Noname manuscript No.
(will be inserted by the editor)

Fast Multiplication of the Algebraic Normal Forms of Two Boolean Functions.

Subhabrata Samajder · Palash Sarkar

the date of receipt and acceptance should be inserted later

Abstract The contribution of this paper is twofold. Firstly, it proposes a simple algorithm which performs the multiplication of two n-variate boolean functions in their algebraic normal forms in $\mathcal{O}(n2^n)$ time and $\mathcal{O}(2^n)$ space. Secondly, it proposes a fast implementation (MultANF$_w$) of the algorithm which works with w-bit words. Results for $w = 8, 32$ and 64 show that the 64-bit implementation is the fastest. To further analyze the performance, a sparse implementation has been done, which we call quadratic implementation. It has been observed that for a w-bit implementation, if the product of the number of monomials of the two input polynomials is $< 2^{n-\log_2 w}$, then the quadratic implementation performs better than MultANF$_w$. It is also found that MultANF$_w$ performs much better than the algorithm internally used by SAGE for all the three variants, i.e., $w = 8, 32$ and 64. Our study also indicates that quadratic implementation performs better than SAGE.

Keywords Multivariate Polynomial Multiplication · Boolean Functions · Algebraic Normal Form (ANF)

1 Introduction

Let $\mathbb{R} = GF(2)\,[x_1, x_2, \ldots, x_n]\,/\,\langle x_1^2 - x_1, \ldots, x_n^2 - x_n\rangle$. We consider polynomials in $\mathbb{R}$. Such polynomials can be considered to be the algebraic normal form of n-variable Boolean functions, which are maps from $\{0,1\}^n$ to $\{0,1\}$. Multiplication of Boolean functions is a basic operation and is of interest in itself. Apart from this, it also has a wide range of applications.

Subhabrata Samajder
Applied Statistics Unit, Indian Statistical Institute
203, B. T. Road, Kolkata - 700108, INDIA E-mail: subhabrata_r@isical.ac.in

Palash Sarkar
Applied Statistics Unit, Indian Statistical Institute
203, B. T. Road, Kolkata - 700108, INDIA E-mail: palash@isical.ac.in

The Buchberger's algorithm ([Buc06], [Buc98]) and its improvements, the F_4 and F_5 algorithms ([Fau99], [Fau02]), to compute the Gröbner basis over $\mathbb{R}$ essentially use polynomial multiplications to cancel out the leading terms. Hence, improving upon polynomial multiplications over $\mathbb{R}$, will speed up these algorithms.

The algebraic immunity ([CM03], [MPC04], [Dal06]) of Boolean functions, is crucial to the security of the block ciphers and stream ciphers. The definition of algebraic immunity says that the algebraic immunity for a Boolean function f is the minimum degree of g, such that $f.g = 0$. Thus, one can see that improving the time taken to multiply two Boolean functions in their ANF's has direct application to algebraic immunity. It also has applications in non-linear codes, like higher order Reed-Muller Codes and Kerdock Codes ([PMS$^+$98]).

Multiplication of two *sparse* polynomials p and q having l_p and l_q terms each will have about $l_q l_q$ terms and so the usual algorithm which takes $\mathcal{O}(l_p l_q)$ time, is optimal. It would be nice to investigate whether this can be improved in case of *dense* polynomials, where the number of variables is, say 30.

Our Results : A simple observation leads to an $\mathcal{O}(n2^n)$ time and $\mathcal{O}(2^n)$ space recursive algorithm. Asymptotically, this is competitive with general purpose Fourier transform based multivariate polynomial multiplication algorithm ([Mat08]) specialized to the binary case. To the best of our knowledge, the binary case does not seem to have received separate attention. On the other hand, for cryptographic application, the binary case is arguably the most important case.

Our contribution is not only in identifying a simple algorithm for multiplication of ANF's of Boolean functions, but also, in carrying out high quality software implementation. We make a careful study of the algorithm and identify ways to speed up. The first issue is to avoid recursion. For this we simulate the recursion tree independently for each of the two input polynomials p and q. We call this as our pre-process step. Next, instead of bit level AND operations, 8-bit table lookups are used to multiply two 3-variate polynomials at once. After table lookups we again traverse up the recursion tree by doing similar set of operation to finally get the product pq. This step is called the post-process step.

