Fast Normal Basis Multiplication Using General Purpose Processors

Arash Reyhani-Masoleh and M. A. Hasan

April 19, 2001

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
Emails: arash@vlsi.uwaterloo.ca and ahasan@ece.uwaterloo.ca

Abstract

For cryptographic applications, normal bases have received considerable attention, especially for hardware implementation. In this document, we consider fast software algorithms for normal basis multiplication over the extended binary field GF(2^m). We present a vector-level algorithm which essentially eliminates the bit-wise inner products needed in the conventional approach to the normal basis multiplication. We then present another algorithm which significantly reduces the dynamic instruction counts. Both algorithms utilize the full width of the data-path of the general purpose processor on which the software is to be executed. We also consider composite fields and present an algorithm which can provide further speed-ups and an added flexibility toward hardware-software co-design of processors for very large finite fields.

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

1 Introduction

The extended binary finite field GF(2^m) of degree m is used in important cryptographic operations, such as, key exchange, signing and verification. For today's security applications the minimum values of m are considered to be 160 in the elliptic curve cryptography and 1024 in the standard discrete log based cryptography. Elliptic curve crypto-systems use relatively smaller field sizes, but require considerable amount of field arithmetic for each group operation (i.e., addition of two points). In such crypto-systems, often the most complicated and expensive module is the finite field arithmetic unit. As a result, it is important to develop suitable finite field arithmetic algorithms and architectures that can meet the constraints of various implementation technologies, such as, hardware and software.

For cryptographic applications, the most frequently used GF(2^m) arithmetic operations are addition and multiplication. Compared to the former, the latter is much more complicated and time consuming operation. The complexity of GF(2^m) multiplication depends very much on how the field elements are represented. For hardware implementation of a multiplier, the use of normal bases has received considerable attention and a number of hardware architectures and implementations have been reported (see for example [1], [2], [7], [19]). A majority of such efforts were motivated by the fact that certain normal bases, e.g., optimal bases, yield area efficient multipliers, and that the field squaring, which is heavily used in exponentiation and Frobenius mapping, is a simple cycle shift of the field element's coordinates and hence in hardware it is almost free of cost. However, the task of implementing a normal basis multiplier in hardware poses a number of challenges. For example, when one has to deal with very large fields, the interconnections among the various parts of the multiplier could be quite irregular which may slow down the clock speed. Also, normal basis multipliers are not easily scalable with m. Given a normal basis multiplier designed for GF(2^{233}), one cannot conveniently make it usable for GF(2^{163}) or GF(2^{283}).

Unlike hardware, so far software implementation of a GF(2^m) multiplier using normal bases has not been very efficient. This is mainly due to a number of practical considerations. Most importantly, normal basis multiplication algorithms require inner products or matrix multiplications over the ground field GF(2). Such computations are not directly supported by most of today's general purpose processors. These computations require bit-by-bit logical

AND and XOR operations, which are not efficiently implemented using the instruction set supported by the processors. Also, when a high level programming language, such as, C is used, the cyclic shifts needed for field squaring operations, are not as efficient as they are in hardware.

In this document, we consider algorithms for fast software normal basis multiplication on general purpose processors. We discuss how the conventional bit-level algorithm for normal basis multiplication fails to utilize the full data-path of the processor and makes its software implementation inefficient. We then present a vector-level normal basis multiplication algorithm which eliminates the matrix multiplication over GF(2) and significantly reduces the number of dynamic instructions. We then derive another scheme for normal basis multiplication to further improve the speed. We also consider normal basis multiplication over certain special classes of composite fields. We show that normal basis multipliers over such composite fields can provide an additional speed-up and a great deal of flexibility toward hardware-software co-design of very large finite field processors.

The organization of this document is as follows. In Section 2, the NB representation and its conventional multiplication algorithm are presented. This algorithm is recommended by NIST for NB multiplication in ECDSA (elliptic curve digital signature algorithm) [13] where all recommended values of m are prime. Then, we extend it to vector level multiplication algorithm to use all data-path of the processor in Section 3. In Section 4, an efficient algorithm for NB multiplication is appeared. Also, its cost is considered and is compared with other algorithms in this section. Then, we present an efficient algorithm for multiplication over finite fields $GF(2^m)$ with composite values of m in Section 5. Finally, the concluding remarks are given in Section 6.

2 Preliminaries

2.1 Normal Basis Representation

It is well known that there exists a normal basis (NB) in the field $GF(2^m)$ over $GF(2)$ for all positive integers m. By finding an element $\beta \in GF(2^m)$ such that $\{\beta, \beta^2, \cdots, \beta^{2^{m-1}}\}$ is a basis of $GF(2^m)$ over $GF(2)$, any element $A \in GF(2^m)$ can be represented as $A = \sum_{i=0}^{m-1} a_i \beta^{2^i} =$

$a_0\beta + a_1\beta^2 + \cdots + a_{m-1}\beta^{2^{m-1}}$, where $a_i \in GF(2)$, $0 \leq i \leq m-1$, is the i-th coordinate of A. In this document, this normal basis representation of A will be written in short as $A = (a_0, a_1, \cdots, a_{m-1})$. In vector notation, element A will be written as $A = \underline{a} \cdot \underline{\beta}^T = \underline{\beta} \cdot \underline{a}^T$, where $\underline{a} = [a_0, a_1, \cdots, a_{m-1}]$, $\underline{\beta} = [\beta, \beta^2, \cdots, \beta^{2^{m-1}}]$, and T denotes vector transposition. Now, consider the following matrix

$$\mathbf{M} = \underline{\beta}^T \cdot \underline{\beta} = \left[\beta^{2^i+2^j}\right]_{i,j=0}^{m-1}, \tag{1}$$

whose entries belong to GF(2^m). Writing these entries with respect to the NB, one obtains the following:

$$\mathbf{M} = \mathbf{M}_0\beta + \mathbf{M}_1\beta^2 + \cdots + \mathbf{M}_{m-1}\beta^{2^{m-1}}, \tag{2}$$

where $\mathbf{M}_i$'s are $m \times m$ *multiplication matrices* whose entries belong to $GF(2)$. Following [12], we now give the definition of the complexity of a NB as follows.

