Efficient Algorithms and Architectures for Field Multiplication Using Gaussian Normal Bases

Arash Reyhani-Masoleh

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
E-mail: arash@secure2.uwaterloo.ca

21st July 2004

Abstract

Recently, implementation of normal basis multiplication over the extended binary field $GF(2^m)$ has received considerable attention. A class of low complexity normal bases called Gaussian normal bases has been included in a number of standards, such as IEEE [9] and NIST [16]. The multiplication algorithms presented there are slow in software since they rely on bit-wise inner product operations. In this report, we present two vector-level algorithms which essentially eliminate such bit-wise operations for Gaussian normal bases. Our analysis and timing results show that the software implementation of the proposed algorithms are faster and/or more memory efficient compared to the best algorithms available in the open literature. Moreover, the hardware structures associated with one algorithm outperform the existing normal basis multiplier structures. We also present some new properties of the multiplication matrix for such Gaussian normal bases.

Keywords: Finite field multiplication, normal basis, Gaussian normal basis, software algorithms, ECDSA.

1 Introduction

The binary finite field $GF(2^m)$ is a set of 2^m elements, where each of these elements can be represented by a bit string of length m. The field can also be considered as a vector space with m linearly independent elements over $GF(2)$ known as a basis. There are two common bases, polynomial basis and normal basis, to represent field elements of $GF(2^m)$ [9]. Field element representations and arithmetic operations (except addition/subtraction) using these two bases are quite different. In normal basis representation, squaring of a field element is done by a cyclic shift of its coordinates and it is free in hardware. Thus, the use of normal basis is more suitable than the polynomial basis in hardware implementation of finite field arithmetic units [1, 13, 23, 21]. For software implementation, multiplication using polynomial bases is considered simpler than that using normal bases. However in some situations such as those where compatibility with other bases and/or efficient squaring is important, one may need to implement normal basis multiplication in software [2, 15, 26, 19].

A Gaussian normal basis (GNB) is a special class of normal basis (NB) which has received considerable attention for efficient implementation of field multiplication [9]. It is well known that for $GF(2^m)$ a normal basis exists for every positive integer m [11], however a GNB exists only if m is not divisible by eight [3]. It is also known that every GNB for $GF(2^m)$ is determined by an integer T and is referred to as type T Gaussian normal basis. For a given pair of integers m and T, there exists at most one type T GNB for $GF(2^m)$. However, there is at least one GNB if m is not a multiple of eight. When there are more than one GNB for a given m, those with smaller values of T yield efficient implementation of field multiplication. This is because the complexity of a type T GNB is proportional to T. A table of GNBs for $GF(2^m)$ for $2 \leq m \leq 1000$ is given in [9].

In this report, we consider GNBs for $GF(2^m)$ where m is an odd number. This implies that the type T of such a GNB for $GF(2^m)$ is an even integer. These classes of GNBs are important for cryptographic applications and include the five $GF(2^m)$ fields, i.e., $m \in \{163, 233, 283, 409, 571\}$, recommended by NIST (National Institute of Standards and Technology) for ECDSA (elliptic curve digital signature algorithm) [16]. In this report, two efficient algorithms for $GF(2^m)$ multiplication using Gaussian normal bases are proposed. For both software and hardware implementations, they are compared with the best algorithms/structures available in the open literature. It is shown that the proposed algorithms/structures outperform the existing algorithms/structures in the open literature.

2 Multiplication Algorithms

In this section, we briefly review multiplication algorithms using an arbitrary normal basis as well as a GNB.

2.1 Multiplication using an Arbitrary Normal Basis

A normal basis of $GF(2^m)$ over $GF(2)$ is a set $N = \{\beta, \beta^2, \cdots, \beta^{2^{m-1}}\}$ whose elements are linearly independent. Let β be a root of an irreducible polynomial $p(t)$ of degree m over $GF(2)$ which is called a normal polynomial. Then, the normal basis N is specified by its normal polynomial. By finding an irreducible normal polynomial one can construct a normal basis of $GF(2^m)$ over $GF(2)$. An algorithm for checking an irreducible polynomial to be normal can be found in [9]. In [9], a procedure to generate an irreducible polynomial is also given.

In $GF(2^m)$, any element $A = (a_0, a_1, \cdots, a_{m-1})$ can be represented as a linear combination of the elements in N, i.e., $A = \sum_{i=0}^{m-1} a_i \beta^{2^i} = a_0\beta + a_1\beta^2 + \cdots + a_{m-1}\beta^{2^{m-1}}$, whose coefficient $a_i \in GF(2)$, $0 \leq i \leq m-1$, denotes the i-th coordinate of A. Let us denote $\mu_{i,j} = \beta^{2^i+2^j} \in GF(2^m)$, $0 \leq i, j \leq m-1$, which is represented with respect to N as $\mu_{i,j} = \sum_{l=0}^{m-1} \mu_{i,j}^{(l)} \beta^{2^l}$. Let $\underline{a} = [a_0, a_1, \cdots, a_{m-1}]$ and $\underline{b} = [b_0, b_1, \cdots, b_{m-1}]$ be the row vectors which correspond to the field elements $A = (a_0, a_1, \cdots, a_{m-1})$ and $B = (b_0, b_1, \cdots, b_{m-1})$, respectively and $C \in GF(2^m)$ denote their product, i.e., $C = (c_0, c_1, \cdots, c_{m-1}) = AB =$

$\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \mu_{i,j}$. Then, the l-th coordinate of C can be written as

$$c_l = \underline{a}\mathbf{M}^{(l)}\underline{b}^{tr},\ 0 \leq l \leq m-1, \tag{1}$$

where $\mathbf{M}^{(l)} = [\mu_{i,j}^{(l)}]_{i,j=0}^{m-1}$, $\mu_{i,j}^{(l)} \in GF(2)$, $0 \leq i,j \leq m-1$, and $\underline{b}^{tr}$ denotes the matrix transpose of row vector $\underline{b}$. In (1), $\mathbf{M}^{(l)}$ is obtained from the l-fold right and down circular shifts of the *multiplication matrix* $\mathbf{M} = \mathbf{M}^{(0)}$. Thus, the entries of $\mathbf{M}^{(l)}$, i.e., $\mu_{i,j}^{(l)}$, can be obtained from the entries of $\mathbf{M}$, i.e., $M(i,j)$, as

$$\mu_{i,j}^{(l)} = M(i-l, j-l),\ 0 \leq i,j,l \leq m-1. \tag{2}$$

To compute the multiplication matrix $\mathbf{M}$, one can see [9] and [20]. Having found the multiplication matrix $\mathbf{M}$ (and hence c_0), one can use the following algorithm to determine the remaining coordinates of C [9].