Notice that the polynomials, can also be seen as a sequence of bits. To make use of the w-bit word arithmetic and hence improve speed, the polynomials are packed in w-bit words. Three different implementations of our algorithm is proposed, by taking $w = 8, 32$ and 64. A detailed comparison amongst these three implementations is given. Comparison with the software package SAGE shows that our implementations work much better than SAGE.

We have also done an efficient sparse implementation, which we call the quadratic implementation. It was then compared with the w-bit implementations mentioned above. We found that for the w-bit implementation if the number $l_p l_q$ is greater or equal to $2^{n-\log_2 w}$, then the MultANF$_w$ algorithm performs better than the quadratic implementation. For sparse case also, we

have compared the quadratic implementation with that of SAGE and found that the quadratic implementation works better than SAGE.

The organization is as follows : in Section 2, we give the basic idea and the propose ways to further improve upon our basic idea. A non-recursive w-bit implementation MultANF_w is proposed in Section 3. In Section 4, we give a detailed comparison of MultANF_w, with its variants and with SAGE. Lastly, in Section 5, we conclude this paper.

2 The Algorithm

In the first half of this section we give the basic idea, then we describe an iterative algorithm for multiplying two boolean functions in their ANF's and lastly, we conclude the section by pointing out ways in which we can further improve our iterative algorithm.

2.1 Basic Idea

Let, $p\left(x_1,\ldots,x_n\right), q\left(x_1,\ldots,x_n\right) \in \mathbb{R}$. Write,

$$\begin{aligned} p\left(x_1,\ldots,x_n\right) &= x_n \cdot p_1\left(x_1,\ldots,x_{n-1}\right) \oplus p_0\left(x_1,\ldots,x_{n-1}\right) \\ q\left(x_1,\ldots,x_n\right) &= x_n \cdot q_1\left(x_1,\ldots,x_{n-1}\right) \oplus q_0\left(x_1,\ldots,x_{n-1}\right). \end{aligned}$$

Then,

$$\begin{aligned} pq &= \left(p_1q_1\right) x_n^2 \oplus \left(p_1q_0 \oplus p_0q_1\right) x_n \oplus p_0q_0 \\ &= \left(p_1q_1 \oplus p_1q_0 \oplus p_0q_1\right) x_n \oplus p_0q_0; \qquad \left[\text{Since, } x_n^2 = x_n \text{ in } \mathbb{R}.\right] \\ &= \left\{\left(p_1 \oplus p_0\right)\left(q_1 \oplus q_0\right) \oplus p_0q_0\right\} x_n \oplus p_0q_0. \end{aligned}$$

Thus, the number of $(n-1)$-variate multiplications required is 2 instead of 4 at the cost of one extra addition.

Note 1 This is a very simple observation and leads naturally to a fast recursive algorithm for multiplication of two ANF's. To the best of our knowledge, it does not seem that the literature records this approach for multiplication of ANF's.

Let, $t(n)$ denote the time taken to multiply two $n-$variate polynomials and $e(n)$ denote the time taken to add two $n-$variate polynomial. Then, we have

$$t(n) = 2t(n-1) + 4e(n-1).$$

Solving, we get

$$\begin{aligned} t(n) = 2^n t(0) + 4 \times \big\{ & e(n-1) + 2 \times e(n-2) + 2^2 \times e(n-3) + \ldots \\ & + 2^{n-2} \times e(1) + 2^{n-1} \times e(0) \big\}. \end{aligned}$$

Since, $e(n) = 2^n \cdot e(0)$, using this we get,

$$t(n) = 2^n t(0) + 4n2^{n-1}e(0),$$

where, $t(0)$ and $e(0)$ denote the time taken for bit-wise AND and XOR. Therefore,

$$t(n) = \mathcal{O}(n2^n) = \mathcal{O}(2^{n+\log_2 n}) = \mathcal{O}(m \log_2 m),$$

where $m = 2^n$. For "dense" polynomials, the size of the input will be about $\mathcal{O}(m)$ and so this $\mathcal{O}(m \log_2 m)$ algorithm is very attractive.