Definition 1. *The numbers of 1's in all $\mathbf{M}_i$'s are equal. Let us define this number by*

$$C_N = H(\mathbf{M}_i), \quad 0 \leq i \leq m-1, \tag{3}$$

which is known as the complexity of the NB. *In (3), $H(\mathbf{M}_i)$ refers to the Hamming weight, i.e., the number of 1's, in $\mathbf{M}_i$.*

It is well known that $C_N \geq 2m-1$ [12]. When $C_N = 2m-1$, the NB is called an optimal normal basis (ONB). Two types of ONBs were constructed by Mullin *et al.* [12]. Gao and Lenstra [5] showed that these two types are all the ONBs in $GF(2^m)$. As an extension of the work on ONBs, Ash et al. in [3] proposed low complexity normal bases of type t where t is a positive integer. These low complexity bases are referred to as *Gaussian Normal Basis* (GNB). When $t = 1$ and 2, the GNBs become the two types of ONBs of [3]. For $GF(2^m)$, a GNB exists if m is not divisible by 8. 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 [8].

2.2 Conventional NB Multiplication Algorithm

Below, we give the conventional algorithm for normal basis multiplication. This algorithm is for t even only (the reader is referred to [8] for algorithm with t odd). The case of t even is of particular interest for implementing high speed crypto-systems based on Koblitz curves. Such curves with points over $GF(2^m)$ exist for $m = 163, 233, 283, 409, 571$, where normal bases have t even. Note that in the following algorithm, $p = tm + 1$, and $A \ll i$ (resp. $A \gg i$) denotes i-fold left (resp. right) cyclic shifts of the coordinates of A. The algorithm requires the input sequence $F(1), F(2), \cdots, F(p-1)$ to be pre-computed using

$$F(2^i u^j \bmod p) = i,\ 0 \le i \le m-1,\ 0 \le j < t, \tag{4}$$

where u is an integer of order $t \bmod p$.

Algorithm 1. (Bit-Level NB Multiplication)

Input: $A, B \in GF(2^m)$, $F(n) \in [0, m-1]$ for $1 \le n \le p-1$

Output: $C = AB$

```
1.   Initialize C = (c_0, c_1, ..., c_{m-1}) := 0
2.   For i = 0 to m - 1 {
3.      For n = 1 to p - 2 {
4.         c_i := c_i + a_{F(n+1)} b_{F(p-n)}
5.      }
6.      A << 1, B << 1
7.   }
```

Software implementation of Algorithm 1 is not very efficient for the following reasons. First, in each execution of line 4, one coordinate of each of A and B are accessed. These accesses are such that their software implementation is rather unsystematic and typically requires more than one instruction. Secondly, in line 4 the mod 2 *multiplication* of the coordinates, which is implemented by bit level logical AND operation, is performed $m(p-2)$ times in total, and the mod 2 *addition*, which is implemented by bit level logical XOR operation, is performed $\frac{1}{4}m(p-2)$ times, on average, assuming that A and B are two random inputs. In the C programming language, these mod 2 multiplication and addition operations correspond to about $m(p-2)$ AND and $\frac{1}{4}m(p-2)$ XOR instructions[1], respectively.

[1] These are dynamic instructions which the underlying processor needs to execute.

3 Vector-Level NB Multiplication

In this section we discuss improvements to Algorithm 1 so that normal basis multiplication can be efficiently implemented in software. One crucial improvement is that most arithmetic operations are done on vectors instead of bits. This enables us to use the full data-path of the processor on which the software is executed. The assumption of Algorithm 1 that t is even is also used in the remaining discussion of this section.

Lemma 1. *For GNB of type t, where t is even, the sequence $F(n)$ of $p-1$ integers as defined above is mirror symmetric around the center, i.e., $F(n) = F(p-n)$, $1 \leq n \leq p-1$.*

Proof. In (4), t is the smallest nonzero integer such that $u^t \mod p = 1$. Then $u^{\frac{t}{2}} \mod p$ must be equal to -1 . For $0 \leq i \leq m-1$ and $0 \leq j \leq t-1$, let $n = 2^i u^j \mod p$. Then $F(n) = F(2^i u^j \mod p) = i$. Also, $F(2^i u^{\frac{t}{2}+j} \mod p) = i$. Thus

$$\begin{aligned} F(n) &= F(2^i u^{\frac{t}{2}+j} \mod p) = F(-2^i u^j \mod p) \\ &= F(p-n). \end{aligned}$$

$\square$

From (4) and Lemma 1, one has $F(1) = F(p-1) = 0$. For $1 \leq n \leq p-2$, let us define

$$\Delta F(n) = F(n+1) - F(n) \mod m. \tag{5}$$

Now we have the following corollary.