Algorithm 1. [9] (Multiplication using an arbitrary NB)

Input: $A = (a_0, a_1, \cdots, a_{m-1})$, $B = (b_0, b_1, \cdots, b_{m-1}) \in GF(2^m)$, and $\mathbf{M}$

Output: $C = (c_0, c_1, \cdots, c_{m-1}) = AB$

1. Initialize $X := A$ and $Y := B$
2. For $k = 0$ to $m-1$ {
3. Compute: $c_k = \underline{x}\mathbf{M}\underline{y}^{tr}$
4. $X := X \ll 1$ and $Y := Y \ll 1$
5. }
6. Output $C = (c_0, c_1, \cdots, c_{m-1})$.

In this algorithm, $X \ll 1 = (x_1, \cdots, x_{m-1}, x_0)$ denotes a left cyclic shift of the coordinates of $X = (x_0, x_1, \cdots, x_{m-1})$. Algorithm 1 is indeed an algorithmic version of the architecture of the bit-serial NB multiplier proposed by Massey and Omura in [12]. In that architecture, the number of 1's in the multiplication matrix $\mathbf{M}$ is a good measure of the number of logic gates needed for hardware implementation. This measure is referred to as the *complexity* of the normal basis and is denoted as C_N.

Unlike hardware implementation, software implementation of Algorithm 1 is not very efficient. This is because in each iteration of the for loop of the algorithm, it generates only one coordinate of the product C and requires one matrix-by-vector multiplication, i.e., $\mathbf{M}\underline{y}^{tr}$, and one vector-by-vector multiplication, i.e., $\underline{x}(\mathbf{M}\underline{y}^{tr})$. Let ω denote the data-path width of the processor. Then the coordinates of the field elements and each column of $\mathbf{M}$ can be divided into $\lceil \frac{m}{\omega} \rceil$ words. Thus, for the purpose of storage, the multiplication matrix will require $\lceil \frac{m}{\omega} \rceil m$ words in total.

In [15, 26], Ning and Ying proposed software algorithms to use the full data path of the processor on which their algorithms are implemented. The idea behind the algorithm is to precompute and store the w different cyclic shifts of A and B into a look-up table (i.e., memory). Then, instead of using cyclic shifts (as shown in line 4 of Algorithm 1 above), the shifted values are read from the memory. This is shown below.

Algorithm 2. [15, 26] (Ning-Ying multiplication for arbitrary NBs)

Input: $A = (a_0, a_1, \cdots, a_{m-1}),\ B = (b_0, b_1, \cdots, b_{m-1}) \in GF(2^m)$, and $\mathbf{M}$

Output: $C = (c_0, c_1, \cdots, c_{m-1}) = AB$

1. Precompute arrays for A_i and B_i
2. Initialize $C := 0$
3. For $i = 0$ to $m-1$ {
4. Initialize $S := 0$
5. For $j = 0$ to $m-1$ {
6. If $M(i, j) = 1$ then $S := S + B_j$
7. }
8. $C := C + A_i \odot S$
9. }
10. Output $C = (c_0, c_1, \cdots, c_{m-1})$.

In this algorithm $M(i, j)$ represents the (i, j)-th entry of the multiplication matrix $\mathbf{M}$.

In step 1 of Algorithm 2, each unit of w bits of the cyclic shifts of A (and similarly B) are stored into two arrays as follows

$$A[i] = (a_i,\ \cdots,\ a_{i+w-1 \bmod m}),\ 0 \le i \le m-1. \tag{3}$$

Thus, the i-fold left cyclic shifts of A can be obtained by reading the following $n = \lceil \frac{m}{w} \rceil$ units from the array as

$$A_i = (A[i],\ A[i+w],\ \cdots,\ A[i+(n-1)w]),$$

where the additions inside the square brackets are reduced modulo m.

2.2 Conventional GNB Multiplication Algorithm

It was shown in [14] that for any normal basis of $GF(2^m)$, the number of 1's in the multiplication matrix is always greater than or equal to $2m-1$, i.e., $C_N \ge 2m-1$. In order to have an efficient and simple implementation, one needs to choose a normal basis such that C_N is as small as possible. The best choice for C_N is obviously $2m-1$, and the corresponding NB is called an optimal normal basis (ONB). Two types of ONBs, i.e., type I and type II, were constructed by Mullin *et al.* [14]. However, such ONBs do not exist for all m. As an extension of optimal normal bases, Gaussian normal bases have been constructed by Ash, Blake, and Vanstone in [3]. Like ONBs, GNBs are special classes of normal bases. Although GNBs are generally less efficient than ONBs, when an ONB does not exist, GNB is considered to be the best alternative.

Definition 1. *A type T GNB for $GF(2^m)$ exists if and only if $p = Tm+1$ is prime and* $\gcd(\frac{Tm}{k},\ m) = 1$*, where k is the multiplicative order of 2 modulo p [10].*

It is noted that when $T = 1$ and 2, we have type 1 and type 2 GNBs which are the same as type I and type II ONBs, respectively. It is also noted that for $GF(2^m)$ when m is not a multiple of eight and no ONB exists but at least one GNB exists.

Below we give the conventional GNB multiplication algorithm as described by NIST in [16] and IEEE in [9]. It is noted that since p should be prime, Tm has to be an even integer. This implies that for any odd value of m, T must be an even integer. The case of T even is of particular interest for implementing high speed crypto-systems based on Koblitz curves. Such curves with points on $GF(2^m)$ exist for $m = 163$, 233, 283, 409, 571, where GNBs have T even.

Instead of using $\mathbf{M}$ as an input in Algorithm 1, the following algorithm requires an explicit formula for the first coordinate of the product C, i.e., $c_0 = F(\underline{a}, \underline{b})$, which can be obtained for m odd as follows [9]:

$$c_0 = F(\underline{a}, \underline{b}) = \sum_{k=1}^{p-2} a_{F(k+1)} b_{F(p-k)}, \tag{4}$$

In (4), the sequence $F(1),\ F(2),\ \cdots,\ F(p-1)$ needs to be pre-computed using

$$F(2^i u^j \mod p) = i,\ 0 \le i \le m-1,\ 0 \le j < T, \tag{5}$$

where u is an integer of order $T \mod p$ and $p = Tm + 1$. In (4), $F(k+1)$ and $F(p-k)$ for $1 \le k \le p-2$, are the locations of non-zero entries in $\mathbf{M}$. Thus, one can obtain that the number of 1's in $\mathbf{M}$ is upper bounded by $p-2$, i.e., $C_N \le Tm - 1$ [3].