On the other hand, if p and q are "sparse" having l_p and l_q monomials respectively, then one would expect the product to have about $l_p l_q$ monomials. The direct algorithm for multiplication will require $\mathcal{O}(l_p l_q)$ time and is about the best that one can expect. So the above $\mathcal{O}(m \log_2 m)$ time algorithm is better only if the two polynomials are "dense". More comparative details are given later.

2.2 An Iterative Algorithm

We represent polynomials in $\mathbb{R}$ using a sequence of bits. In this sequence, we denote the presence of every monomial by a single bit. Since the number of such possible monomials in $\mathbb{R}$ is 2^n, we thus use 2^n bits to represent any polynomial in $\mathbb{R}$.

It is clear that one can compute the values of p_0, $(p_0 \oplus p_1)$, q_0 and $(q_0 \oplus q_1)$ independently and then multiply them to get the required $p_0 q_0$ and $(p_0 \oplus p_1) \cdot (q_0 \oplus q_1)$. Thus one needs to compute p_0 and $p_0 \oplus p_1$ (respectively, q_0 and $q_0 \oplus q_1$) from p (respectively, q). Using the same idea recursively, we thus get two recursive tree (one each for p and q). Notice that both p_0 and $p_1 \oplus p_0$ are polynomials in $n-1$ variables, namely $x_1, \ldots, x_{n-1}$. Thus, we see that at every step of the recursion the number of variables gets reduced by 1.

Suppose, p is represented by a 2^n bit array A (say). Then, p_0 corresponds to the first 2^{n-1} bits of p and p_1 the last 2^{n-1} bits of p. Hence, $p_0 \oplus p_1$ is nothing but bit-wise XOR of the 1st half with the 2nd half of A. In the next step of the recursion tree, we repeat the same, each for p_0 and $p_0 \oplus p_1$ with n now reduced to $n-1$. We go on doing this until $n = 1$. In which case, we reach the leaves of the recursion tree. Notice that multiplication here is equivalent to bit-wise AND-ing. Thus to multiply two polynomials p and q, we first build two recursion tree independently, each having 2^n 1-bit leaf nodes and then do a bit-wise AND among the corresponding leaf nodes of p and q. Now, to get the final result pq, we traverse upwards from the leaves (which contains the bit-wise AND of the corresponding leaf nodes of p and q) to the root by doing similar kind of operations. Notice that we now have $p_0 q_0$ and $(p_0 \oplus p_1)(q_0 \oplus q_1)$ and we need $(p_0 q_0 \oplus (p_0 \oplus p_1)(q_0 \oplus q_1))\, x_n$, which is equivalent to XOR-ing $p_0 q_0$ with $(p_0 \oplus p_1)(q_0 \oplus q_1)$ and then concatenating the result with $p_0 q_0$ (see Figure 1).

Extracting a bit from a byte is costly. Hence, we use table - lookups to avoid this. Instead of going all the way down to the nth level, we stop at

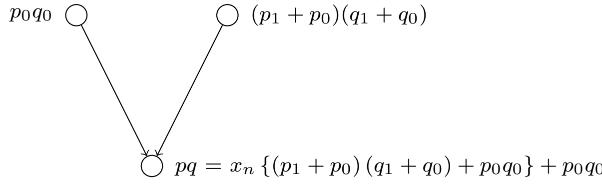

Fig. 1 Figure depicting the basic recursion step while returning back.

level $n-\beta$ and use table lookups to perform multiplication of two β variable polynomials. The value of β is taken to be 3, because the table corresponding to $\beta=4$ becomes very large. We thus pack the polynomials p and q in 8-bit arrays and use 8-bit XOR to multiply p and q.

2.3 Further Improvement

One may use w-bit XOR instead of 8-bit, assuming the architecture allows w-bit word arithmetic, where $w=2^k, k\geq 3$. The motivation is to save on the number of 8-bit XOR's. Thus, using one w-bit XOR, one can save $2^{\log_2 w-3}$ many XOR's. However, doing it this way one can only go up to $n-\log_2 w$ level, since, as mentioned in the previous section, maintaining a table of size greater than 3-variables is not feasible. Hence using w-bit words, involves, an additional task of UNPACKING and PACKING the w-bit word into bytes so that one can use the 8-bit table lookup. The naive approach to do this, is to copy each w-bit word into a byte array and basically use the same method to multiply two $\log_2 w$-variate polynomials using an 8-bit table lookup. And after multiplication copy back the result into a w-bit word.