Corollary 1. *For $\Delta F(n)$ as defined above and for t even, the following holds*

$$\Delta F(p-n) = m - \Delta F(n-1) \mod m,\ 1 \leq n \leq p-2.$$

Proof. Using (5), one obtains $F(n+1) = \sum_{i=1}^{n} \Delta F(i)$. Applying Lemma 1 into (5), one can also write

$$\Delta F(p-n) = -\Delta F(n-1) \mod m,\ 2 \leq n \leq p-1$$

which results in

$$\Delta F(p-n) = m - \Delta F(n-1),\ \ 2 \leq n < \frac{p-1}{2},\ \text{and}\ \Delta F(\frac{p-1}{2}) = 0.$$

$\square$

In Algorithm 1, the i-th coordinate of the product $C = AB$ is computed in its inner loop which can be written as follows

$$c_i = \sum_{n=1}^{p-2} a_{F(n+1)+i} b_{F(p-n)+i}, \; 0 \leq i \leq m-1. \tag{6}$$

Using Lemma 1 and equation (5), one can write

$$\begin{aligned} c_i &= \sum_{n=1}^{p-2} a_{F(n+1)+i} b_{F(n)+i}, \; 0 \leq i \leq m-1, && (7) \\ &= \sum_{n=1}^{p-2} a_{F(n)+\Delta F(n)+i} b_{F(n)+i}, \; 0 \leq i \leq m-1. && (8) \end{aligned}$$

For a particular GNB, the values of $\Delta F(n)$, $1 \leq n \leq p-2$, are fixed and are to be determined only once, i.e., at the time of choosing the basis. Additionally, Corollary 1 implies that it is sufficient to store only half (i.e., $\frac{p-1}{2}$) of these $\Delta F(n)$'s. We now state the vector-level algorithm for t even as follows. Similar algorithm for odd values of t is given in Appendix A.

Algorithm 2. (Vector-Level NB Multiplication)

Input: $A,\, B \in GF(2^m)$, $\Delta F(n) \in [0, m-1]$, $1 \leq n \leq p-1$

Output: $C = AB$

1. Initialize $S_A := A$, $S_B := B$, $C := 0$
2. For $n = 1$ to $p-2$ {
3. $\quad S_A \ll \Delta F(n)$
4. $\quad R := S_A \odot S_B$
5. $\quad C := C + R$
6. $\quad S_B \ll \Delta F(n)$
7. }

In line 4 of Algorithm 2, for $X, Y \in GF(2^m)$, $X \odot Y$ denotes the bit-wise AND operation between coordinates of X and Y, i.e., $X \odot Y = (x_0 y_0,\, x_1 y_1,\, \cdots,\, x_{m-1} y_{m-1})$. The following example can be used to illustrate the operation of this algorithm.

n	1	2	3	4	5	6	7	8	9	10
$F(n)$	0	1	3	2	4	4	2	3	1	0
$\Delta F(n)$	1	2	4	2	0	3	1	3	4	-

Table 1: Values of F of type 2 GNB for $GF(2^5)$.

n	$\Delta F(n)$	S_A	S_B	R	C
-	-	01110	10101	-	00000
1	1	11100	10101	10100	10100
2	2	10011	01011	00011	10111
3	4	11001	01101	01001	11110
4	2	00111	10110	00110	11000
5	0	00111	11010	00010	11010
6	3	11001	11010	11000	00010
7	1	10011	10110	10010	10000
8	3	11100	01101	01100	11100
9	4	01110	01011	01010	10110

Table 2: Contents of variables in Algorithm 2 for multiplication of $A = (01110)$ and $B = (10101)$.

Example 1. *For the type 2 GNB in $GF(2^5)$, one has $p = 11$ and $u = 10$. Thus, the values of $F(n)$, $1 \leq n \leq 10$, using (4) are given in Table 1. By using (6), the coordinates of C are obtained as*

$$\begin{aligned} c_i &= a_{i+1}b_i + a_{i+3}b_{i+1} + a_{i+2}b_{i+3} + a_{i+4}b_{i+2} + a_{i+4}b_{i+4} \\ &\quad + a_{i+2}b_{i+4} + a_{i+3}b_{i+2} + a_{i+1}b_{i+3} + a_ib_{i+1}, \ 0 \leq i \leq m-1. \end{aligned} \tag{9}$$

Also, using (5), one can obtain $\Delta F(n)$'s. The results are also shown in Table 1.

Here the multiplication of $A = (01110)$ and $B = (10101)$ is shown using Algorithm 2. Table 2 shows contents of various variables of the algorithm as they are updated. The row with n being '-' is for the initialization step (i.e., line 1) of the algorithm.

In order to obtain an overall computation time for a $GF(2^m)$ multiplication using Algorithm 2, the coordinates of the field elements can be divided into $\lceil \frac{m}{\omega} \rceil$ units where ω

corresponds to the data-path width of the processor. We assume that the processor can perform bit-wise XOR and AND of two ω-bit operands using one single XOR instruction and one single AND instruction, respectively. Since the loop in Algorithm 2, has $p-2$ iterations, the total number of bit-wise AND and bit-wise XOR instructions are the same and is $(p-2)\left\lceil\frac{m}{\omega}\right\rceil = (tm-1)\left\lceil\frac{m}{\omega}\right\rceil$. Also, this algorithm needs $2\,(p-2)\left\lceil\frac{m}{\omega}\right\rceil = 2\,(tm-1)\left\lceil\frac{m}{\omega}\right\rceil$ cyclic shifts. We assume that an i-fold, $1 \leq i < \omega$, left/right shift can be emulated in the C programming language using a total of ρ instructions. The value of ρ is typically 4 when simple logical instructions, such as AND, SHIFT, and OR are used. We can now state the following theorem.

Theorem 1. *The dynamic instruction count for Algorithm 2 is given by*

$$\#\mathit{Instructions} \approx 2(1+\rho)\,(tm-1)\left\lceil\frac{m}{\omega}\right\rceil.$$

4 Efficient NB Multiplication over $GF(2^m)$

Although the previous algorithm utilizes the full data-path of the processor for NB multiplication, one can obtain a more efficient algorithm which requires fewer instructions and memory accesses. This algorithm is developed in this section. The cost of this algorithm in terms of dynamic instruction counts and memory requirements is analyzed and then they are compared with those of similar other algorithms.