Algorithm 3. [9, 16] (GNB multiplication for m odd)

Input: $A = (a_0, a_1, \cdots, a_{m-1})$, $B = (b_0, b_1, \cdots, b_{m-1}) \in GF(2^m)$, and $F(\underline{x}, \underline{y})$

Output: $C = (c_0, c_1, \cdots, c_{m-1}) = AB$

1. Initialize $X := A$ and $Y := B$
2. For $k = 0$ to $m-1$ {
3. Compute: $c_k = F(\underline{x}, \underline{y})$
4. $X := X \ll 1$ and $Y := Y \ll 1$
5. }
6. Output $C = (c_0, c_1, \cdots, c_{m-1})$.

In the following sections we discuss two algorithms for efficient software implementation of Gaussian normal basis multiplication and their associated hardware structures can be derived from the algorithms easily. Unlike the algorithms presented in [9, 16] which use bit-wise operations , the proposed algorithms use vectors of bits. This enables us to use the full data-path of the processor on which the software is executed.

3 Modified Ning-Ying Multiplication Algorithm

First, we introduce the following lemmas which are subsequently used in our formulation of a multiplication algorithm (i.e., Algorithm 4).

Lemma 1. *[24] For a self-dual normal basis*[1] *over $GF(2^m)$, when m is odd, the entries of $\mathbf{M}^{(m-1)}$ have the following properties:*

(i) $\mu^{(m-1)}_{i,m-1} = \mu^{(m-1)}_{m-1,i} = \begin{cases} 1 & \mathit{if}\, i = 0 \\ 0 & \mathit{otherwise}, \end{cases}$

(ii) $\mu^{(m-1)}_{i,j} = \mu^{(m-1)}_{(m-1+i-j),(m-j-2)} = \mu^{(m-1)}_{(j-i-1),(m-i-2)}$, *for $i < j$ and $0 \le i, j \le m-1$.*

Lemma 2. *[19] For type T GNBs, where T is an even integer,*

$$F(k) = F(p-k), \;\; 1 \le k \le p-1, \tag{6}$$

where $F(k)$ is defined in (5).

It is well known that the multiplication matrix is symmetric, i.e., $\mathbf{M} = \mathbf{M}^{tr}$. It is also known that all diagonal entries of $\mathbf{M}$ are zero except the last one, i.e., $M(i,i) = \begin{cases} 1, & \text{if } i = m-1 \\ 0, & \text{otherwie.} \end{cases}$

Now we state the following theorem to present certain properties of $\mathbf{M}$ that our first algorithm relies on.

[1]A self-dual normal basis is a basis whose dual basis is itself.

Lemma 3. *For type T Gaussian normal bases over $GF(2^m)$, where m is an odd integer (hence T is even), the multiplication matrix* $\mathbf{M}$ *has the following properties:*

(i) *There is only one non-zero entry in row 0 (or column 0), i.e.,*

$$M(0,j) = M(j,0) = \begin{cases} 1, & \text{if } j = 1 \\ 0, & \text{otherwie.} \end{cases}$$

(ii) *For other rows (or columns), the number of 1's in each row (or column) of* $\mathbf{M}$ *is even and less than or equal to T.*

Proof. Using (2) with $l = j = m-1$ into part (i) of Lemma 1, one can see that the first property exists for all self-dual normal bases. Also, it is proved in [7] that all GNBs of even type are self-dual normal basis. Thus, the proof of the first part is complete.

In order to prove the second part, first we use the above mentioned property of $\mathbf{M}$ and (6) to simplify (4) as follows:

$$\begin{aligned} c_0 &= F(\underline{a}, \underline{b}) = F(\underline{b}, \underline{a}) = a_0 b_1 + \sum_{k=2}^{p-2} a_{F(k)} b_{F(k+1)} \\ &= a_0 b_1 + \sum_{i=1}^{m-1} a_i \left(\sum_{F(k)=i} b_{F(k+1)} \right), \text{ for } 2 \le k \le p-2 \end{aligned} \tag{7}$$

where

$$k = 2^i u^j \mod p,\ 1 \le i \le m-1,\ 0 \le j < T. \tag{8}$$

Since u is an integer of order $T \mod p$ and using (8), one can find that for a given value of i, $1 \le i \le m-1$,

$$F(k) = i, \tag{9}$$

has at most T unique solutions for k, $2 \le k \le p-2$, say $k_1, k_2, \cdots, k_T$. For each i in (7) if $F(k_u + 1) \ne F(k_v + 1)$, $u \ne v$, $1 \le u, v \le T$, then row i has exactly T number of 1's in the multiplication matrix. Otherwise, the number of 1's will be an even integer and it

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$F(k)$	0	1	5	2	1	6	5	3	3	2	4	0	4	6
k	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$F(k)$	6	4	0	4	2	3	3	5	6	1	2	5	1	0

Table 1: The sequence of F for Type 4 GNB over $GF(2^7)$.

will be less than T. This is because in (7) when the terms labeled as $b_{F(k+1)}$ are summed for $F(k) = i$, any two of these terms with the same subscript value would cancel each other.

□

As an example, for type 4 GNB over $GF(2^7)$, one has $p = 29$ and $u = 12$ or 17. The sequence of F is shown in Table 1 which is obtained from [16].

Thus, using (4), we can obtain

$$\begin{aligned} c_0 &= a_0b_1 + a_1(b_0 + b_2 + b_5 + b_6) + a_2(b_1 + b_3 + b_4 + b_5) + a_3(b_2 + b_5) \\ &\quad + a_4(b_2 + b_6) + a_5(b_1 + b_2 + b_3 + b_6) + a_6(b_1 + b_4 + b_5 + b_6), \end{aligned} \tag{10}$$

and corresponding $\mathbf{M}$ can be obtained as

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}. \tag{11}$$

It is seen in (11) that $C_N = 21$.

Remark 1. *For $T = 2$, all rows of $\mathbf{M}$ have two non-zero entries except for row 0.*

Using Lemma 3 and by counting the number of ones in the multiplication matrix, we can obtain the following.

Corollary 1. *The complexity of type T GNB over $GF(2^m)$, is*

$$C_N \leq Tm - T + 1, \tag{12}$$

where m is an odd integer.