We however, instead of directly copying the w-bit words to and back from byte arrays, use a constant amount of extra space to get an algorithm which not only saves us the cost of copying but also saves on the number of XOR's. The idea is to use $2^{\log_2 w-3}$ many w-bit word masks, say $M_1,\dots M_{2^{\log_2 w-3}}$ plus an additional temporary variable "temp", where M_i contains 1 in the bit positions $j\cdot 2^{\log_2 w-i}+k, j\in\left\{0,2,4,\dots 2^i-2\right\}$ and $k\in\left\{0,1,2,\dots,2^{\log_2 w-i}-1\right\}$ and 0 elsewhere. The M_i's actually simulate each level of the tree, corresponding to each w-bit word. Thus, for example, during the 1st level of the tree for each w-bit word, M_1 consists of 1 in the bit positions $0,1,\dots,2^{\log_2 w-1}-1$ positions and 0 elsewhere. M_1 is then AND-ed with the w-bit word to pick the corresponding p_0 (here we assume the left-most bit to be our LSB) and is stored in the temporary word "temp"; "temp" is then right shifted by $2^{\log_2 w-1}$ and XOR-ed with the w-bit word to get the corresponding $p_0\oplus p_1$. Thus after doing this we have p_0 in the first half of the w-bit word and $p_0\oplus p_1$ in the second half, which is what we wanted. Hence, for each level we need 3 (1 AND, 1 SHIFT and 1 XOR) w-bit operations. The PACKING process is the same as that of UNPACKING, except that the masks are used in a reverse order. For table look-ups we use $2^{\log_2 w-3}$ many additional masks, $B_{i+1}, i\in\{0,1,2,\dots,$

$2^{\log_2 w-3} - 1\}$, to extract the corresponding i^{th} byte from a w-bit word, where B_{i+1} contains 1 in the bit positions $8*i, 8*i+1, 8*i+2, \ldots, 8*i+7$ and 0 elsewhere.

Thus, for each level of PACKING and UNPACKING, we need 1 w-bit AND, 1 SHIFT on a w-bit word and 1 w-bit XOR operations. Therefore, for each PACKING and UNPACKING procedure we require $3 \cdot (\log_2 w - 3)$ w-bit operations. Also, for each table look-up we require 2 extra w-bit operations. We need 1 AND for extracting a particular byte and 1 SHIFT to bring the value of the extracted w-bit word within the range of 0 to 255. Since for each w-bit word we require $2^{\log_2 w-3}$ many table look-ups therefore, for each w-bit word we need $2 \cdot 2^{\log_2 w-3}$ many w-bit operations for table look-ups.

3 A w-bit Non-recursive Algorithm

In this section, we summarize our discussion in Section 2 to give a w-bit non-recursive algorithm, called MultANF_w (see Algorithm 6). The routine MultANF_w takes as input T, A, B, n, w, where A and B are the corresponding w-bit word representation of two n-variate polynomials ($n > \log_2 w \geq 3$) and $T_{256\times256}$ is a 8-bit table look-up. MultANF_w multiplies the polynomials A and B with the help of table T and stores the result in C.

To do this, the MultANF_w routine calls the subroutines "PRE_PROCESS" (Algorithm 1), "UNPACK" (Algorithm 2), "EXTRACT_AND_LOOKUP" (Algorithm 3), "PACK" (Algorithm 4) and "POST_PROCESS" (Algorithm 5). The subroutine PRE_PROCESS corresponds to the operations while descending down the recursion tree, whereas the subroutine "POST_PROCESS" corresponds to the operations while ascending up the recursion tree. Notice that the subroutine 'UNPACK" is called twice once each for the w-bit words $A[i]$ and $C[i]$.

The subroutines PACK and UNPACK are the same as PACKING and UNPACKING, as described in the previous section (Section 2.3). The subroutine EXTRACT_AND_LOOKUP extracts each byte from the w-bit words A and B; does the corresponding table lookup and then stores the value returned by the table in the exact byte position of C.

