On Implementation of Quadratic and Sub-Quadratic Complexity Multipliers using Type II Optimal Normal Bases

Ayad F.Barsoum and M.Anwar Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada.

Abstract. Finite field arithmetic has received a considerable attention in the current cryptographic applications. Many researchers have focused on finite field multiplication due to its importance in various cryptographic operations. Moreover, finite field multiplication can be considered as a cornerstone for elliptic curve cryptosystems. Fan and Hasan [1] introduced a new sub-quadratic computational complexity approach for finite field multiplication. It is based on Toeplitz matrix-vector products. In this paper we consider efficient implementation of this approach on general purpose processors using Type II Optimal Normal Basis (ONB II). To this end, a memory and time efficient implementation scheme is proposed for the Fan and Hasan approach. Also, in this paper we provide a modified version of the best quadratic complexity multiplication algorithm due to Reyhani-Masoleh [2]. The proposed modification reduces the number of OR and SHIFT instructions by 50% and the number of AND instructions by about 25%. We simulate the implementation on three different architectures and present the results. Furthermore, we present an idea to fully parallelize the implementation of the Fan and Hasan scheme.

Keywords: Finite fields, normal bases, type II optimal normal bases, sub-quadratic complexity multiplication

1 Introduction

Finite or Galois field GF(2^m) can be viewed as an m-dimensional vector space, where each component is either 0 or 1. Arithmetic operations (addition, subtraction, multiplication, and inversion) over the finite fields have several applications in cryptography and error control coding [3]. These different applications require efficient algorithms to perform field operations. It is noted that there is a strong relation between the performance of the field operations and the representation of the field elements [4]. Despite there are many representations for the field elements, the two most common representations are the polynomial basis and the normal basis representations [5].

Among the field operations, the multiplication operation has received a considerable attention because it is very time-consuming and it represents the backbone for different cryptographic operations. The complexity of this operation depends on the representation of the field elements. Many research papers have focused on normal basis representation [6], [7], [8], [9], [10], and [4]. This focus on normal basis representation is due to the fact that the squaring operation in a normal basis is almost free, especially in hardware implementation. Squaring in a normal basis is just a *cyclic* shift operation. For example, if $A = (a_0, a_1, \dots, a_{m-1}) \in \mathrm{GF}(2^m)$, $\mathrm{a}_i \in \{0, 1\}$, and A is represented in the normal basis then $A^2 = (a_{m-1}, a_0, a_1, \dots, a_{m-2})$.

Although the squaring in normal basis is almost free, the multiplication in arbitrary normal basis is time-consuming [5]. Therefore, the recent trend is to concentrate on a special class of normal basis called *Gaussian normal bases*. A normal basis exists for every finite field GF(2^m) where m is a positive integer. On the other hand, a *Gaussian normal basis* exists only if m is not divisible by 8 [5]. There are two types of *Gaussian normal bases* which have the most efficient multiplication rule over all normal bases. These two types are called Type I and Type II Optimal Normal Bases(ONBs) [11]. In some cryptosystems, Type I ONB is avoided for security reasons [1]. One of the common methods for ONB multiplication is the Massey-Omura algorithm [6] with computational complexity $O(m^2)$, while the computational complexity for multiplication using an arbitrary normal basis is greater than $O(m^2)$ [1].

Many schemes and algorithms have been proposed to provide multiplication over GF(2^m) with computational complexity less than $O(m^2)$ (see [12], [13] and [14]). Fan and Hasan [1] proposed a sub-quadratic complexity scheme based on Toeptiz matrix-vector products. The computational complexities of their approach is $O(m^{1.58})$ and $O(m^{1.63})$ for *two-way* and *three-way* splittings, respectively. On the other hand, the scheme proposed by Reyhani-Masoleh [2] is one of the best quadratic schemes that have been proposed. A comparable work to Reyhani-Masoleh is done by Dahab et al. [7]. Their algorithm, i.e., Algorithm 4 of [7] is similar to Reyhani-Masoleh's algorithm. Then, it requires the same number of operations and memory requirements as the one proposed by Reyhani-Masoleh [2]. In this paper we will focus on Reyhani-Masoleh's work [2].

In this paper we consider efficient implementation of both approaches [1] and [2] and compare their memory requirements and timing performance. To this end, a memory and time efficient implementation scheme is proposed for the Fan and Hasan approach[1]. In our work, we also provide a modified version of the Reyhani-Masoleh algorithm and show that the former is faster than the latter. Our analysis and timing results show that the software implementation of the Fan and Hasan approach is faster than those of the original Reyhani-Masoleh algorithm and its modification we have proposed. Moreover, the memory requirement of the Fan and Hasan approach is less than that of the Reyhani-Masoleh algorithm. Also in this paper, we identify a way to fully parallelize the implementation of the Fan and Hasan scheme.

The rest of the paper is organized as follows: in Section 2 we review the conventional normal basis multiplication, and in Section 3 we present the quadratic and sub-quadratic multiplication schemes proposed by Reyhani-Masoleh [2] and Fan and Hasan [1] respectively. The modified Reyhani-Masoleh multiplication algorithm is provided in Section 4. In Section 5 we present our software implementation of the field multiplication schemes. Comparisons of memory requirements and timing results are also presented in Section 5. Concluding remarks and future work are given in Section 6.

2 Conventional Normal Basis Multiplication over GF(2^m)

A normal basis of GF(2^m) over GF(2) is a set N = $\{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\}$ whose elements are over GF(2^m) and linearly independent. Any element A $\in$ GF(2^m) can be represented as : A $= \sum_{i=0}^{m-1} a_i\beta 2^i$, , where $a_i \in$ GF(2). Field multiplication in normal basis is usually carried out using a multiplication matrix, which is an *m-by-m* matrix M with each entry $M_{ij} \in$ GF(2). The multiplication matrix M can be computed by the following rule:

$$\begin{aligned} M = \underline{\beta}^T \cdot \underline{\beta} &= (\beta, \beta^2, \ldots \beta^{2^{m-1}})^T \cdot (\beta, \beta^2, \ldots \beta^{2^{m-1}}) \\ &= \begin{pmatrix} \beta & \beta^2 & \beta^5 & \ldots & \beta^{2^{m-1}+1} \\ & & \vdots & & \\ \beta^{2^{m-1}+1} & \beta^{2^{m-1}+2} & \beta^{2^{m-1}+2^2} & \ldots & \beta^{2^{m-1}+2^{m-1}} \end{pmatrix} \end{aligned} \quad (1)$$