Remark 2. *It has been proved in [3] that $C_N \leq Tm - 1$. Thus, the new upper bound in (12) is a tighter upper bound for all even integers $T \geq 2$.*

Remark 3. *For type $T = 4$ GNBs of $GF(2^m)$, $m < 2000$, we have observed that all values of m are odd. For $T = 4$, it has been shown in [3] that $C_N = 4m - 7$. This implies that only two rows of $\mathbf{M}$ should have two non-zero entries. We have verified it using a computer program to generate the non-zero entries of $\mathbf{M}$. Our results show that rows $\frac{m-1}{2}$ and $\frac{m+1}{2}$ of $\mathbf{M}$ have two 1's.*

In order to efficiently implement GNB multiplication in software, it is better to store the columns of 1's of $\mathbf{M}$ instead of storing the whole $\mathbf{M}$. Since the location of the single 1 of row 0 of $\mathbf{M}$ is fixed, we need to store those in rows 1 up to $m-1$. Let $\mathbf{R}$ be an $(m-1)$ by T matrix whose entry $R(i,j)$, $0 \leq R(i,j) \leq m-1$, $1 \leq i \leq m-1$, $1 \leq j \leq T$, contains the columns of 1's in row i of $\mathbf{M}$. If the number of 1's in row i of $\mathbf{M}$ is T, then all $R(i,j)$, $1 \leq j \leq T$, contain an integer in $[0, m-1]$. Otherwise, we initialize the remaining entries, whose number is even, with a constant value, say 0. For example, if we consider $GF(2^7)$ then using (11) the $\mathbf{R}$ matrix of GNB of type 4 is

$$\mathbf{R} = \begin{pmatrix} 0 & 2 & 5 & 6 \\ 1 & 3 & 4 & 5 \\ 2 & 5 & 0 & 0 \\ 2 & 6 & 0 & 0 \\ 1 & 2 & 3 & 6 \\ 1 & 4 & 5 & 6 \end{pmatrix}.$$

Thus, one can obtain c_0 from (4) as

$$c_0 = a_0 b_1 + \sum_{i=1}^{m-1} a_i \left(\sum_{j=1}^{T} b_{R(i,j)} \right). \tag{13}$$

This is because, compared to (4), equation (13) has even number of $b_{R(i,j)}$ terms. Since their associated entries of $R(i,j)$ are the same, the modulo 2 addition of such terms will be zero.

Using (13), one can obtain c_l by adding l modulo m to all indices in (13), i.e.,

$$c_l = a_l b_{l+1 \bmod m} + \sum_{i=1}^{m-1} a_{l+i \bmod m} \left(\sum_{j=1}^{T} b_{l+R(i,j) \bmod m} \right), \; 0 \leq l \leq m-1.$$

Thus, we can state the following.

Theorem 1. *Let C be the product of A and B in $GF(2^m)$, then*

$$C = A \odot B_1 + \sum_{i=1}^{m-1} A_i \odot (B_{R(i,1)} + B_{R(i,2)} + \cdots + B_{R(i,T)}), \tag{14}$$

where $X \odot Y$ denotes the bit-wise AND operation between coordinates of $X = (x_0, x_1, \cdots, x_{m-1})$ and $Y = (y_0, y_1, \cdots, y_{m-1})$, and $X_i = (x_i, x_{i+1}, \cdots, x_{i-1})$ is the $i-$fold left cyclic shift of X.

Based on Theorem 1, we can present the following algorithm for efficient software implementation of type T GNB multiplication for $GF(2^m)$. As seen earlier, Algorithm 2 has an implicit second level of loop in line 3. However, the following algorithm does not have such a loop and this helps to increase the speed of its execution when implemented in software. For the smallest non-zero positive even integer (i.e., $T = 2$), we have type 2 GNBs which are identical to type II ONBs, and for this particular value of T, the following algorithm is the same as the type II ONB multiplication algorithm presented in [15].

Algorithm 4. (Modified Ning-Ying multiplication for m odd)

Input: $A = (a_0,\, a_1,\, \cdots,\, a_{m-1})$, $B = (b_0,\, b_1,\, \cdots, b_{m-1}) \in GF(2^m)$, and $m-1 \times T$ matrix **R**

Output: $C = (c_0,\, c_1,\, \cdots,\, c_{m-1}) = AB$

1. Precompute arrays for A_i and B_i
2. Initialize $C := A \odot B_1$
3. For $i = 1$ to $m-1$ {
4. $\quad C := C + A_i \odot (B_{R(i,1)} + B_{R(i,2)} + \cdots + B_{R(i,T)})$
5. }
6. Output $C = (c_0,\, c_1,\, \cdots,\, c_{m-1})$.

For the efficient software implementation of the above algorithm, the following can be considered.

One important issue is to minimize the number of CPU instructions that need to be executed. In the above algorithm, three different operations namely bit-wise AND, bit-wise XOR and cyclic shifts, are used. In programming language like C, two former operations can be easily implemented by AND and XOR instructions. Although the cyclic shift operation is essentially free in hardware, but it is costly in software. One way to reduce the number of cyclic shift operations is to store cyclicly shifted versions of A and B into memory and read them from memory. In [22], all m different cyclic shifts of field elements of A and B are stored into two m arrays, each of which contains one cyclic shift of A and B. Thus, its memory requirement for each field element is about $2m^2$ bits. We use another efficient method which requires relatively less memory and is used in Algorithm 2 [15, 26]. This method stores only w cyclic shifts of each element and reduces the total memory requirement to about $2wm$ bits which is in line 1 of the algorithm. It precomputes two arrays of

$$\begin{matrix} A[i] = (a_i,\, \cdots,\, a_{i+w-1 \bmod m}) \\ B[i] = (b_i,\, \cdots,\, b_{i+w-1 \bmod m}) \end{matrix},\ 0 \le i \le m-1,$$

and then by reading $n = \lceil \frac{m}{w} \rceil$ units from the memory for generating each cyclic shifts of the elements as

$$\begin{array}{l} A_i = (A[i],\ A[i+w \bmod m],\ \cdots,\ A[i+(n-1)w \bmod m]) \\ B_i = (B[i],\ B[i+w \bmod m],\ \cdots,\ B[i+(n-1)w \bmod m]) \end{array},\ 0 \leq i \leq m-1.$$

4 A More Efficient Algorithm

Let us introduce the following lemma which helps us obtain the multiplication matrix $\mathbf{M}$ for GNBs.