Algorithm 1: PRE_PROCESS (A, B, n, i)

```
Input: A, B, n, i
for j = 0, 1, 2, ..., 2^i − 1 do
    for k = 0, 1, ..., 2^(n−i−1) − 1 do
        A[2^(n−i−1) + j · 2^(n−i) + k] = A[2^(n−i−1) + j · 2^(n−i) + k] ⊕ A[j · 2^(n−i) + k]
        B[2^(n−i−1) + j · 2^(n−i) + k] = B[2^(n−i−1) + j · 2^(n−i) + k] ⊕ B[j · 2^(n−i) + k]
    end
end
```

Algorithm 2: UNPACK $(X,\ n)$: Unpacks a w-bit word to a byte array.

```
Input: a w-bit word X; n := log_2 w − 3.
for i = 0, 1, 2, ..., n − 1 do
    temp = Bit-wise AND of X and M_{i+1}
    temp = SHIFT right temp by 2^{n+3−i−1} (according to our assumption, the
    left-most bit is the LSB)
    X = temp XOR X
end
```

Algorithm 3: EXTRACT_AND_LOOKUP $(X,\ Y,\ Z,\ n)$: Extracts bytes from w-bit words X and Y, does a table look-up and stores the result in the corresponding byte of Z.

```
Input: w-bit words X, Y, Z; table T; n := log_2 w − 3
for i = 0, 1, ..., 2^n − 1 do
    if i = 0 then
        Z := T[X AND B_1][Y AND B_1]
    end
    else
        temp := T[(X AND B_{i+1}) SHIFT left by i ·
        2^3 bits.][(Y AND B_{i+1}) SHIFT left by i · 2^3 bits.] (According to our
        assumption the left-most bit is the LSB).
        Z := temp XOR Z

    end
end
```

Algorithm 4: PACK $(Z,\ n)$: Packs a w-bit word into a byte array.

```
Input: a w-bit word Z; n := log_2 w − 3.
for i = n − 1, n − 2, n − 3, ..., 0 do
    temp = Bit-wise AND of Z and M_{i+1}
    temp = SHIFT right temp by 2^{n+3−i−1} (according to our assumption, the
    left-most bit is the LSB)
    Z = temp XOR Z
end
```

Cost Analysis For MultANF$_w$: In both Algorithms 1 and 5, the loops run for $2^i \cdot 2^{(n-\log_2 w)-i-1} = 2^{n-\log_2 w-1}$. For each such iteration, we do two w-bit XOR's for PRE_PROCESS and one w-bit XOR for POST_PROCESS. Hence, the total number of w-bit XOR operations for each PRE_PROCESS and POST_ PROCESS call are $2 \cdot 2^{n-\log_2 w-1} = 2^{n-\log_2 w}$ and $2^{n-\log_2 w-1}$, respectively. Also, notice that in MultANF$_w$, PRE_PROCESS and POST_PROCESS are each called $n - \log_2 w$ many times. Therefore, the total number of w-bit XOR operations required in the PRE_PROCESS and POST_PROCESS part of MultANF$_w$ is $(n - \log_2 w) \cdot 2^{n-\log_2 w} + (n - \log_2 w) \cdot 2^{n-\log_2 w-1} = 3 \cdot (n - \log_2 w) \cdot 2^{n-\log_2 w-1}$.

As discussed in Section 2.3, for each w-bit word we require $3 \cdot (\log_2 w - 3)$ many w-bit operations for each call to PACK and UNPACK algorithm and

Algorithm 5: POST_PROCESS (C, n, i)

```
Input: C, n, i
for j = 0, 1, 2, ..., 2^i − 1 do
    for k = 0, 1, ..., 2^(n−i−1) − 1 do
        C[2^(n−i−1) + j · 2^(n−i) + k] = C[2^(n−i−1) + j · 2^(n−i) + k] ⊕ C[j · 2^(n−i) + k]
    end
end
```

Algorithm 6: MultANF_w (T, A, B, C, n, w) : A non recursive algorithm to multiply two boolean functions in their ANF's.

```
Input: 8-bit Look-up Table T; Two polynomials A and B; C for Result; number
       of varibles n; word size w
Output: C := Product of A and B
for i = 0, 1, 2, ..., n − log2 w − 1 do
    PRE_PROCESS(A, B, n − log2 w, i)
end
for i = 0, 1, 2, ..., 2^(n−log2 w) − 1 do
    UNPACK (A[i], log2 w − 3)
    UNPACK (B[i], log2 w − 3)
    EXTRACT_AND_LOOKUP (A[i], B[i], C[i], log2 w − 3)
    PACKING (C[i], log2 w − 3)
end
for i = n − log2 w − 1, n − log2 w − 2, n − log2 w − 3, ..., 0 do
    POST_PROCESS(C, n − log2 w, i)
end
```