Details on how to compute the matrix M can be found in [5]. For field multiplication, the complexity of the basis $\{\beta_i\}$, denoted by C_β, is defined by the number of 1's in M. It is well known that $C_\beta \geq 2m-1$ [4]. Let $A = (a_0, a_1, \dots, a_{m-1})$, and $B = (b_0, b_1, \dots, b_{m-1})$ be two elements represented in the normal basis, and let $C = (c_0, c_1, \dots, c_{m-1})$ be their product. Then each coordinate c_k is computed as follows: *(the addition in the subscript is done mod m)* [4].

$$c_k = (a_k,\, a_{k+1}, \dots, a_{k+m-1})M(b_k,\, b_{k+1}, \dots, b_{k+m-1})^T. \tag{2}$$

This means that the coordinates of the product C are subsequently produced by shifting the coordinates of A and B. This conventional scheme uses bit-level multiplication. As mentioned earlier, the computational complexity of the multiplication using the conventional normal basis is ***greater*** than $O(m^2)$. The following is the multiplication algorithm using an arbitrary normal basis $\{\beta_i\}$[3].

Algorithm 1 Arbitrary NB multiplication Algorithm for GF(2^m)

```
Input: $A = (a_0, a_1, \dots, a_{m-1}), B = (b_0, b_1, \dots, b_{m-1}) \in GF(2^m)$ and matrix $M$
Output: $C = (c_0, c_1, \dots, c_{m-1}) = AB$
for $k$ = 0 to m-1 do
    $c_k = (a_k, a_{k+1}, \dots, a_{k+m-1})\ M\ (b_k, b_{k+1}, \dots, b_{k+m-1})^T$
    (the addition in the subscript is done mod m)
end
Output $C = (c_0, c_1, \dots, c_{m-1})$
```

The multiplication matrix M can be expressed as $M = M_0\beta + M_1\beta^2 + M_0\beta + \cdots + M_{m-1}\beta^{2^{m-1}}$. Entries of $M_i \in$ GF(2). Let $a_{i(j,k)}$ be the entry in row j and column k of the matrix M_i, then

$$a_{i(j,k)} = \begin{cases} 0 & \text{if entry } M_{j,k} \text{ of matrix } M \text{ does not contain } \beta^{2^i} \\ 1 & \text{if entry } M_{j,k} \text{ of matrix } M \text{ contains } \beta^{2^i} \end{cases} \tag{3}$$

Then we can compute the coordinates of the product C by the following:

$$c_{m-1-k} = \underline{(a^{2^k})} \cdot M_{m-1} \cdot \underline{(b^{2^k})^T} \qquad k = 0, 1, \dots, m-1 \tag{4}$$

Equation (4) corresponds to the well known Massey-Omura normal basis multiplication algorithm [6].

3 Multiplication Schemes using Type II ONB

3.1 Quadratic Scheme for Multiplication using ONB II

Reyhani-Masoleh [2] proposed his scheme based on some observations of the multiplication matrix M. First, it is well-known that M is symmetric. Second, all diagonal entries of M are zeros except the last one. Third, for type II ONB over GF(2^m) where m is odd, there is only one non-zero entry in row 0 or column 0. For other rows (or columns), the number of 1's in each row (or column) is two. Based on these observations Reyhani-Masoleh [2] proposed to store the indices of the non-zero columns of M instead of storing the whole M. Since the location of the single 1 of row 0 of M is fixed, he needed to store those in rows 1 up to $m-1$. Reyhani-Masoleh [2] used an $(m-1) \times 2$ matrix R whose entries contain the column indices of 1's in row i of M. Furthermore, in the multiplication matrix M for GF(2^m)(where m is odd), $row(m-i)$ is the i-fold left cyclic shift of $row(i), i = 1, 2, \dots, \frac{m-1}{2}$. Therefore, only the first half of the matrix R is needed to be stored and is denoted as R' which is an $\frac{(m-1)}{2} \times 2$ matrix. In addition, Reyhani-Masoleh added more improvements for normal basis multiplication by using vector level multiplication instead of using bit level multiplication as in Equation (2).

One of the requirements of the algorithm presented in [2] is to compute the different cyclically shifted versions of the element B. Although the *cyclic* shift operation is almost free in hardware, this is not the case in software. So, instead of storing the different cyclically shifted versions of the element B and thus requiring m^2 bits or $m\lceil\frac{m}{w}\rceil$ words (w is the word length of the underlying processor), Reyhani-Masoleh used the method presented in [4] and [15] which requires relatively less memory. In this method the element B is stored as an array in which $B[i] = (b_i, b_{i+1}, \ldots, b_{i+w-1 \; mod \; m}), \;\; i : 0 \ldots m-1$. A cyclic shift of the element B can be generated by *reading* $\lceil\frac{m}{w}\rceil$ words from memory. For example if we are working with GF(2^{163}), the element B can be represented as shown in Fig.1 [2].

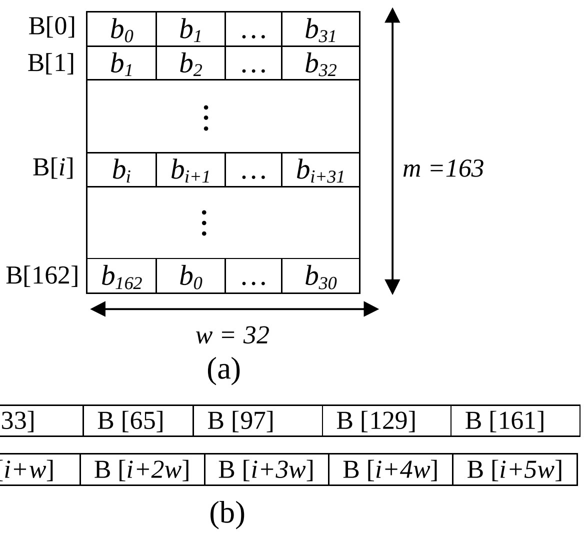

B[0]
b_0 b_1 … b_{31}
B[1]
b_1 b_2 … b_{32}
B[i]
b_i b_{i+1} … b_{i+31}
B[162]
b_{162} b_0 … b_{30}
m =163
w = 32
(a)
B_1
B[1] B [33] B [65] B [97] B [129] B [161]
B_i
B [i] B [$i+w$] B [$i+2w$] B [$i+3w$] B [$i+4w$] B [$i+5w$]
(b)

Fig. 1. (a) Precompute arrays for $B[i]$ (m=163 and w = 32).(b) Generating B_1 and B_i using the arrays.