Lemma 4. *For the type T GNB over $GF(2^m)$, when m is odd, the (i,j)-th entry of the multiplication matrix $\mathbf{M}$ is*

$$M(i,j) = \mu_{i,0}^{(j)},\ \ 0 \leq i,j \leq m-1, \tag{15}$$

where $\mu_{i,0} = \beta^{2^i+1} = \sum_{j=0}^{m-1} \mu_{i,0}^{(j)} \beta^{2^j}$.

Proof. Since $\mathbf{M} = \mathbf{M}^{tr}$, it is sufficient to prove it only for $i < j$ and $0 \leq i \leq m-1$. For $i = 0$, (15) is obtained directly from part (i) of Lemma 3. Using (2) into (15), we need to show that

$$M(i,j) = M(m+i-j, m-j),\ 0 < i < j \leq m-1. \tag{16}$$

Using part (ii) of Lemma 1 and (2), one can see that (16) holds for self-dual normal bases. Thus, the proof is complete because all GNBs of even type are self-dual normal bases [7].

□

4.1 Software Implementation

The actual timing of software implementation of an algorithm depends not only on the total number of instructions but also on the memory requirement and the number of memory accesses. The memory issue plays an important role in the analysis of algorithms that are to be implemented in software. This is especially important for resource constrained cryptosystems such as smart cards where the storage memory is not only limited but also slower than what one would find in many other environments. Thus, in addition to reducing the number of instructions, we need to minimize the memory requirement and the number of memory accesses. To do so, in this section we develop another algorithm which is based on the following property of the multiplication matrix $\mathbf{M}$ for GNBs.

Corollary 2. *In the multiplication matrix* $\mathbf{M}$ *of GNBs for* $GF(2^m)$*, when* m *is odd, row* $m-i$*,* $1 \leq i \leq \frac{m-1}{2}$*, is the* $i-$*fold left cyclic shift of row* i*, i.e.,*

$$M(m-i,j) = M(i, j+i \mod m), \; 1 \leq i \leq \frac{m-1}{2}, \; 0 \leq j \leq m-1. \tag{17}$$

Proof. The proof follows from (15) and the fact that $\mu_{i,0} = \mu_{m-i,0}^{2^i}$ for $i > \frac{m-1}{2}$. $\square$

Based on the above property, we can obtain the following relationship among the entries of matrix $\mathbf{R}$:

$$R(m-i,j) = R(i,j) + i \mod m, \; 1 \leq i \leq \frac{m-1}{2}, \; 1 \leq j \leq T. \tag{18}$$

By applying (18) for a given i and all values of j, we then have

$$B_{R(m-i,1)} + B_{R(m-i,2)} + \cdots + B_{R(m-i,T)} = \left(B_{R(i,1)} + B_{R(i,2)} + \cdots + B_{R(i,T)}\right) \ll i \tag{19}$$

Let $\mathbf{R}'$ be an $\frac{m-1}{2} \times T$ matrix which contains the first half of matrix $\mathbf{R}$. For example, considering type 4 GNB for $GF(2^7)$ one has

$$\mathbf{R}' = \begin{pmatrix} 0 & 2 & 5 & 6 \\ 1 & 3 & 4 & 5 \\ 2 & 5 & 0 & 0 \end{pmatrix}.$$

Then using (19) and Theorem 1, one can conclude the following which is the key equation to our next multiplication algorithm.

Theorem 2. *For GNBs in $GF(2^m)$, when m is an odd number, the product of A and B is*

$$C = A \odot B_1 + \sum_{i=1}^{\frac{m-1}{2}} \left((A \ll i) \odot S + (A \gg i) \odot (S \ll i) \right), \tag{20}$$

where

$$S = S(i) = B_{R'(i,1)} + B_{R'(i,2)} + \cdots + B_{R'(i,T)}. \tag{21}$$

In (20), $i-$fold left and right cyclic shifts of A, i.e., $A \ll i$ and $A \gg i$, are used instead of A_i and A_{m-i} as appeared in (14). This is because in the software implementation of Theorem 2 which is given below, we only store the cyclic shifts of one operand namely B. This will help us reduce the memory requirement to one half of that of Algorithm 4. The algorithm presented below is based on Theorem 2 and is given for odd m only. For m even, one can use the algorithms presented in [19].

Algorithm 5. (Efficient GNB Multiplication for $GF(2^m)$ when m is odd)

Input: $A = (a_0, a_1, \cdots, a_{m-1})$, $B = (b_0, b_1, \cdots, b_{m-1}) \in GF(2^m)$, and $\frac{m-1}{2} \times T$ matrix $\mathbf{R}'$

Output: $C = (c_0, c_1, \cdots, c_{m-1}) = AB$

1. Precompute array for B_i
2. Initialize $LA := A$, $RA := A$, $C := A \odot B_1$
3. For $i = 1$ to $\frac{m-1}{2}$ {
4. $\quad LA := LA \ll 1$, $S := B_{R'(i,1)} + B_{R'(i,2)} + \cdots + B_{R'(i,T)}$
5. $\quad C := C + LA \odot S$
6. $\quad RA := RA \gg 1$, $S := S \ll i$
7. $\quad C := C + RA \odot S$
8. }
9. Output $C = (c_0, c_1, \cdots, c_{m-1})$.

Compared to Algorithm 4, in Algorithm 5 not only the number of XOR instructions is reduced but also the number of memory accesses for reading B_i has become half. This will decrease the computational time of the algorithm when it is implemented on resource constrained cryptosystems.

4.2 Hardware Implementations

Bit-parallel implementation of Algorithm 5 is straitforward. In such a case, there is no need to store the cyclic shifts of B into the memory and they can be obtained as free as re-wiring. Let h_i, $1 \leq i \leq \frac{m-1}{2}$ be the number of 1's in the row i of the multiplication matrix $\mathbf{M}$. Then, by implementing (20) and only considering the first h_i terms in (21), one can obtain the bit-parallel multiplier. Let T_A and T_X be the time delay of an AND gate and a XOR gate, respectively. Then, the complexities of the bit-parallel can be obtained as follows.