$2 \cdot 2^{\log_2 w-3}$ many w-bit operations for table look-ups, plus $2^{log_2 w-3}$ many 8-bit table look-ups. The total number of such w-bit words is $2^{n-\log_2 w}$. Also notice that UNPACK is called twice whereas PACK is called once. Therefore, the total cost to multiply two n-variate polynomial using our w-bit non-recursive algorithm is :

1. $2^{\log_2 w-3} \cdot 2^{n-\log_2 w} = 2^{n-3}$ many 8-bit table look-ups.
2. $2 \cdot 2^{n-\log_2 w} \cdot 2^{\log_2 w-3} = 2^{n-2}$ many w-bit operations for table look-ups.
3. $2^{n-\log_2 w} \cdot (3 \cdot (3 \cdot (\log_2 w - 3))) = 9 \cdot (\log_2 w - 3) \cdot 2^{n-\log_2 w}$ many w-bit operations for PACKING and UNPACKING.
4. $3 \cdot (n - \log_2 w) \cdot 2^{n-\log_2 w-1}$ many w-bit XOR's for the PRE_PROCESS and POST_PROCESS.

4 Experimental Results

We present experimental results based on three separate implementations of MultANF$_w$, with $w = 8$, $w = 32$ and $w = 64$. We have used "C" language for our implementation. To further gain in speed we did some further modifications to our algorithm like using macro calls instead of function calls. The table T is implemented as one-dimensional array instead of a two-dimensional one. Thus, the entry corresponding to $T[A[i]][B[i]]$ is now $T[(A[i] << 8) + B[i]]$. For code

n	Average Cycles for 8 bit	Average Cycles for 32 bit	Speedup of 32-bit w.r.t 8 bit	Average Cycles for 64 bit	Speedup of 64-bit w.r.t 8-bit	Speedup of 64-bit w.r.t 32-bit
6	498.53	121.01	4.12	92.73	5.38	1.31
7	1138.23	428.95	2.65	199.38	5.71	2.15
8	2273.35	1032.89	2.20	1022.83	2.22	1.01
9	5013.86	1853.20	2.71	1276.61	3.93	1.45
10	11055.29	3871.94	2.86	2437.25	4.54	1.59
11	23608.47	8357.06	2.83	6010.26	3.93	1.39
12	34680.06	7711.84	4.50	5341.51	6.50	1.44
13	53976.73	16093.17	3.35	11153.91	4.84	1.44
14	103962.07	34223.26	3.04	23296.39	4.46	1.47
15	221928.42	73352.13	3.03	49992.79	4.44	1.47
16	466755.57	153265.65	3.05	101450.16	4.60	1.51
17	1014411.71	321682.42	3.15	212650.40	4.77	1.51
18	2075710.70	681210.39	3.05	441465.78	4.70	1.54
19	4401203.98	1433646.38	3.07	915821.38	4.81	1.57
20	9786430.84	3132142.40	3.13	2500430.46	3.91	1.25
21	20418478.40	6441914.73	3.17	5112594.99	3.99	1.26
22	43212647.62	13552823.50	3.19	10629153.25	4.07	1.28
23	89719530.45	28183683.11	3.18	21806265.54	4.11	1.29
24	190141764.33	59136263.78	3.22	45559914.11	4.17	1.30
25	401052397.73	130650693.03	3.07	106224818.55	3.78	1.23
26	838518978.22	299963811.34	2.80	272976258.05	3.07	1.10
27	1759215397.18	646245016.94	2.72	600701064.94	2.93	1.08
28	3635571731.89	1323794840.80	2.75	1239783643.15	2.93	1.07
29	7543793814.89	2735720452.18	2.76	2541063909.56	2.97	1.08
30	15606584912.85	5572652029.49	2.80	5109022401.64	3.06	1.09

Table 1 Table showing the speed (in cycles) comparisons between 8-bit, 32-bit and 64-bit implementations.

optimization we have used “O1” and the “funroll-all-loops” directives of the “gcc” compiler.