Note that $B[i]$ is the i^{th} unit (w-bits) of the element B, while B_i is the i^{th} cyclically shifted version of the element B. For example, B_1 is the element B shifted one time, B_2 is the element B shifted two times and so on. Therefore it requires m words (mw bits) to store the element B using the previous method. Full details about Reyhani-Masoleh's work can be found in [2]. The following is the algorithm proposed by Reyhani-Masoleh [2].

Algorithm 2 Reyhani-Masoleh Algorithm [2] for GF(2^m) multiplication using type II ONB , *m is odd*

```
Input: $A = (a_0, a_1, \dots, a_{m-1}), B = (b_0, b_1, \dots, b_{m-1}) \in GF(2^m)$ and matrix $R'$
Output: $C = (c_0, c_1, \dots, c_{m-1}) = AB$
Precompute arrays for $B_i$
Initialize $L = A$, $C = A \odot B_1$
for $i = 1$ to $\frac{m-1}{2}$ do
    1. $L = L \lll 1$
    2. $S = B_{R'(i-1,0)} \oplus B_{R'(i-1,1)}$
    3. $C = C \oplus (L \odot S)$
    4. $R = (A \odot S) \ggg i$
    5. $C = C \oplus R$

end
Output $C = (c_0, c_1, \dots, c_{m-1})$
```

3.2 Sub-Quadratic Scheme for Multiplication using ONB II

The work of Fan and Hasan [1] includes a new scheme for GF(2^m) multiplication using type II ONB. The proposed scheme has a sub-quadratic computational complexity. It is based on Toplitz matrix-vector products. An $m \times m$ Toplitz matrix is a matrix with the property that $m_{i,j} = m_{i-1,j-1}$.

For $m = 2^i (i > 0)$, let T be an $m \times m$ Toplitz matrix and V an $m \times 1$ column vector over GF(2). Then the following formula can be used to compute the Toeplitz matrix-vector product TV [16]

$$TV = \begin{pmatrix} T_1 & T_0 \\ T_2 & T_1 \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \end{pmatrix} = \begin{pmatrix} P_0 + P_2 \\ P_1 + P_2 \end{pmatrix} \quad (5)$$

where T_0, T_1 and T_2 are $(m/2) \times (m/2)$ matrices and are individually in Toeplitz form, V_0 and V_1 are $(m/2) \times 1$ column vectors, $P_0 = (T_0 + T_1)V_1, P_1 = (T_1 + T_2)V_0$, and $P_2 = T_1(V_0 + V_1)$.

For $m = 3^i (i > 0)$, the following formula can be used to compute the Toeplitz matrix-vector product TV [16]

$$TV = \begin{pmatrix} T_2 & T_1 & T_0 \\ T_3 & T_2 & T_1 \\ T_4 & T_3 & T_2 \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \end{pmatrix} = \begin{pmatrix} P_0 & P_3 & P_4 \\ P_1 & P_3 & P_5 \\ P_2 & P_4 & P_5 \end{pmatrix} \quad (6)$$

where $T_i (0 \le i \le 4)$ are $(m/3) \times (m/3)$ Toeplitz matrices,

$$\begin{aligned} P_0 &= (T_0 + T_1 + T_2)V_2, & P_3 &= T_1(V_1 + V_2), \\ P_1 &= (T_1 + T_2 + T_3)V_1, & P_4 &= T_2(V_0 + V_2), \\ P_2 &= (T_2 + T_3 + T_4)V_0, & P_5 &= T_3(V_0 + V_1). \end{aligned}$$

Formulas (5) and (6) can be used recursively to compute the Toeplitz matrix-vector product TV. Formula (5) is called the *two-way* splitting method, while formula (6) is called the *three-way* splitting method.

It is well known that type II ONB does not exist for all m. For type II ONB to exist over GF(2^m) the following conditions must be satisfied [11]:

- $2m + 1$ is prime,
- 2 is primitive in $\mathbb{Z}_{2m+1}$ or
- $2m + 1 \equiv 3 \pmod 4$ and 2 generates the quadratic residues in $\mathbb{Z}_{2m+1}$.

There are adequate numbers of optimal normal bases that are of practical interest. For example, there are 430 values of $m \leq 2000$ for which ONBs exist[3]. We can apply the Toeplitz matrix-vector approach using any m.

Now we will illustrate how to use type II ONB to construct a sub-quadratic scheme. Let $N' = \{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\}$ be type II ONB of GF(2^m). We can write $\{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\} = \{\beta_1, \beta_2, \ldots, \beta_m\}$[1]. Therefore, $N = \{\beta_1, \beta_2, \ldots, \beta_m\}$ can be considered to be a basis of GF(2^m). Given a field element A represented in the above two bases, i.e., $A = \sum_{i=0}^{m-1} a'_i \beta^{2^i} = \sum_{i=1}^{m} a_i \beta_i$ where a'_i and a_i are the coordinates of the element A in basis N' and N respectively, the coordinate transformation is given as follows [10]: $a_j = a'_i$

$$j = \begin{cases} k & k \in [1, m] \\ (2m+1) - k & k \in [m+1, 2m] \end{cases}, where\, k = 2^i (mod\, 2m+1) \quad i : 0 \cdots m-1 \tag{7}$$

This coordinate transformation is a permutation between the coordinates.