Proposition 1. *The time delay and the number of AND and XOR gates of the bit-parallel multiplier using GNBs are* $\leq T_A + (\lceil \log_2 T \rceil + \lceil \log_2 m \rceil) T_X$, m^2 *and* $\leq \frac{m}{2}(C_N + m - 2)$, *respectively.*

In [1], Agnew et al. presented an architecture for bit-serial multiplier using the normal bases. The output coordinates of this multiplier are generated in parallel after m clock cycles. Using (17) and (20), one can reduce the number of XOR gates required in (21) to one half of the one proposed in [1]. For the type 4 GNB in $GF(2^7)$, one can use (10) to obtain all coordinates of C as

$$\begin{aligned} c_l \;=\; & a_l b_{l+1} + a_{l+1}(b_l + b_{l+2} + b_{l+5} + b_{l+6}) + a_{l+2}(b_{l+1} + b_{l+3} + b_{l+4} + b_{l+5}) \\ & + a_{l+3}(b_{l+2} + b_{l+5}) + a_{l+4}(b_{l+2} + b_{l+6}) \\ & + a_{l+5}(b_{l+1} + b_{l+2} + b_{l+3} + b_{l+6}) + a_{l+6}(b_{l+1} + b_{l+4} + b_{l+5} + b_{l+6}), \end{aligned} \tag{22}$$

where $0 \leq l \leq m-1$ and the additions in the subscript indices are reduced modulo $m = 7$. In (22), if one implements the first term, i.e., $a_0 b_1$ for c_0, the second term, i.e., $a_2(b_1+b_3+b_6+b_0)$ for c_1 and up to the final term, i.e., $a_5(b_0 + b_3 + b_4 + b_5)$ for c_6, the architecture of Agnew et al. [1] is obtained. The term $a_{f_l}(\ldots)$ for c_l, $0 \leq l \leq m-1$, can be selected in other ways as

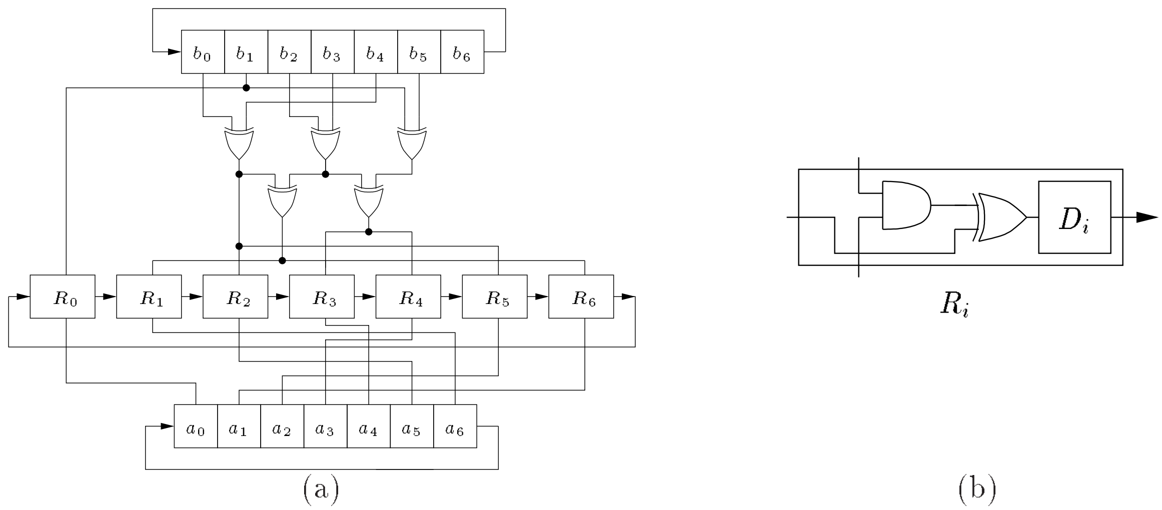

Figure 1: (a) Proposed $GF(2^7)$ bit-serial multiplier. (b) Details of the R_i cell.

long as we have $f_i \neq f_j$ for any $0 \leq i, j \leq m-1$, $i \neq j$. In total, there are $m!$ combinations for the initial term selections and among them, one is used in [1]. If one implements the term $a_{m-l \bmod m}(\ldots)$ for c_l, $0 \leq l \leq m-1$, then the property (17) is used to minimize the number of XOR gates. Thus, if one uses (22) to implement the above mentioned initial term selections for c_l, i.e., a_0b_1 for c_0, $a_6(b_2+b_3+b_4+b_0)$ for c_1, $a_5(b_4+b_0)$ for c_2, $a_4(b_3+b_5+b_1+b_2)$ for c_3, $a_3(b_5+b_1+b_2+b_3)$ for c_4, $a_2(b_0+b_4)$ for c_5, and $a_1(b_0+b_2+b_3+b_4)$ for c_6, then the proposed bit-serial architecture is obtained. As seen in the terms for c_l and c_{m-l}, their expressions in the brackets are the same. The proposed bit-serial multiplier using the type 4 GNB for $GF(2^7)$ is shown in Figure 1(a). The initial contents of shift registers A and B are shown in the figure. Details of the R_i cell are shown in Figure 1(b). Initially, the D_i latches of R_i's are cleared to zero and after m clock cycles the D_i's contain the coordinates of $C = AB$.

It is noted that there may exist another initial term selections for c_l's to further reduce the number of XOR gates. Investigation is being carried out to obtain the best multiplier such that the number of XOR gates is minimum for the five binary field recommended by NIST for ECDSA. Thus, we can conclude the following.

Proposition 2. *The time delay and the number of AND and XOR gates of the bit-serial multiplier using GNBs are* $T_A + (1 + \lceil \log_2 T \rceil)T_X$, m *and* $\leq \frac{C_N+m}{2}$, *respectively.*

5 Comparison and Conclusions

In this report, we have proposed $GF(2^m)$ multiplication algorithms and architectures using type T Gaussian normal bases, when m is odd. Also, their implementations in both software and hardware have been considered. Below, we have compared the proposed algorithms and architectures with the existing ones in the open literature.

Software Algorithms

For the software implementations, Table 2 compares the multiplication algorithms of Reyhani-Hasan (Alg. 3 in [19]) and Ning-Ying (Alg. 2 and Alg. 3 of [15, 26]) in terms of number of processor or CPU instructions and memory requirements. The main arithmetic and logical operations used in these algorithms include AND, XOR and cyclic shifts. These operations are shown with corresponding CPU instructions in three different columns of the table. The instruction counts shown in the table are based on the assumption that one XOR, one AND and one cyclic operation require n XOR, n AND and ρn *assorted* instructions, respectively, where $n = \lceil \frac{m}{\omega} \rceil$ and ρ is a small integer and is typically 4 for the C programming language.