All our implementations were run on a HP Z800 Workstation. The machine has 96 GB RAM, 12 Intel(R) Xeon(R) CPU X5675 3.07GHz processor, 384 kB L1 cache, 1536 kB L2 cache and 12288 kB L3 cache. As for OS, we have used ”Ubuntu 12.04 LTS” with Linux 3.2.0-24-generic x86_64 kernel version. To get the running time in terms of number of cycles, we have used the “RDTSC” register, available in Intel processors. To train the “cache” and “branch predictors”, we have used one-fourth of the total number of iterations (For further details see Shay Gueron [Gue11]).

Table 1, compares the speed of our three implementations for number of variables “n” ranging from 6 to 30. As expected, our 64-bit implementation works faster than the other two implementations. A single multiplication of 30-variate polynomial using MultANF_w can be done in 1.66 secs on an average.

We next compare our 8-bit implementation MultANF_8 in “C” with that of SAGE. Table 2 gives the comparison of the performance of MultANF_8 with SAGE. The entries in the tables, denote the running time in seconds (s) and nanoseconds (ns). To get the timings in seconds and nanoseconds we have used the functions “timeit” for SAGE. Same inputs were used for the two

n	$\mathbf{MultANF}_w$	sage
3	0.80 ns	94773.05 ns
4	1.84 ns	127928.97 ns
5	55.55 ns	197319.98 ns
6	70.78 ns	354038.95 ns
7	161.31 ns	762128.12 ns
8	718.90 ns	1700400.83 ns
9	799.88 ns	3205805.06 ns
10	1644.70 ns	7070338.01 ns
11	7151.90 ns	14413833.62 ns
12	15372.56 ns	32285171.03 ns
13	18514.16 ns	69974661.11 ns
14	36287.44 ns	162460117.1 ns
15	77486.74 ns	336447609.9 ns

Table 2 Comprison with SAGE. In each case, the timings are averaged over 1000 runs.

different implementations (i.e., C and SAGE). The table shows the running time for $n = 3$ to $n = 15$. For $n = 16$ and 17, the running time for SAGE was significantly slower and for $n = 18$, SAGE had actually failed to compute the product. Note here that, although the program corresponding to the given input polynomials for $n = 18$ failed to compute in case of SAGE, this is not true for every input. In fact, one observes that the algorithm used in SAGE depends not only on the number of variables but also the on size of polynomials it is multiplying. SAGE has no problem multiplying two polynomials if the polynomials are sparse, even for number of variables much higher than 18.

4.1 Multiplying Sparse Polynomials

For sparse implementation, a monomial is represented by a δ-bit word and two polynomials are given as two arrays A and B of monomials. Multiplication of two polynomials corresponds to the bit-wise OR of the corresponding δ-bit words. Suppose we want to multiply two sparse polynomials p with l_q monomials and q with l_q monomials. For our sparse implementation (let us call it the quadratic implementation), we take the input arrays A and B and OR every element of array A with that of array B, and store them in another array C. The array C is then sorted using a non recursive (the process stack is simulated internally) implementation of randomized quick sort. Repetitions are removed by either deleting the monomial (if its number of repetitions is even) or replacing all the entries by just one entry (if the number of repetitions is odd).

Experiments were done to compare the speeds of SAGE for sparse polynomials with that of quadratic implementation. The experimental results not only show that the algorithm used by SAGE is slower than the quadratic implementation but also suggests that the SAGE algorithm depends both on the sizes of A and B (i. e., l_p and l_q) and the number of variables involved. But

the quadratic implementation only depends on l_p and l_q. For example to multiply two polynomials each with 1000 monomials SAGE took 7.43 seconds for $n = 30$ and 34 seconds for $n = 63$, whereas for the quadratic implementation it took 0.17 seconds for bot $n = 30$ and $n = 63$.

Based on experimental results, we also found that if $l_p l_q < 2^{n-\alpha}$, then the quadratic algorithm performs better than $\text{MultANF}_{2^\alpha}$, where $\alpha = 3, 5, 6$.