4.1 Algorithm

For the normal basis $\{\beta,\ \beta^{2^1}, \cdots,\ \beta^{2^{m-1}}\}$, let $\delta_j = \beta^{1+2^j},\ j = 1,\ \cdots,\ v$, where $v = \left\lceil\frac{m-1}{2}\right\rceil$. Then one has the following result from [16].

Lemma 2. *Let A and B be two elements of $GF(2^m)$ and C be their product. Then*

$$C = \begin{cases} \sum\limits_{i=0}^{m-1}\left[a_i b_i \beta^{2^{i+1}} + \left(\sum\limits_{j=1}^{v} x_{i,j}\delta_j^{2^i}\right)\right], & \text{for } m \text{ odd} \\ \sum\limits_{i=0}^{m-1}\left[a_i b_i \beta^{2^{i+1}} + \left(\sum\limits_{j=1}^{v-1} x_{i,j}\delta_j^{2^i}\right) + a_i b_{v+i}\delta_v^{2^i}\right], & \text{for } m \text{ even} \end{cases}$$

where a_i's and b_i's are the NB coordinates of A and B, respectively. Also, indices and exponents are reduced $\bmod m$ *and*

$$x_{i,j} = a_i b_{i+j} + a_{i+j} b_i, \qquad 1 \leq j \leq v,\ 0 \leq i \leq m-1. \tag{10}$$

Let h_j, $1 \leq j \leq v$, be the number of 1's in the normal basis representation of δ_j. Let $w_{j,1}, w_{j,2}, \cdots, w_{j,h_j}$ denote the positions of 1's in the normal basis representation of δ_j, i.e.,

$$\delta_j = \sum_{k=1}^{h_j} \beta^{2^{w_{j,k}}},\ 1 \leq j \leq v, \tag{11}$$

where $0 \leq w_{j,1} < w_{j,2} < \cdots < w_{j,h_j} \leq m-1$. Now, using (11) into Lemma 2, we have the following for m odd.

$$\begin{aligned} C &= \sum_{i=0}^{m-1} a_i b_i \beta^{2^{i+1}} + \sum_{i=0}^{m-1} \sum_{j=1}^{v} x_{i,j} \left(\sum_{k=1}^{h_j} \beta^{2^{w_{j,k}}} \right)^{2^i} \\ &= \sum_{i=0}^{m-1} a_i b_i \beta^{2^{i+1}} + \sum_{i=0}^{m-1} \sum_{j=1}^{v} x_{i,j} \left(\sum_{k=1}^{h_j} \beta^{2^{i+w_{j,k}}} \right) \\ &= \sum_{i=0}^{m-1} a_i b_i \beta^{2^{i+1}} + \sum_{j=1}^{v} \sum_{k=1}^{h_j} \left(\sum_{i=0}^{m-1} x_{i,j} \beta^{2^{i+w_{j,k}}} \right). \end{aligned} \tag{12}$$

Also, for even values of m, one has $v = \frac{m}{2}$ and $\delta_v = \delta_v^{2^{\frac{m}{2}}}$. This implies that in the normal basis representation of δ_v, its i-th coordinate is equal to its $(\frac{m}{2} + i \bmod m)$-th coordinate. Thus, h_v is even and one can write

$$\delta_v = \sum_{k=1}^{\frac{h_v}{2}} (\beta^{2^{w_{v,k}}} + \beta^{2^{w_{v,k}+v}}), \quad v = \frac{m}{2}. \tag{13}$$

Now, using (13) into Lemma 2 (for m even) and using (12), we have the following theorem, where all indices and exponents are reduced modulo m.

Theorem 2. *Let A and B be two elements of $GF(2^m)$ and C be their product. Then*

$$C = \begin{cases} \sum_{i=0}^{m-1} a_i b_i \beta^{2^{i+1}} + \sum_{j=1}^{v} \sum_{k=1}^{h_j} \left(\sum_{i=0}^{m-1} x_{i,j} \beta^{2^{i+w_{j,}}} \right), & \text{for } m \text{ odd} \\ \sum_{i=0}^{m-1} a_i b_i \beta^{2^{i+1}} + \sum_{j=1}^{v-1} \sum_{k=1}^{h_j} \left(\sum_{i=0}^{m-1} x_{i,j} \beta^{2^{i+w_{j,k}}} \right) + F, & \text{for } m \text{ even} \end{cases} \tag{14}$$

where

$$F = \sum_{k=1}^{\frac{h_v}{2}} \sum_{i=0}^{v-1} x_{i,v}(\beta^{2^{i+w_{v,k}}} + \beta^{2^{i+w_{v,k}+v}}),\ \textit{and}\ v = \frac{m}{2}.$$

Note that for a normal basis, the representation of δ_j is fixed and so is $w_{j,k}$, $1 \leq j \leq v$, $1 \leq k \leq h_j$. Now, define

$$\Delta w_{j,k} \triangleq w_{j,k} - w_{j,k-1},\ 1 \leq j \leq v,\ 1 \leq k \leq h_j,\ w_{j,0} = 0, \tag{15}$$

where $w_{j,k}$'s are as given in (11). For a particular normal basis, all $w_{j,k}$'s are fixed. Hence, all $\Delta w_{j,k}$'s need to be determined only at the time of choosing the basis. Using $\Delta w_{j,k}$'s, below we present an efficient NB (ENB) multiplication algorithm over $GF(2^m)$ for odd values of m. The corresponding algorithm for even values of m is shown in Appendix B. Also, an efficient scheme to compute $\Delta w_{j,k}$'s is given later in this section.