Given two elements A and B in basis N, where $\underline{a} = (a_1, a_2, \ldots, a_m)$ and $\underline{b} = (b_1, b_2, \ldots, b_m)$ are the coordinates of A and B respectively. Then the multiplication $C = AB$ can be computed as follows [1]:

$$\underline{c} = Z_1 \underline{a}^T + Z_2 \underline{a}^T \tag{8}$$

where $\underline{c} = (c_1, c_2, \ldots, c_m)$ is an $m \times 1$ vector corresponding to the coordinates of the element C, $\underline{a}^T$ is the transpose of $\underline{a}$, and

$$Z_1 = \begin{pmatrix} b_2 & b_3 & b_4 & \cdots & b_{m-1} & b_m & b_m \\ b_3 & b_4 & b_5 & \cdots & b_m & b_m & b_{m-1} \\ b_4 & b_5 & b_6 & \cdots & b_m & b_{m-1} & b_{m-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ b_{m-1} & b_m & b_m & \cdots & b_5 & b_4 & b_3 \\ b_m & b_m & b_{m-1} & \cdots & b_4 & b_3 & b_2 \\ b_m & b_{m-1} & b_{m-2} & \cdots & b_3 & b_2 & b_1 \end{pmatrix} \quad Z_2 = \begin{pmatrix} 0 & b_1 & b_2 & \cdots & b_{m-3} & b_{m-2} & b_{m-1} \\ b_1 & 0 & b_1 & \cdots & b_{m-4} & b_{m-3} & b_{m-2} \\ b_2 & b_1 & 0 & \cdots & b_{m-5} & b_{m-4} & b_{m-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ b_{m-3} & b_{m-4} & b_{m-5} & \cdots & 0 & b_1 & b_2 \\ b_{m-2} & b_{m-3} & b_{m-4} & \cdots & b_1 & 0 & b_1 \\ b_{m-1} & b_{m-2} & b_{m-3} & \cdots & b_2 & b_1 & 0 \end{pmatrix}$$

It is clear that, Z_1 is not in a Toeplitz form, it is a Hankel matrix. A Hankel matrix is a matrix with the property that $Z(i,j) = Z(i-1, j+1)$. To convert a Hankel matrix to a Toeplitz matrix, swap the columns Z_i and Z_{m-1-i}, $i = 0, 1, \ldots, m/2$. So, to compute $Z_1 \underline{a}^T$ in Equation (8), Z_1 will be converted to a Toeplitz form and then multiplied by $\underline{a}^{rev} = (a_m, a_{m-1}, \ldots, a_1)^T$, where $\underline{a}^{rev}$ is the reverse of $\underline{a}^T$. Therefore, we have two Toeplitz matrix-vector products to compute $C = AB$. The computational complexities of Fan and Hasan scheme are presented in Table 1 [1].

Table 1. Computational complexities of the Fan and Hasan multiplication scheme for $m = b^i$

ONB	b	#AND	#XOR	Gate delay
Type II	2	$2m^{\log_2 3}$	$11m^{\log_2 3} - 12m + 1$	$(2\log_2 m + 1)T_X + T_A$
	3	$2m^{\log_3 6}$	$\frac{48}{5}m^{\log_3 6} - 10m + \frac{2}{5}$	$(3\log_3 m + 1)T_X + T_A$

From the previous table we can see that #AND and #XOR gates for $b = 2$ is less than that number for $b = 3$. On the other hand, the gate delay for $b = 3$ is slightly less than that delay for $b = 2$ (This will impact the timing results of the software implementation as will be shown in section 5).

4 Modified Reyhani-Masoleh Multiplication Algorithm

In this section, we propose a modification to the field multiplication algorithm presented by Reyhani-Masoleh[2]. In this modification we remove the first *cyclic* shift left operation from the *for* loop of Algorithm 2 and thus increasing the speed of the algorithm. First we introduce the software implementation of the *cyclic* shift operation and the number of instructions needed to perform this operation, then we will propose our modification. It should be noted that, most software implementations of finite field arithmetic are done using high level programming languages like C, C++, and C# (see [7], [4], [2], and [17]) that, unlike most assembly languages, do not directly support *cyclic* shift operations.

4.1 Cyclic Shift Operations of GF(2^m) Elements

Shift operation in hardware is almost free. It can be done in one clock cycle. But this is not the case in software implementation. The shift operations are relatively costly when it comes to software implementation. First, we should note that the shift operations used in the normal basis field multiplication are ***cyclic*** shift not just shift, i.e. rotate operations. Second, the shift instruction in any programming language only shifts a w-bits data item at a time (w is the word length of the underlying processor). Therefore, to shift a data item of size grater than w we need some extra processing. In addition, to obtain the carry bit of the shift operation, some masks are needed. It is important to note that, there is no carry flag in high level programming languages like C and C#.

To explain the cyclic shift operation in software, we would like to show how field elements are represented in our implementation. An element $A \in \mathrm{GF}(2^m)$, can be represented by a vector corresponding to its coordinates $(a_0, a_1, \ldots, a_{m-1})$, each coordinate a_i is one bit. So, A can be represented as an array of size $\lceil \frac{m}{w} \rceil$. For example, let us consider an element $A \in \mathrm{GF}(2^m)$, i.e. $A = (a_0, a_1, \ldots, a_{232})$. If we are dealing with 32-bit architecture, the following figure shows the element A before and after the *cyclically* shift left operation.

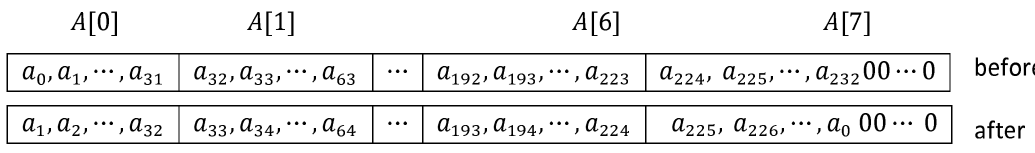

$A[0]$
$A[1]$
$A[6]$
$A[7]$
$a_0, a_1, \cdots, a_{31}$
$a_{32}, a_{33}, \cdots, a_{63}$
$\cdots$
$a_{192}, a_{193}, \cdots, a_{223}$
$a_{224}, a_{225}, \cdots, a_{232} 00 \cdots 0$
before
$a_1, a_2, \cdots, a_{32}$
$a_{33}, a_{34}, \cdots, a_{64}$
$\cdots$
$a_{193}, a_{194}, \cdots, a_{224}$
$a_{225}, a_{226}, \cdots, a_0\ 00 \cdots 0$
after

Fig. 2. An element $A \in \mathrm{GF}(2^{233})$, before and after cyclic shift left.