5 Conclusion

In this paper we have proposed a new non-recursive algorithm MultANF_w, which multiplies two Boolean functions in their ANF's. MultANF_w tries to use the w-bit word arithmetic, if the architecture supports it. With this in mind, three variants of MultANF_w are proposed for $w = 8, 32$ and 64. We show that the 64-bit implementation is better than the other two. A detailed comparison of MultANF_w with a sparse implementation tells us, when one should switch from the sparse implementation to the dense implementation, i.e., MultANF_w. Lastly, a comparison between our implementations (sparse and dense implementations) with that of the software package SAGE shows that, our implementations are faster than SAGE.

The MultANF_{64} algorithm is used to symbolically compute TRIVIUM. This is still a work in progress. We wish to do a thorough structural analysis of the output polynomials of TRIVIUM and conduct different randomness tests on it.

References

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians, 2001. `cr.yp.to/papers/m3.pdf`.

[BGTZ08] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in GF(2)[x]. In van der Poorten and Stein [vdPS08], pages 153–166.

[Bod07] Marco Bodrato. Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In Carlet and Sunar [CS07], pages 116–133.

[Buc98] B. Buchberger. An algorithmic criterion for the solvability of a system of algebraic equations. *Gröbner Bases and Applications*, 251:535–545, 1998.

[Buc06] B. Buchberger. Bruno buchbergers phd thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of Symbolic Computation*, 41(3):475–511, 2006.

[BZ] M. Bodrato and A. Zanoni. Karatsuba and toom-cook methods for multivariate polynomials. `www.emis.de/journals/AUA/ictami2011/Paper1-Ictami2011.pdf`.

[CM03] N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. *Advances in CryptologyEUROCRYPT 2003*, pages 644–644, 2003.

[CS07] Claude Carlet and Berk Sunar, editors. *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*, volume 4547 of *Lecture Notes in Computer Science*. Springer, 2007.

[Dal06] D.K. Dalai. *On some necessary conditions of boolean functions to resist algebraic attacks*. PhD thesis, Ph D thesis, Indian Statistical Institute, Kolkata, India, 2006.

[DCP] Christophe De Canniere and Bart Preneel. Trivium-specifications. estream, ecrypt stream cipher project, report 2005/030 (2005).

[Fau99] J.C. Faugere. A new efficient algorithm for computing gröbner bases (F_4). *Journal of pure and applied algebra*, 139(1):61–88, 1999.

[Fau02] J.C. Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (F_5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 75–83. ACM, 2002.

[FSGL07] Haining Fan, Jiaguang Sun, Ming Gu, and Kwok-Yan Lam. Overlap-free karatsuba-ofman polynomial multiplication algorithms. *IACR Cryptology ePrint Archive*, 2007:393, 2007.

[Gue11] Shay Gueron. Software optimizations for cryptographic primitives on general purpose x86_64 platforms. In *INDOCRYPT'11*, pages 399–400, 2011. (presentation available at `2011.indocrypt.org/slides/gueron.pdf`).

[Mat08] T. Mateer. *Fast Fourier transform algorithms with applications.* PhD thesis, Clemson University, 2008.

[Moe76] R.T. Moenck. Practical fast polynomial multiplication. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148. ACM, 1976.

[MPC04] W. Meier, E. Pasalic, and C. Carlet. Algebraic attacks and decomposition of boolean functions. In *Advances in Cryptology-EUROCRYPT 2004*, pages 474–491. Springer, 2004.

[Ose11] Ivan V. Oseledets. Improved n-term karatsuba-like formulas in gf(2). *IEEE Trans. Computers*, 60(8):1212–1216, 2011.

[O07] S. ONeil. Algebraic structure defectoscopy. In *Special ECRYPT Workshop–Tools for Cryptanalysis*, 2007.

[PMS+98] V. Pless, FJ MacWilliams, NJA Sloane, RE Blahut, and RJ McEliece. Introduction to the theory of error-correcting codes, 3rd. 1998.

[vdPS08] Alfred J. van der Poorten and Andreas Stein, editors. *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*, volume 5011 of *Lecture Notes in Computer Science*. Springer, 2008.

[Zan] Alberto Zanoni. Iterative Karatsuba for multivariate polynomial multiplication. `http://bodrato.it/papers/zanoni/`.