Algorithm 3. (ENB Multiplication for m Odd)

Input: $A,\ B \in GF(2^m)$, $\Delta w_{j,k} \in [0, m-1]$, $1 \leq j \leq v$, $1 \leq k \leq h_j$, $v = \frac{m-1}{2}$

Output: $C = AB$

1. Initialize $C := A \odot B$, $S_A := A$, $S_B := B$
2. $C \gg 1$
3. For $j = 1$ to v {
4. $\quad S_A \ll 1$, $S_B \ll 1$
5. $\quad T_A := A \odot S_B$, $T_B := B \odot S_A$
6. $\quad R := T_A + T_B$
7. $\quad$ For $k = 1$ to h_j {
8. $\quad\quad R \gg \Delta w_{j,k}$
9. $\quad\quad C := C + R$
10. $\quad$ }
11. }

In the above algorithm, shifted values of A and B are stored in S_A and S_B, respectively. In line 6, $R \in GF(2^m)$ contains $(x_{0,j},\ x_{1,j},\ \cdots,\ x_{m-1,j})$, i.e., $\sum_{i=0}^{m-1} x_{i,j}\beta^{2^i}$. Also, the right cyclic shift of R in line 8 corresponds to $\sum_{i=0}^{m-1} x_{i,j}\beta^{2^{i+w_{j,k}}}$. After the final iteration, C is the normal basis representation of the required product AB. To illustrate the operation of

j	S_A	S_B	T_A	T_B	k	$\Delta w_{j,k}$	R	C
-	01110	10101	-	-	-	-	-	00010
1	11100	01011	01010	10100			11110	
					1	0	11110	11100
					2	3	11011	00111
2	11001	10110	00110	10001			10111	
					1	3	11110	11001
					2	1	01111	10110

Table 3: Contents of variables in Algorithm 3 for multiplication of $A = (01110)$ and $B = (10101)$.

the above algorithm and compare it with the previous algorithm, we present Example 1 as follows.

Example 2. *Consider the finite field $GF(2^5)$ generated by the irreducible polynomial $F(z) = z^5 + z^2 + 1$ and let α be its root,* i.e., *$F(\alpha) = 0$. We choose $\beta = \alpha^5$, then $\{\beta,\ \beta^2,\ \beta^4,\ \beta^8,\ \beta^{16}\}$ is a type 2 GNB. Here $m = 5$, and $v = \frac{5-1}{2} = 2$. Using Table 2 in [12], one has*

$$\delta_1 = \beta^3 = \beta + \beta^8, \quad h_1 = 2, \quad [w_{1,k}]_{k=1}^{h_1} = [0,\ 3],$$
$$\delta_2 = \beta^5 = \beta^8 + \beta^{16}, \quad h_2 = 2, \quad [w_{2,k}]_{k=1}^{h_2} = [3,\ 4].$$

Let $A = \beta^2 + \beta^4 + \beta^8 = (01110)$ and $B = \beta + \beta^4 + \beta^{16} = (10101)$ which are the same two field elements as we used in Example 1. Table 3 shows contents of various variables of the algorithm as they are updated (see Table 2 for comparison). The row with j being '-' is for the initialization step (i.e., line 1) of the algorithm.

As it can be seen in Algorithm 3, all $\Delta w_{j,k}$'s have to be pre-computed. In the above example, they are determined by calculating δ_j's, which is essentially a multiplication process all by itself. For this multiplication, one can use either Algorithm 1 or Algorithm 2. However, an efficient scheme which does not need multiplication is presented below.

4.2 Algorithm for Computing $w_{j,k}$

Using $A = \beta = (1, 0, 0, \cdots 0)$ and $B = \beta^{2^j} = (\underbrace{0, 0, \cdots 0}_{j}, 1, 0, \cdots 0)$ as inputs in Algorithm 1 or Algorithm 2, one obtains the coordinates of $\delta_j = \beta \cdot \beta^{2^j}, 1 \leq j \leq v$, and, consequently, the values of $\Delta w_{j,k}$'s. Since only one coordinate of each of β and $\beta^{2^j}$is 1, during the calculation of the $w_{j,k}$-th coordinate of δ_j we have at most one non-zero term in the summation of (7) if and only if there exists an $n \in [1, p-2]$ such that the following holds:

$$F(n+1) + w_{j,k} \mod m = 0,$$
$$F(n) + w_{j,k} \mod m = j.$$

Using (5), we then have the following set of equations, for $1 \leq n \leq p-2$, $1 \leq j \leq v$,

$$\begin{aligned} \Delta F(n) &= m - j \mod m, \\ w_{j,k} &= j - F(n) \mod m. \end{aligned} \tag{16}$$

Based on (16), one can have an algorithm for computing δ_j (and hence $\Delta w_{j,k}$'s) which does not require the explicit multiplication of β and β^{2^j} over $GF(2^m)$. The algorithm along with an example for even values of t is given below. An algorithm for odd values of t is given in Appendix C.

Algorithm 4. (*Finding the Coordinates of* δ_j, $1 \leq j \leq \frac{m-1}{2}$.)

Input: $F(n) \in [0,\ m-1]$ for $1 \leq n \leq p-1$

Output: $w_{j,k}$, h_j

1. For $j = 1$ to $\frac{m-1}{2}$ {
2. Initialize $k := 1$, $h_j := 0$
3.   For $n = 1$ to $p - 2$ {
4.   If $(\Delta F(n) = m - j)$ then{
5. $w_{j,k} = j - F(n) \mod m$
6. $k := k + 1$, $h_j := h_j + 1$
7.   }
8.   }
9. }

After executing this algorithm, the values of $w_{j,k}$'s have to be sorted such that $0 \leq w_{j,1} < w_{j,2} < \cdots < w_{j,h_j} \leq m-1$. By applying this algorithm to type 2 GNB over $GF(2^5)$, the same field we have seen in Example 2, and using the corresponding values of $F(n)$ (see Table 1), one can obtain

$$\begin{aligned} j=1,\ \Delta F(3) = \Delta F(9) = 4, \quad h_1 = 2, \quad [w_{1,k}]_{k=1}^{h_1} = [0,\ 3], \\ j=2,\ \Delta F(6) = \Delta F(8) = 3, \quad h_2 = 2, \quad [w_{2,k}]_{k=1}^{h_2} = [3,\ 4]. \end{aligned}$$

For each even value of t, this algorithm produces t coordinates for each δ_j, $1 \leq j \leq \frac{m-1}{2}$, i.e., $\delta_j = \sum_{k=1}^{t} \beta^{2^{w_{j,k}}}$. However, for $t > 2$, there may exist k_1 and k_2 such that $w_{j,k_1} = w_{j,k_2}$. Thus, $\beta^{2^{w_{j,k_1}}}$ and $\beta^{2^{w_{j,k_2}}}$ in the NB representation of δ_j cancel each other and we have $\delta_j = \sum_{k=1}^{h_j} \beta^{2^{w_{j,k}}}$, where $h_j \leq t$. This property is shown for $m = 13$, 17 and 19 in the next section.