As seen in the table, Alg. 2 of [15, 26] (which is also Algorithm 2 presented in this report) requires C_N field additions and m bit-wise AND operations. It also requires $2w$ cyclic shift operations which are implemented using $2\rho w n$ *assorted* instructions. For realizing one *if* condition in line 6, we assume that q instructions are needed and this results in a total of qm^2 instructions for all *if* conditions. The typical value of q is 2. For the memory requirement, we assume that each column of $\mathbf{M}$ in Algorithm 2 is stored in n units of w bits each with a total of mnw bits. Also in step 1 of this algorithm another $2wn$ bits are required. For the number of memory accesses needed in Algorithm 2, one needs to consider accesses to each memory unit of w bits. This number is determined by taking into account the accesses due to $\mathbf{M}$, A_i's and B_i's which are nm, nm, and nC_N, respectively.

For other algorithms presented in Table 2, one can perform analyses as above. The results are shown in Table 2. In our analysis of memory size, we assume that each entry of $\mathbf{R}$ and

Algorithms	# Instructions			Memory	
	XOR	AND	Others	Size in bits	# Accesses
Alg. 3 of [19]	$\frac{n}{2}(C_N + m - 2)$	mn	$\frac{\rho n}{2}(C_N + 2m - 1)$	$\frac{C_N - 1}{2}\lceil \log_2 m \rceil$	$\frac{C_N - 1}{2}$
Alg. 2 of [15, 26]	nC_N	mn	$2\rho wn + qm^2$	$mnw + 2wm$	$2mn + nC_N$
Alg. 3 of [15, 26]	$2n(m-1)$	mn	$2\rho wn$	$(2m-1)\lceil \log_2 m \rceil + 2wm$	$2m - 1 + (2m-1)n$
Algorithm 4	$Tn(m-1)$	mn	$2\rho wn$	$Tm\lceil \log_2 m \rceil + 2wm$	$T(m-1) + T(m-1)n$
Algorithm 5	$\frac{T+1}{2}n(m-1)$	mn	$\rho wn + 1.5\rho n(m-1)$	$\frac{T(m-1)}{2}\lceil \log_2 m \rceil + wm$	$\frac{T(m-1)}{2} + \frac{T(m-1)n}{2}$

Table 2: Generic comparison of multiplication algorithms in terms of number of instructions and memory requirements, $n = \lceil \frac{m}{\omega} \rceil$. Note that $2m - 1 \le C_N \le Tm - T + 1$ and Alg. 3 of [15, 26] is only for $T = 2$ (i.e., $C_N = 2m - 1$).

$\mathbf{R}'$ requires $\lceil \log_2 m \rceil$ bits as used in [19]. As seen in the table, our proposed Algorithm 5 requires the least amount of memory after Alg. 3 of [19].

We have coded the above mentioned algorithms and Table 3 compares the GNB multiplication algorithms for the five binary fields recommended by NIST for ECDSA applications. The codes have been written using the C programming language and have been executed on two platforms– a Sun Blade 100 with SparcV9 processor @ 502MHz, 512MB of RAM and a PC with Pentium 4 @ 3.00 GHz, 1.00 GB of RAM. As seen in this table, our implementation on the slower machine i.e., Sun Blade 100, shows that the proposed Algorithm 5 is faster than any other algorithms for all type T GNBs. The implementation on a faster machine i.e., Pentium 4, also shows that the proposed Algorithm 5 is faster than any other algorithms for all type T GNBs, except $T = 2$. This is perhaps due to the bigger RAM available on our PC. Nevertheless, for this particular value of T the size of memory required for Algorithm 5 is about half of what is needed in Alg. 3 of [15, 26]. Moreover, Algorithm 5 is expected to be faster when it is run with the same amount of memory as required in Alg. 3 of [15, 26]. That is done by storing the precomputed $A[i]$'s and reading them from memory as we have done for $B[i]$'s.

Hardware Architectures

Tables 4 and 5, respectively, compare the proposed bit-parallel and bit-serial architectures with the existing ones in terms of number of AND gates, XOR gates and time delay. As seen

Platform (time unit)	m	T	Reyhani-Hasan	Ning-Ying [15, 26]		Proposed algorithms	
			Alg. 3 of [19]	Alg. 2	Alg. 3	Algorithm 4	Algorithm 5
Sun Blade 100 (ms)	163	4	0.596	14.623	-	0.372	0.267
	233	2	0.723	49.602	0.424	-	0.351
	283	6	1.969	82.380	-	1.343	0.846
	409	4	3.055	246.771	-	1.986	1.34
	571	10	11.397	706.932	-	9.219	4.888
Pentium 4 (μs)	163	4	101.5	1664	-	57.8	56.2
	233	2	125	4375	65.6	-	76.6
	283	6	342.2	7623	-	215.6	168.8
	409	4	529.7	23375	-	306.3	276.5
	571	10	2006.3	60609	-	1342.2	942.2

Table 3: Comparison of software implementations of the GNB multiplication for the five binary fields recommended by NIST for ECDSA. Note that Alg. 3 of [15] is a special case (i.e., with $T = 2$) of Algorithm 4 presented here.

Multipliers	#AND	#XOR	Time Delay
Massey-Omura [25]	m^2	$m(C_N - 1)$	$T_A + \lceil \log_2 C_N \rceil T_X$
IMO [6]	m^2	$m(C_N - 1)$	$\leq T_A + (\lceil \log_2 T \rceil + \lceil \log_2 m \rceil) T_X$
RRMO [17]	m^2	$\leq \frac{m}{2}(C_N + m - 2)$	$T_A + \lceil \log_2 C_N \rceil T_X$
LCNB [18]	$\frac{m}{2}(m-1)$	$\leq \frac{m}{2}(C_N + 2m - 3)$	$T_A + \lceil \log_2 C_N \rceil T_X$
Presented here	m^2	$\leq \frac{m}{2}(C_N + m - 2)$	$\leq T_A + (\lceil \log_2 T \rceil + \lceil \log_2 m \rceil) T_X$

Table 4: Comparison among bit-parallel multipliers for type T GNBs over $GF(2^m)$, where m is odd. Note that $2m - 1 \leq C_N \leq Tm - T + 1$ and IMO=Improved Massey-Omura, RRMO=Reduced Redundancy Massey-Omura, LCNB=Low Complexity Normal Basis.

in Table 4, the complexities of the bit-parallel multiplier match the ones proposed in [17].

Also, Table 5 shows that the proposed bit-serial structure has the same total space as the best one available in the open literature, i.e., [8] and [21]. Moreover, it has the same time delay as the fastest available architecture in the open literature, i.e.,[4] and [1].

As seen in these tables, the proposed hardware structures outperform the best ones available in the open literature.

