fast ANFT** performs two main steps:

Step 1) Bitwise ANFT on each computer word, as it was described above. Since we have 2 additional assignments, we obtain exactly $4n + 2$ operations for one computer word. So $\Theta((4n + 2).2^{n-6})$ operations will be performed.

Step 2) The usual ANFT (as in Algorithm 1) on whole computer words. Here the number of operations is $\Theta((n - 6).2^{n-7})$

Therefore the total time complexity of the fast ANFT is:

$$\Theta((4n + 2).2^{n-6}) + \Theta((n - 6).2^{n-7}) = \Theta((9n - 2).2^{n-7}) .$$

The correctness of the fast ANFT follows by the notes in the beginning of this section and also by the correctness of Algorithm 1, since the fast ANFT does the same, but in an optimized way. Here is its code in C language.

Listing 1: The C code of fast ANFT

```
typedef unsigned long long ull;
const ull m1= 0XAAAAAAAAAAAAAAAA, m2= 0XCCCCCCCCCCCCCCCC,
          m3= 0XF0F0F0F0F0F0F0F0, m4= 0XFF00FF00FF00FF00,
          m5= 0XFFFF0000FFFF0000, m6= 0XFFFFFFFF00000000;
const int num_of_vars= 8; //number of variables
const int num_of_steps= num_of_vars - 6; //for Step (2) of ANFT
const int num_of_comp_words= 1<<num_of_steps; //=4 for 8 vars

void ANF (ull f[]) {
// Step 1 - bitwise ANF on computer words
    for (int k= 0; k < num_of_comp_words; k++) {
        ull temp= f[k];
        temp ^= (temp & m1)>>1;
        temp ^= (temp & m2)>>2;
        temp ^= (temp & m3)>>4;
        temp ^= (temp & m4)>>8;
        temp ^= (temp & m5)>>16;
        temp ^= (temp & m6)>>32;
        f[k]= temp;
    }
```

```
// Step 2
    int blocksize= 1;
    for (int step= 1; step <= num_of_steps; step++) {
        int source= 0; //init. pos. of the first source block
        while (source < num_of_comp_words) {
            int target= source + blocksize;
            for (int i= 0; i < blocksize; i++) {
                f[target + i] = f[target + i] ^ f[source + i];
            }
            source += 2*blocksize; //beginning of next source bl.
        }
        blocksize *= 2;
    }
    return;
}
```

4 Experimental results and conclusions

To compare the real time complexities of the usual byte-wise implementation of ANFT and the fast ANFT (the implementation from Listing 1), we conduct a sequence of tests on Intel Pentium CPU G4400, 3.3 GHz, 4GB RAM, under Windows 10 OS. All tests are performed 3 times and their running times are taken in average. The time for reading from file, for converting the input to array of bytes (for the byte-wise ANFT), etc. is excluded. The given running times are for execution of the byte-wise ANFT and the Fast ANFT only.

For all 2^{32} Boolean functions of 5 variables, the byte-wise ANFT runs in 311.459 sec., whereas fast ANFT runs in 3.808 sec. – so it is more than 80 times faster.

For the other tests we created 4 files with $10^6, 10^7, 10^8$ and 10^9 integers, randomly generated in 64-bits computer words. They were used as input data, i.e. as TT vectors of functions. The results in Table 2 concern the largest file ($\approx$ 1.4 GB).

	Running times in seconds for:				
Number of variables	6	8	10	12	16
Number of functions	10^9	$10^9/4$	$10^9/16$	$10^9/64$	$10^9/1024$
Byte-wise ANFT	171,993	170,773	234,666	280,353	363,212
Fast ANFT	5,357	6,699	8,985	10,925	12,893
Ratio:	32,108:1	25,494:1	26,117:1	25,662:1	28,171:1

Table 2: Experimental results about byte-wise ANFT and Fast ANFT

The theoretical time complexity for fast ANFT shows that it is at least 8 times faster than byte-wise ANFT. The experimental results show that it is at

least 20 times faster for smaller input files and this ratio grows when the filesize increases. As Table 2 shows, the fast ANFT is more than 25 times faster than the byte-wise ANFT for the largest input test file.

Next possible step and a future goal for improvement the running time of this version of ANFT is to develop a parallel implementation, which will yield to faster computing of algebraic degree in design of S-boxes.

References

[1] Bakoev V. and Manev K., Fast computing of the positive polarity Reed-Muller transform over GF(2) and GF(3), in *Proc. of the XI Intern. Workshop on Algebraic and Combinatorial Coding Theory (ACCT), 16-22 June, 2008, Pamporovo, Bulgaria*, 2008, 13–21.

[2] Canteaut A., *Lecture notes on Cryptographic Boolean Functions*, Inria, Paris, France, 2016.

[3] Carlet C., *Boolean Functions for Cryptography and Error Correcting Codes*. In: Crama C, Hammer PL, (Eds.), *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, Cambridge Univ. Press, 2010, 257–397.

[4] Fuller J., *Analysis of affine equivalent Boolean functions for cryptography*, Ph.D. Thesis, Queensland University of Technology, Australia, 2003.

[5] Joux A., *Algorithmic Cryptanalysis*, Chapman & Hall/CRC Cryptography and Network Security, 2012.

[6] Harking B., Efficient algorithm for canonical Reed-Muller expansions of Boolean functions, *IEE Proc. Comput. Digit. Tech.*, **137**, 5, 1990, 366–370.

[7] Manev K., *Introduction to Discrete Mathematics*, Fourth Edition, KLMN, Sofia, Bulgaria, 2007. (in Bulgarian)

[8] Manev K. and Bakoev V., Algorithms for performing the Zhegulakin transfroemation, in *Proc. of the XXVII Spring Conf. of the Union of Bulgarian Mathematicians, (Pleven, Bulgaria, April 9–11, 1998)*, Mathematics and education in mathematics, Sofia, 1998, 229–233.

[9] Porwik P., Efficient calculation of the Reed-Muller form by means of the Walsh transform, *Int. J. Appl. Math. Comput. Sci.*, **12**, 4, 2002, 571-579.

[10] Yablonski S., *Introduction to discrete mathematics*, Fourth Edition, Visshaia Shkola, Moskow, 2003. (in Russian)