4.3 Cost and Comparison

In an effort to determine the cost of Algorithm 3, we give the dynamic instruction counts for its software implementation. We also consider the number of memory accesses to read the pre-computed values of $\Delta w_{j,k}$. For software implementation of the above algorithm, one would heavily relay on instructions, such as, XOR, AND and others which can be used to emulate cyclic shifts (in the C like programming language). XOR instructions are needed in lines 6 and 9, which are repeated v and $\sum_{j=1}^{v} h_j$ times, respectively. Since $v = \frac{m-1}{2}$ and $\sum_{j=1}^{v} h_j = \frac{C_N - 1}{2}$ [10], the total number of XOR instructions is $\frac{1}{2}(C_N + m - 1)\left\lceil \frac{m}{\omega} \right\rceil$. Because of the $\odot$ operations in lines 1 and 5, one can also see that the above algorithm requires $m \left\lceil \frac{m}{\omega} \right\rceil$ AND instructions. We assume that each i-fold cyclic shift, $1 \leq i \leq m-1$, in lines 2, 4 and 8 needs $\rho \left\lceil \frac{m}{\omega} \right\rceil$ instructions where ρ is as defined earlier. In Algorithm 3, the number of cyclic shifts in lines 2, 4 and 8 are 1, $2v$ and $\sum_{j=1}^{v} h_j$, respectively. Thus, the total number of cyclic shifts in this algorithm is $1 + 2v + \sum_{j=1}^{v} h_j = \frac{1}{2}(C_N + 2m - 1)$ and so the total number of instructions to emulate cyclic shifts used in Algorithm 3 is $\frac{\rho}{2}(C_N + 2m - 1)\left\lceil \frac{m}{\omega} \right\rceil$. Based on the above discussion, we have the following theorem.

Theorem 3. *The dynamic instruction count for Algorithm 3 is given by*

$$\#\mathit{Instructions} \approx \left(\frac{1+\rho}{2} C_N + \frac{3+2\rho}{2} m - \frac{2+\rho}{2} \right) \left\lceil \frac{m}{\omega} \right\rceil.$$

Algorithms	# Instructions		
	XOR	AND	Others (to emulate $\gg$ / $\ll$)
Alg. 1	$\frac{1}{4}m\,(tm-1)$	$m\,(tm-1)$	$2\rho m \left\lceil \frac{m}{\omega} \right\rceil$
Alg. 2	$(tm-1)\left\lceil \frac{m}{\omega} \right\rceil$	$(tm-1)\left\lceil \frac{m}{\omega} \right\rceil$	$2\rho\,(tm-1)\left\lceil \frac{m}{\omega} \right\rceil$
Alg. 3	$\frac{1}{2}\,(C_N+m-2)\left\lceil \frac{m}{\omega} \right\rceil$	$m\left\lceil \frac{m}{\omega} \right\rceil$	$\frac{\rho}{2}\,(C_N+2m-1)\left\lceil \frac{m}{\omega} \right\rceil$
Ratio of Alg. 2 to Alg. 3	$\approx \frac{2t}{t+1}$	$\approx t$	$\approx \frac{4t}{t+1}$

Table 4: Comparison of multiplication algorithms in terms of number of instructions.

Algorithms	Memory	
	Size in bits	# Accesses
Alg. 1	$(tm-1)\lceil \log_2 m \rceil$	$2m(tm-1)$
Alg. 2	$\frac{tm}{2}\lceil \log_2 m \rceil$	tm
Alg. 3	$\frac{C_N-1}{2}\lceil \log_2 m \rceil$	$\frac{C_N-1}{2}$
Ratio of Alg. 2 to Alg. 3	≈ 1	≈ 2

Table 5: Comparison of multiplication algorithms in terms of memory requirements.

For software implementation of Algorithm 3, if the loops are not unrolled and the values of $\Delta w_{j,k}$'s are not hard-coded, one needs to store $\Delta w_{j,k}$, $1 \le j \le v$, $1 \le k \le h_j$. Since the total number of $\Delta w_{j,k}$'s is $\sum_{j=1}^{v} h_j$ and each $\Delta w_{j,k} \in [0, m-1]$ needs $\lceil \log_2 m \rceil$ bits of memory, a total of about $\frac{C_N-1}{2}\lceil \log_2 m \rceil$ bits of memory is needed to store the pre-computed $\Delta w_{j,k}$'s.

Table 4 compares the number of dynamic instructions of the three algorithms we have described so far. As it can be seen in Table 4, both our proposed schemes (i.e., Algorithms 2 and 3) are superior to the conventional bit-level multiplication scheme (i.e., Algorithm 1). Also, Table 5 gives memory sizes and numbers of memory accesses of these algorithms. The final rows of these tables give approximate improvement factors of Algorithm 3 to Algorithm 2. A more detailed comparison of these two algorithms are given in Table 6 and for the five binary fields recommended by NIST for ECDSA (elliptic curve digital signature algorithm) [13].