Acknowledgments

The author would like to thank Professor Anwar Hasan for his valuable comments and suggestions and Ray Zhao for his help with implementations of the algorithms. I also would like

bit-serial Multiplier	#AND gates	#XOR gates	#Latches	Critical path delay	Output format
Massey-Omura[12]	C_N	$C_N - 1$	$2m$	$T_A + \lceil \log_2 C_N \rceil T_X$	serial
IMO [6]	m	$C_N - 1$	$2m$	$T_A + (\lceil \log_2 T \rceil + \lceil \log_2 m \rceil) T_X$	serial
Beth-Gollmann [4]	m	C_N	$2m$	$T_A + (1 + \lceil \log_2 T \rceil) T_X$	parallel
GG [8], b-SMPO$_{\text{II}}$[21]	m	$\leq \frac{C_N+m}{2}$	$3m$	$T_A + (1 + \lceil \log_2 (T+1) \rceil) T_X$	parallel
Feng [5]	$2m-1$	$\leq C_N + m - 1$	$3m-2$	$T_A + (3 + \lceil \log_2 T \rceil) T_X$	parallel
Agnew et al. [1]	m	C_N	$3m$	$T_A + (1 + \lceil \log_2 T \rceil) T_X$	parallel
b-SMPO$_{\text{I}}$[21]	$\frac{m+1}{2}$	$\leq \frac{C_N+2m-1}{2}$	$3m$	$T_A + (1 + \lceil \log_2 (T+1) \rceil) T_X$	parallel
Presented here	m	$\leq \frac{C_N+m}{2}$	$3m$	$T_A + (1 + \lceil \log_2 T \rceil) T_X$	parallel

Table 5: Comparison among bit-serial multipliers for type T GNBs over $GF(2^m)$, where m is odd. Note that $2m-1 \leq C_N \leq Tm - T + 1$ and IMO=Improved Massey-Omura, GG=Geiselmann-Gollmann.

to thank the managing board and staff of the Centre for Applied Cryptographic Research for providing a good research enviornment during my work in the depatment of Combinatorics and Optimization. This work has been supported in part by an NSERC postdoctoral fellowship awarded to A. Reyhani-Masoleh.

References

[1] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone. "An Implementation for a Fast Public-Key Cryptosystem". *Journal of Cryptology*, 3:63–79, 1991.

[2] K. Aoki and K. Ohta. "Fast Arithmetic Operations over F_{2^n} for Software Implementation". In *Fourth Annual Workshop on Selected Areas in Cryptography, SAC' 97*, 1997.

[3] D. W. Ash, I. F. Blake, and S. A. Vanstone. "Low Complexity Normal Bases". *Discrete Applied Mathematics*, 25:191–210, 1989.

[4] T. Beth and D. Gollman. "Algorithm Engineering for Public Key Algorithms". *IEEE J. Selected Areas in Communications*, 7(4):458–465, May 1989.

[5] M. Feng. "A VLSI Architecture for Fast Inversion in $GF(2^m)$". *IEEE Transactions on Computers*, 38(10):1383–1386, October 1989.

[6] L. Gao and G. E. Sobelman. "Improved VLSI Designs for Multiplication and Inversion in $GF(2^M)$ over normal bases". In *Proceedings of 13th Annual IEEE International ASIC/SOC Conference*, pages 97 –101, 2000.

[7] S. Gao, J. Gathen, D. Panario, and V. Shoup. "Algorithms for exponentiation in finite fields". *J. Symbolic Computation*, 29:879–889, 2000.

[8] W. Geiselmann and D. Gollmann. "Symmetry and Duality in Normal Basis Multiplication". In *LNCS-357, Proc. AAECC-6*, pages 230–238, Rome, July 1988. Springer-Verlag.

[9] IEEE Std 1363-2000. "IEEE Standard Specifications for Public-Key Cryptography". January 2000.

[10] D. Johnson, A. Menezes, and S. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". *International Journal of Information Security*, 1:36–63, 2001.

[11] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 1994.

[12] J. L. Massey and J. K. Omura. "Computational Method and Apparatus for Finite Field Arithmetic". *US Patent No. 4,587,627*, 1986.

[13] R. C. Mullin. "Multiple Bit Multiplier". *US Patent No. 5,787,028*, July 1998.

[14] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. "Optimal Normal Bases in $GF(p^n)$". *Discrete Applied Mathematics*, 22:149–161, 1988/89.

[15] P. Ning and Y. L. Yin. Efficient Software Implementation for Finite Field Multiplication in Normal Basis. In *Proceedings of Information and Communications Security, ICICS 2001*, pages 177–181. LNCS 2229, Springer, November 2001.

[16] National Institute of Standards and Technology. *Digital Signature Standard*. FIPS Publication 186-2, January 2000.

[17] A. Reyhani-Masoleh and M. A. Hasan. "A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$". *IEEE Transactions on Computers*, 51(5):511–520, May 2002.

[18] A. Reyhani-Masoleh and M. A. Hasan. "Efficient Multiplication Beyond Optimal Normal Bases". *IEEE Transactions on Computers, Special Issue on Cryptographic Hardware and Embedded Systems*, 52(4):428–439, April 2003.

[19] A. Reyhani-Masoleh and M. A. Hasan. "Fast Normal Basis Multiplication Using General Purpose Processors". *IEEE Transactions on Computers*, 52(11):1379–1390, Nov. 2003.

[20] A. Reyhani-Masoleh and M. A. Hasan. "Efficient Digit-Serial Normal Basis Multipliers over $GF(2^m)$". *ACM Transactions on Embedded Computing Systems (TECS), Special issue on embedded system security*, Aug. 2004.

[21] A. Reyhani-Masoleh and M. A. Hasan. "Low Complexity Word-Level Sequential Normal Basis Multipliers". *IEEE Transactions on Computers*, to appear 2004. Also, available at "www.cacr.math.uwaterloo.ca" as the Technical Report CORR 2004-14.

[22] M. Rosing. *Implementing Elliptic Curve Cryptography*. Manning Publications Company, 1999.

[23] B. Sunar and C. K. Koc. "An Efficient Optimal Normal Basis Type II Multiplier". *IEEE Transactions on Computers*, 50(1):83–88, Jan. 2001.

[24] C. C. Wang. "An Algorithm to Design Finite Field Multipliers Using a Self-Dual Normal Basis". *IEEE Transactions on Computers*, 38(10):1457–1460, October 1989.

[25] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed. "VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$". *IEEE Transactions on Computers*, 34(8):709–716, Aug. 1985.

[26] Y. L. Yin and P. Ning. "Efficient finite field multiplication in normal basis". *US Patent No. 6,389,442*, May 2002. Assignee: RSA Security Inc.