Where each component $A[i]$ is 32 bits (*unsigned int* in C#). Here we note that the last component ($A[7]$) is less than 32 bits, so the remaining bits($w\lceil \frac{m}{w} \rceil - m$) are zeros. In our implementation we consider a_0 to be the most significant bit of $A[0]$ and a_{31} to be the least significant bit and this is the case in all other components $A[i]$. To cyclically shift the element A to the left, we need to shift all the components starting from $A[7]$ down to $A[0]$. The carry bit, the most significant bit of $A[7]$, *i.e.* a_{224}, should be shifted to the least significant position of $A[6]$ and the carry from $A[6]$ should be shifted to $A[5]$ and so on. For these operations to be implemented in software, we need to know the values of the most significant bits of each component $A[i]$ before the shifting is done. In our implementation, we use ***carryMask*** = `0x80000000` to know the value of the most significant bit of each component $A[i]$. Moreover, we use another mask which is ***rotateMask*** $=$ `0x800000` to rotate the most significant bit of $A[0](a_0)$ to the position of the least significant bit of $A[7](a_{232})$. The operations of the cyclic shift left in 32-bit architecture for an element $A \in \mathrm{GF}(2^m)$, are shown in the following piece of code:

```
int carry = 0;
For(int i = Math.Ceiling(m / w)-1 i>=0; i--)
{
  if(carry != 0)  // checking previous carry
  {
    carry = A[i] & 0x80000000;  //calculate new carry
    A[i] = (A[i] << 1) | 0x1;   //shift and insert previous carry
  }
  else
  {
    carry = A[i] & 0x80000000;  //calculate new carry
    A[i] =  A[i] << 1;         //only shift
  }
}
//rotating the most significant bit.
If(carry != 0)
  A[Math.Ceiling(m / w)-1] =  A[Math.Ceiling(m / w)-1] | 0x800000;
```

From the previous code we can compute the number of instructions needed to perform ***ONE*** *cyclic* shift for an element $A \in \mathrm{GF}(2^m)$. On average it requires, $\lceil \frac{m}{w} \rceil$ SHIFT + $\lceil \frac{m}{w} \rceil$ AND + $\lceil \frac{m}{w} \rceil/2$ OR instructions. We call the number of instructions required for the cyclically shift operation CycSHIFT. To the best of our knowledge, there is no other ***optimized*** way to cyclically shift an element $A \in \mathrm{GF}(2^m)$ using high level programming language. Thus, the software implementation of the cyclic shift operation is not free as in hardware implementation. The same thinking can be used in the cyclic shift right operation but using different masks.

4.2 Proposed Modification to Reyhani-Masoleh Algorithm

Taking a second look at Algorithm 2, we found that it can be improved by removing the first cyclic shift operation from the *for* loop (step 1 in the *for* loop: $L = L \lll 1$). As we have showed in the previous section, the cyclic shift operation is time consuming in software implementation. In total, step 1 of the *for* loop in Algorithm 2 requires on average $(\frac{m-1}{2})(\lceil \frac{m}{w} \rceil SHIFT + \lceil \frac{m}{w} \rceil AND + \lceil \frac{m}{w} \rceil/2\ OR)$ instructions which affects the running time of the algorithm. We can remove all this overhead at the expense of a few more memory bits.

The variable L in Algorithm 2 is a copy of the element A. At each iteration of the loop, L is cyclically shifted left one time. It means that at each iteration we cyclically shift the element A once. In our modification, we will use the method presented in [4] and [15] to store different cyclically shifted versions of the element A. So, both elements A and B will be stored the same way. They will be pre-computed before the main *for* loop. In addition, instead of using L in the third step of the *for* loop($C = C \oplus (L \odot S)$), we will use A_i. This modification to the Reyhani-Masoleh algorithm requires $m(w-1)$ *extra* bits. On the positive side, the modified version eliminates the cyclic shift left operations and gives a speed up compared to the original version of algorithm. The proposed modification is presented below

Algorithm 3 Modified Reyhani-Masoleh Algorithm for GF(2^m) multiplication using ONB type II, *m is odd*

```
Input: $A = (a_0, a_1, \dots, a_{m-1}), B = (b_0, b_1, \dots, b_{m-1}) \in GF(2^m)$ and matrix $R'$
Output: $C = (c_0, c_1, \dots, c_{m-1}) = AB$
Precompute arrays for $A_i$ and $B_i$
Initialize $C := A \odot B_1$
for $i = 1$ to $\frac{m-1}{2}$ do
    1. $S = B_{R'(i-1,0)} \oplus B_{R'(i-1,1)}$
    2. $C = C \oplus (A_i \odot S)$
    3. $R = (A \odot S) \gg i$
    4. $C = C \oplus R$
end
Output $C = (c_0, c_1, \dots, c_{m-1})$
```

In the above algorithm, three different operations are used: AND($\odot$), XOR ($\oplus$) and *cyclic* shift($\gg$). In any programming language, AND and XOR operations can be implemented using $\lceil \frac{m}{w} \rceil$ AND and XOR instructions. Thus, the initialization step ($C := A \odot B_1$) requires $\lceil \frac{m}{w} \rceil$ AND instructions, step 1 of the *for* loop requires $\lceil \frac{m}{w} \rceil$ XOR instructions, step 2 requires $\lceil \frac{m}{w} \rceil$ AND + $\lceil \frac{m}{w} \rceil$ XOR instructions, step 3 requires $\lceil \frac{m}{w} \rceil$ AND + CycSHIFT instructions, and step 4 requires $\lceil \frac{m}{w} \rceil$ XOR instructions. The number of instructions needed for the original and the modified Reyahni-Masoleh multiplication algorithms is summarized in Table 2.