These algorithms have been coded in software using the C programming language. Table

Parameters			Algorithm 2, Algorithm 3					
			# Instructions				Memory	
m	t	C_N	XOR	AND	Others/ρ	Total ($\rho = 4$)	Size in bits	# Accesses
163	4	645	3906, 2418	3906, 978	7812, 2910	39060, 15036	2608, 2576	652, 322
233	2	465	3720, 2784	3720, 1864	7440, 3720	37200, 19528	1864, 1856	466, 232
283	6	1677	15273, 8811	15273, 2547	30546, 10089	152730, 51714	7641, 7542	1698, 838
409	4	1629	21255, 13234	21255, 5317	42510, 15899	212550, 82147	7362, 7326	1636, 814
571	10	5637	102762, 55854	102762, 10278	205524, 61002	1027620, 310140	28550, 28180	5710, 2818

Table 6: Comparison of the proposed algorithms for binary fields recommended by NIST for ECDSA applications ($\omega = 32$).

7 shows timing (in μs) for these codes executed on Pentium III 533 MHz PC[2]. Our codes are parameterized in the sense that they can be used for various m and t without major modifications. For high speed implementation, the codes can be optimized for special values of m and t.

Agnew et. al. in [1] have proposed a bit-serial architecture for the NB multiplication. Although their work has been targeted to hardware implementation, the main idea can be used for software implementation similar to the vector level method proposed here. For such a software implementation of [1], one would require $(C_N - 1)\left\lceil \frac{m}{\omega} \right\rceil$ XOR instructions, $m\left\lceil \frac{m}{\omega} \right\rceil$ AND instructions, and $\rho\,(C_N + m - 1)\left\lceil \frac{m}{\omega} \right\rceil$ other instructions. Thus, the dynamic instruction count would be $(\rho + 1)\,(C_N + m - 1)\left\lceil \frac{m}{\omega} \right\rceil$ which is about twice of that in Algorithm 3 (see Theorem 3). In [18], one can find software implementation of the NB multiplication for two special cases, namely, two optimal normal bases. The method used in [18] is similar to that of the NB multiplication of [1].

Some of the recently proposed *polynomial basis* multiplication algorithms, for example [6], [9], create a look-up table on the fly based on one of the inputs (say B) and yield significant speed-ups by processing a group of bits of the other input (i.e., A) at a time. At

[2]The PC has 64 M bytes of RAM, 32 K bytes of L1 cache and 512 K bytes of L2 cache.

Parameters				Algorithm 2, Algorithm 3	
				Timing	
m	t	u	C_N	in μs	Ratio of Alg. 2 to Alg. 3
163	4	149	645	307, 99	3.1
233	2	466	465	346, 126	2.75
283	6	398	1677	1005, 318	3.16
409	4	316	1629	1466, 473	3.1
571	10	3421	5637	8423, 2949	2.86

Table 7: Actual timing of the proposed algorithms for binary fields recommended by NIST for ECDSA applications.

this point, it is not clear whether such a group-level processing of A can be incorporated into our Algorithm 3. However, if m is a composite number, then one can essentially achieve similar kind of group-level processing by performing computations in the sub-fields. This idea is explored in the following section.

5 Efficient Composite Field NB Multiplication Algorithm

In this section, we consider multiplications in the finite field GF(2^m) where m is a composite number. These fields are referred to as composite fields and have been used in the recent past to develop efficient multiplication schemes [14], [15]. When these fields are to be used for elliptic curve crypto-systems, one must choose m such that its factor are large enough to resist the attack described by Galbraith and Smart [4].

Lemma 3. *[11] Let* $\gcd(m_1, m_2) = 1$. *Let* $N_1 = \{\beta_1^{2^j} \mid 0 \le j \le m_1 - 1\}$ *be a normal basis of*

$GF(2^{m_1})$ over $GF(2)$. Then N_1 is also a normal basis of $GF(2^{m_1 m_2})$ over $GF(2^{m_2})$.

Here, we consider composite fields with only two prime factors[3] (i.e., both m_1 and m_2 are prime). Thus, in the following we give all equations and algorithms for odd degrees (i.e., m_1 and m_2). The reader can easily extend it for even degrees using the results of the previous section. Also, the parameters, namely δ_j, h_j, v, β, and $\Delta w_{j,k}$ of the previous section are used here in the context of the sub-fields $GF(2^{m_1})$ and $GF(2^{m_2})$ by putting an extra sub/superscript for example $\delta_j^{(1)}$ for $GF(2^{m_1})$ and $\delta_j^{(2)}$ for $GF(2^{m_2})$.

Let A and B be two elements of $GF(2^{m_1})$ over $GF(2)$ and C be their product. Then we have the following from [17].

$$C = \sum_{i=0}^{m_1-1} a_i b_i \beta_1^{2^i} + \sum_{j=1}^{v_1} \sum_{k=1}^{h_j^{(1)}} \left(\sum_{i=0}^{m_1-1} y_{i,j} \beta_1^{2^{i+w_{j,k}^{(1)}}} \right), \text{for } m_1 \text{ odd} \tag{17}$$

where

$$y_{i,j} = (a_i + a_{i+j})(b_i + b_{i+j}),\ 1 \le j \le v_1,\ 0 \le i \le m_1 - 1,$$

$$v_1 = \frac{m_1 - 1}{2}, \quad \beta_1^{2^j+1} = \sum_{k=1}^{h_j^{(1)}} \beta_1^{2^{w_{j,k}^{(1)}}}.$$

By combining Lemma 3 with (17), the following is obtained.