Table 2. Comparing the original and the modified Reyahi-Masoleh schemes in terms of # of instructions

	# of instructions			
	XOR	AND	SHIFT	OR
Original [2]	$\frac{3}{2}(m-1)\lceil \frac{m}{w} \rceil$	$2(m-1)\lceil \frac{m}{w} \rceil + \lceil \frac{m}{w} \rceil$	$(m-1)\lceil \frac{m}{w} \rceil$	$\frac{1}{2}(m-1)\lceil \frac{m}{w} \rceil$
Modified [2]	$\frac{3}{2}(m-1)\lceil \frac{m}{w} \rceil$	$\frac{3}{2}(m-1)\lceil \frac{m}{w} \rceil + \lceil \frac{m}{w} \rceil$	$\frac{1}{2}(m-1)\lceil \frac{m}{w} \rceil$	$\frac{1}{4}(m-1)\lceil \frac{m}{w} \rceil$
Reduction Percent	0%	$\approx 25\%$	50%	50%

Table 2 shows that the modified version of the Reyahni-Masoleh algorithm reduces the number of OR and SHIFT instructions by 50%, and the number of AND instructions by $\approx 25\%$. This improvement speeds up the running time of the software implementation of the multiplication algorithm. Performance analysis and timing results of the original and the modified Reyahni-Masoleh will be presented in section 5.

5 Software Implementation

In our work we develope the software implementation for the original Reyhani-Masoleh, the modified Reyhani-Masoleh, and the Fan and Hasan schemes. We compare the timing performance and memory requirements of the three algorithms. In our implementation, we simulate three different architectures: 32-bit, 16-bit, and 8-bit architectures. As we have showed previously, the field elements are represented in our implementation as vectors (arrays). An element $A \in$ GF(2^m), can be represented by a vector corresponding to its coordinates $(a_0, a_1, \dots, a_{m-1})$, where each coordinate a_i is one bit. Let w denote the word size of the underlying architecture, then the element A can be stored in $\lceil \frac{m}{w} \rceil$ words. For the 32-bit architecture, the elements of GF(2^m) are stored in *unsigned int* arrays (*unsigned int* in C# is a data field of size 32-bits), for the 16-bit architecture the elements are stored in $UInt16$ arrays, and for the 8-bit architecture the elements are stored in *byte* arrays. We use the XOR, AND, OR, and $SHIFT$ instructions available in the language to perform the XOR, AND, OR, and $SHIFT$ operations of the algorithms.

It is well known that type II ONB does not exist for every m [3]. National Institute of Standards & Technology (NIST) [18], recommended 10 finite fields: 5 prime fields and 5 binary fields. The recommended NIST binary fields are GF(2^{163}), GF(2^{233}), GF(2^{283}) , GF(2^{409}), and GF(2^{571}). Among the recommended NIST binary fields, ONB II only exists for GF(2^{233}). In our implementation we use some fields that are close to the recommended fields by NIST. Namely, we used GF(2^{173}), GF(2^{233}), GF(2^{281}), GF(2^{419}),and GF(2^{593}).

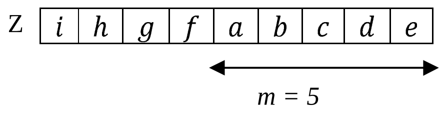
5.1 Memory Requirements

In this sub-section we analyze the memory requirements of the algorithm by Fan and Hasan [1] and the modified version of Reyhani-Masoleh multiplication scheme. As we have showed in section 4, the modified Reyhani-Masoleh uses the method presented in [4] and [15] to store the elements A and B and thus generating the cyclically shifted versions by reading $\lceil \frac{m}{w} \rceil$ word from memory. This method requires $2mw$ bits to store the shifted versions of the elements A and B. Moreover, the multiplication algorithm uses $\frac{m-1}{2} \times 2$ matrix R' to store the indices of non-zero column in the multiplication matrix M. Since each element in R' is an index between $0 \dots m-1$, the matrix R' requires $(m-1)\lceil \log_2(m) \rceil$ bits. Furthermore, the modified algorithm uses some other temporary variables: S, C, and R each of size m bits. In total, the modified Reyhani-Masoleh multiplication algorithm requires $3m + 2mw + (m-1)\lceil \log_2(m) \rceil$ bits. The original Reyhani-Masoleh requires $4m + mw + (m-1)\lceil \log_2(m-1) \rceil$ bits

On the other hand, the Fan and Hasan [1] scheme is based on the Toeplitz matrix-vector product. In our implementation, we represent the Toplitz matrix using only the first row and the first column of the matrix. The following matrix is an example of (5 $\times$ 5) Toeplitz matrix:

$$\begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{bmatrix} \qquad T(0,0) = T(1,1) = T(2,2) = T(3,3) = T(4,4) = a$$

This matrix can be represented using only the first row and the first column. The remaining rows can be obtained by shifting operations for the first row and first column. So, to store an $m \times m$ Toeplitz matrix, we just need $(2m-1)$ bits. From the previous (5 $\times$ 5) Toeplitz matrix, we can store the first row and first column in an array Z as:



The first row of the Toeplitz matrix is the first m bits of the Z array. To obtain the second row of the matrix, shift Z to the right one position and take the first m bits after shifting. Therefore, the scheme presented in [1] requires $(2m-1)$ bits to store Z_1 or Z_2 matrix in Equation (8). Of course, we do not need to store both of them at the same time. Since we can first compute

$Z_1\underline{a}^T$ then $Z_2\underline{a}^T$. So, only one matrix is needed to be stored at a time. Moreover, the Fan and Hasan [1] scheme requires m bits to store the element A. For the *two-way* splitting method some additional variables are used, namely P_0, P_1, and P_2 each of which is w-bits. In addition to the previous used variables, we use three additional variables: P_3, P_4, and P_5 of size w-bit each for the *three-way* splitting method. The result of the matrix-vector product is stored in an m-bit vector. In total, the Fan and Hasan approach requires $3w + 4m - 1$ bits and $6w + 4m - 1$ bits for the *two-way* and *three-way* schemes respectively.

The memory requirements of the Fan and Hasan algorithm is much less than that of the Reyhani-Masoleh algorithm. So, the Fan and Hasan scheme is more appropriate to be used in a constraint environment like smart cards. For smart cards with 16-bit microprocessor, the Fan and Hasan algorithm over GF(2^{233}) requires 123 bytes and 129 bytes for the *two-way* and the *three-way* splittings respectively.

5.2 Timing Results for field multiplication

To evaluate the performance of the Reyhani-Masoleh [2], the modified Reyhani-Masoleh, and the Fan and Hasan [1] schemes, we performed a series of experiments for type II ONB on Intel(R) Xeon (R) CPU with clock speed 2 GHZ and 3 GB RAM running Windows XP. The program was written in the C# language. The timing results were computed by averaging 10000 multiplications of random filed elements. As mentioned in the beginning of section 5, ONB II does not exist for every m. We used the following binary fields: GF(2^{173}),GF(2^{233}),GF(2^{281}),GF(2^{419}), and GF(2^{593}). Tables 2, 3, and 4 show the timing results of our simulations for the three architectures: 32-bit, 16-bit, and 8-bit architectures. The *speed-up*$(S_{Rey}) = \frac{T_{[2]} - T_{[2]modified}}{T_{[2]}} \times 100$ and the *speed-up*$(S_{FH}) = \frac{T_{[2]} - T_{[1]}}{T_{[2]}} \times 100$ are also presented.

Table 3. Timings for field multiplication over 32-bit architecture

Times in μs					Speed-ups		
m	[2]	Modified [2]	[1] 2-*way*	[1] 3-*way*	S_{Rey}	S_{FH} 2-*way*	S_{FH} 3-*way*
173	65.62	55.04	42.5	39.55	16.13%	35.24%	39.73%
233	115.28	95.64	70.59	67.76	17.04%	38.77%	41.23%
281	157.52	131.58	99.86	89.48	16.47%	36.61%	43.2%
419	349.67	286.03	206.25	199.64	18.2%	41.02%	42.91%
593	674.03	543.88	386.37	366.79	19.31%	42.68%	45.58%

Table 4. Timings for field multiplication over 16-bit architecture

Times in μs					Speed-ups		
m	[2]	Modified [2]	[1] 2-*way*	[1] 3-*way*	S_{Rey}	S_{FH} 2-*way*	S_{FH} 3-*way*
173	106.76	88.26	67.22	62.11	17.33%	37.04%	41.82%
233	189.16	155.98	120.15	109.37	17.54%	36.48%	42.18%
281	269.12	218.76	160.43	151.18	18.71%	40.39%	43.82%
419	589.57	481.61	360.62	334.42	18.31%	38.83%	43.28%
593	1169.32	944.81	688.48	633.77	19.2%	41.12%	45.8%

Table 5. Timings for field multiplication over 8-bit architecture

Times in μs					Speed-ups		
m	[2]	Modified [2]	[1] 2-*way*	[1] 3-*way*	S_{Rey}	S_{FH} 2-*way*	S_{FH} 3-*way*
173	197.71	162.81	121.81	118.44	17.65%	38.39%	40.09%
233	360.43	298.32	226.36	209.89	17.23%	37.2%	41.77%
281	523.09	427.83	326.69	297.47	18.21%	37.55%	43.13%
419	1153.52	930.65	703.85	638.05	19.32%	38.98%	44.69%
593	2369.21	1909.58	1395.71	1272.54	19.4%	41.09%	46.29%

Tables 3, 4, and 5 show that the timing results of the original Reyhani-Masoleh scheme [2] are greater than that of both the modified version and the Fan and Hasan [1] schemes. The speed ups are with respect to the Reyhani-Masoleh [2]. For the modified Reyahni-Masoleh, the speed ups range from 16% to 19%. This increase in the speed up is at the expense of a few more bits ($m(w-1)$ bits). The Fan and Hasan approach improves timings up to 46% speed up on different architectures. From the previous tables, we can see that the *three-way* splitting method of the Toeplitz matrix has timing results slightly less than that of the *two-way* splitting. This slight difference is due to the fact that the gate delay for the *three-way* splitting method is slightly less than the gate delay for the *two-way* splitting (refer to Table 1). This difference in the gate delay in hardware corresponds to a difference in the number of loop iterations in software. Figure 3 presents a graphical view of the timing results for 16-bit architecture. The x-axis represents the field size and y-axis represents the timings in μs.

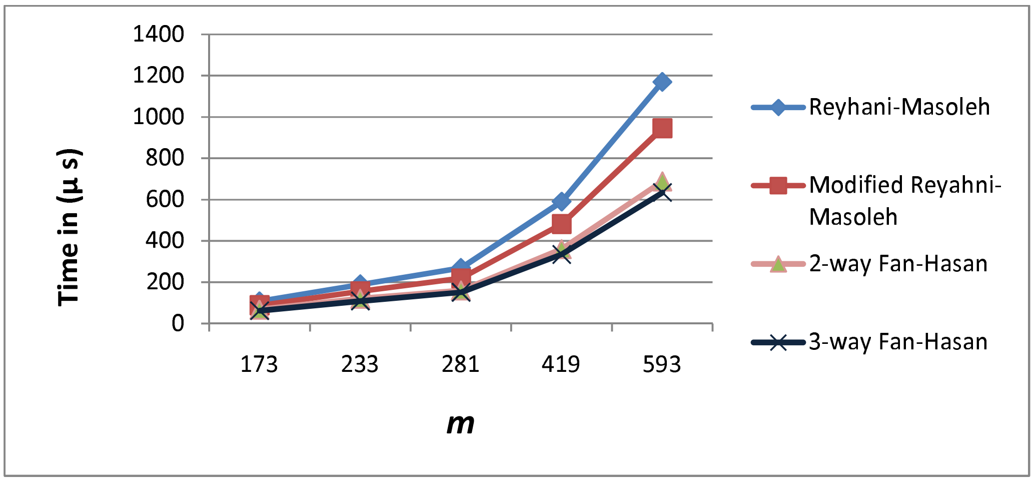


Fig. 3. Timing measurements for field multiplication over 16-bit architecture

As shown in Figure 3, the timing curve of the original Reyhani-Masoleh scheme is the highest curve, while the lowest timing curve is for the *three-way* Fan and Hasan approach. The timing curve for the modified Reyhani-Masoleh is lower than that for the original Ryahni-Masoleh. It is noted that the timing curve for the *three-way* Fan and Hasan is slightly lower than that curve for the *two-way* method.

5.3 Parallelization

A distinctive feature of the Fan and Hasan multiplication algorithm [1] is parallel processing. In Equation (8), the product $\underline{c}$ is computed as the sum of two matrix-vector products: $Z_1\underline{a}^T$ and $Z_2\underline{a}^T$. In our software implementation, we have computed this sum in serial fashion: first $Z_1\underline{a}^T$ then $Z_2\underline{a}^T$ and finally add them to get $\underline{c}$. Over parallel processors these two products can be computed simultaneously. One of the processors can be dedicated to compute $Z_1\underline{a}^T$, and the other to $Z_2\underline{a}^T$. In this case the timing results showed in section 5.2 of the Fan and Hasan [1] algorithm will be reduced by approximately 50% leading to much better performance.

Unfortunately, we could not do this parallelization on our platform since we do not have parallel processors. We tried to simulate this parallelization using multi-threading programming techniques. One thread is created to compute $Z_1\underline{a}^T$ and another thread to compute $Z_2\underline{a}^T$. Since there is no guarantee that both threads will be finished in the same instance of time, we used *semaphore* to block the fastest thread and waited until the lowest thread is done to add the results of the two products and finally get $\underline{c}$. Actually, creating and initializing threads plus using *semaphore* add more overhead on the running time. Moreover, the two threads share the same main memory. This sharing of memory adds some more overhead on the multi-threading technique. In order to obtain the best performance, the two matrix-vector products need to be computed in two processors running in parallel and using their own local memories. One should note that, in case of parallelization the memory requirements of the Fan and Hasan algorithm will be doubled. It will be approximately $6w + 8m - 2$ bits and $12w + 8m - 2$ bits for the *two-way* and the *three-way* splittings respectively (still less than Reyhani-Masoleh [2]).

6 Conclusion

In this paper we have presented some issues related to the implementation of two multiplication schemes. One of them is the sub-quadratic computational complexity scheme by Fan and Hasan and the other one is the quadratic computational complexity scheme by Reyhani-Masoleh. In addition, we presented a modified version of the Reyhnai-Masoleh algorithm. By adding $m(w-1)$ bits to the original Reyhani-Masoleh approach, we could improve timings up to 19%. We have showed some important features of the sub-quadratic algorithm. First, it requires much less memory than the quadratic one which makes it more suitable to a constrained environment like smart cards. Second, it achieves much better timing performance on different architectures. Third, by taking advantage of the parallelization feature of the sub-quadratic algorithm, one can potentially further reduce its timing results by approximately 50%.

There is another sub-quadratic complexity approach for ONB II field multiplication that came into our knowledge after preparing this work. This sub-quadratic approach is proposed by Gather, Amin, and Jamshid Shokrollahi [19]. In their work, they use a linear transformation to convert the representation of field elements from normal basis to polynomial basis then multiply the elements using a suitable polynomial multiplier. As a future work, we will compare the memory requirement and the timing performance of the sub-quadratic approach of Fan and Hasan [1] and the work in [19].

References

1. H. Fan and M. A. Hasan, "Subquadratic computational complexity schemes for extended binary field multiplication using optimal normal bases," *IEEE Trans. Comput.*, vol. 56, no. 10, pp. 1435–1437, 2007.
2. A. Reyhani-Masoleh, "Efficient algorithms and architectures for field multiplication using Gaussian normal bases," *IEEE Trans. Comput.*, vol. 55, no. 1, pp. 34–47, 2006.
3. A.J. Menezes, *Applications of Finite Fileds.* Kluwer Academic,Boston ,MA, 1993.
4. P. Ning and Y. L. Yin, "Efficient software implementation for finite field multiplication in normal basis," in *ICICS '01: Proceedings of the Third International Conference on Information and Communications Security.* London, UK: Springer-Verlag, 2001, pp. 177–188.
5. IEEE Standard 1363-2000, IEEE Standard Specifications for Public-Key Cryptography, 2000.
6. J. Omura and J. Massey, "Computational method and apparatus for finite field arithmetic," *U.S. Patent Number 4,587,627*, 1986.
7. R. Dahab, D. Hankerson, F. Hu, M. Long, J. Lóреz, and A. Menezes, "Software multiplication using Gaussian normal bases," *IEEE Trans. Comput*, vol. 55, pp. 974–984, 2006.
8. M. A. Hasan, M. Z. Wang, V. K. Bhargava, "A modified massey-omura parallel multiplier for a class of finite fields," *IEEE Trans. Comput.*, vol. 42, no. 10, pp. 1278–1280, 1993.
9. M.Feng, "A VLSI architecture for fast inversion in *GF*(2^m)," *IEEE Trans. Computers*, vol. 38, no. 10, pp. 1383–1386, Oct. 1989.
10. B. Sunar and Ç. K.Koç, "An efficient optimal normal basis type II multiplier," *IEEE Trans. Comput.*, vol. 50, no. 1, pp. 83–87, 2001.
11. R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, "Optimal normal bases in *GF*(2^n)," *Discrete Appl. Math.*, vol. 22, no. 2, pp. 149–161, 1989.
12. M. Leone, "A new low complexity parallel multiplier for a class of finite fields," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems.* London, UK: Springer-Verlag, 2001, pp. 160–170.
13. H. Fan and M. A. Hasan, "A new approach to subquadratic space complexity parallel multipliers for extended binary fields," *IEEE Trans. Comput.*, vol. 56, no. 2, pp. 224–233, 2007.
14. B. Sunar, "A generalized method for constructing subquadratic complexity *GF*(2^k) multipliers," *IEEE Transactions on Computers*, vol. 53, pp. 1097–1105, 2004.
15. Y. Yin and P. Ning, "Efficient finite field multiplication in normal basis," *US Patent*, no. 6,389,442, 2002.
16. S.Winograd, *Arithmetic Complexity of Computations.* SIAM, 1980.
17. A. Reyhani-Masoleh and M. A. Hasan, "Fast normal basis multiplication using general purpose processors," *IEEE Trans. Comput.*, vol. 52, no. 11, pp. 1379–1390, 2003.
18. National Institute of Standards and Technology, Digital Signature Standard, FIPS Publication 186-2, February 2000.
19. J. von zur Gathen, A. Shokrollahi, and J. Shokrollahi, "Efficient multiplication using type 2 optimal normal bases," in *WAIFI '07: Proceedings of the 1st international workshop on Arithmetic of Finite Fields.* Berlin, Heidelberg: Springer-Verlag, 2007, pp. 55–68.