Lemma 4. *Let $A = (A_0, A_1, \cdots, A_{m_1-1})$ and $B = (B_0, B_1, \cdots, B_{m_1-1})$ be two elements of $GF(2^{m_1 m_2})$ over $GF(2^{m_2})$ and C be their product. Then*

$$C = \sum_{i=0}^{m_1-1} A_i B_i \beta_1^{2^i} + \sum_{j=1}^{v_1} \sum_{k=1}^{h_j^{(1)}} \left(\sum_{i=0}^{m_1-1} Y_{i,j} \beta_1^{2^{i+w_{j,k}^{(1)}}} \right), \textit{for } m_1 \textit{ odd} \tag{18}$$

where

$$Y_{i,j} = (A_i + A_{i+j})(B_i + B_{i+j}),\ 1 \le j \le v_1,\ 0 \le i \le m_1 - 1, \tag{19}$$

and $A_i = (a_{i,0}, a_{i,1}, \cdots, a_{i,m_2-1})$, $B_i = (b_{i,0}, b_{i,1}, \cdots, b_{i,m_2-1}) \in GF(2^{m_2})$ are sub-field coordinates of A and B.

[3]This is mainly because our target application of this work is elliptic curve crypto-systems. For such systems in today's security applications, the values of m appear to be in the range of 160 to several hundreds only (571 as given in [13]). To avoid the attack of [4], one however may like to choose m such that it has no small factors such as 2, 3, 5, 7, 11. This basically makes one to choose m as the product of two primes.

Lemma 4 leads to an algorithm for multiplication in composite fields using normal basis. The algorithm is stated below.

Algorithm 5. (ECFNB Multiplication of $GF(2^{m_1 m_2})$ over $GF(2^{m_2})$)

Input: $A,\ B \in GF(2^m),\ \Delta w^{(1)}_{j,k} \in [0, m_1 - 1],\ 1 \leq j \leq v_1,\ v_1 = \frac{m_1-1}{2},\ 1 \leq k \leq h^{(1)}_j$

Output: $C = AB$

1. Initialize $C := A \otimes B,\ S_A := A,\ S_B := B$
2. For $j = 1$ to v_1 {
3. $\quad S_A \ll m_2,\ S_B \ll m_2$
4. $\quad T_A := A + S_A,\ T_B := B + S_B$
5. $\quad R := T_A \otimes T_B$
6. $\quad$ For $k = 1$ to $h^{(1)}_j$ {
7. $\quad\quad R \gg m_2 \Delta w^{(1)}_{j,k}$
8. $\quad\quad C := C + R$
9. $\quad$ }
10. }

In lines 1 and 5 of Algorithm 5, $A \otimes B = (A_0B_0,\ A_1B_1, \cdots,\ A_{m_1-1}B_{m_1-1})$ denotes parallel sub-field multiplications of A and B. This sub-field multiplication can be implemented with an extension of Algorithm 3 such that it produces m_1 sub-field multiplications over $GF(2^{m_2})$. This is shown in Algorithm 6 where $A \rhd i$ (resp. $A \lhd i$) $0 \leq i \leq m_2 - 1$, denotes an i-fold right (resp. left) sub-field cyclic shift of all sub-field elements of A, i.e., $A_0,\ A_1, \cdots,\ A_{m_1-1}$, respectively.

Algorithm 6. (Parallel Sub-Field Multiplication over $GF(2^{m_2})$)

Input: $A,\ B \in GF(2^m),\ \Delta w^{(2)}_{j,k} \in [0, m_2 - 1],\ 1 \leq j \leq v_2,\ 1 \leq k \leq h^{(2)}_j,\ v_2 = \frac{m_2-1}{2}$

Output: $C = A \otimes B$

1. Initialize $C := A \odot B,\ S_A := A,\ S_B := B$
2. $C \rhd 1$
3. For $j = 1$ to v_2 {
4. $\quad S_A \lhd 1,\ S_B \lhd 1$
5. $\quad T_A := A \odot S_B,\ T_B := B \odot S_A$
6. $\quad R := T_A + T_B$

# Instructions	XOR	$\frac{m_1}{2}\left[(C_1+2m_1-3)+\frac{(m_1+1)}{2}(C_2+m_2-2)\right]$
	AND	$\frac{m_1 m_2 (m_1+1)}{2}$
	Others	$\frac{\mu m_1}{4}(m_1+1)(C_2+2m_2-1)$
Memory	Size in bits	$\frac{C_1-1}{2}\lceil \log_2 m_1 \rceil + \frac{C_2-1}{2}\lceil \log_2 m_2 \rceil$
	# Accesses	$\frac{C_1-1}{2}+\frac{(m_1+1)(C_2-1)}{4}$

Table 8: Cost of Algorithm 5.

7. For $k = 1$ to $h_j^{(2)}$ {
8. $R \triangleright \Delta w_{j,k}^{(2)}$
9. $C := C + R$
10. }
11. }

In order to obtain the cost of Algorithm 5, we need to evaluate the cost of Algorithm 6 which is called $1 + v_1 = \frac{m_1+1}{2}$ times by the former. Like Algorithm 3, one can determine the dynamic instruction counts of Algorithm 6 to be $\frac{1}{2}(C_2 + m_2 - 2)$ XOR, m_2 AND and $\frac{1}{2}(C_2+2m_2-1)$ others to emulate cyclic shifts. The total cost of Algorithm 5 also depends on how sub-field elements, each of m_2 bits, are stored in registers. For the sake of simplicity we assume that an element of $GF(2^{m_2})$ is stored in one ω-bit register (for software implementation of elliptic curve crypto-systems with both m_1 and m_2 being prime, most general purpose processors would have ω bit registers where $\omega \geq m_2$). For $\omega = 24$ and 32, the best values of m_2 are those which have ONBs, i.e., 23 and 29, respectively. Thus, each element of $GF(2^m)$ needs m_1 registers and the cyclic shifts in lines 3 and 7 of Algorithm 5 are almost free of cost (or at best register renaming). Based on this assumption, we give the dynamic instruction counts of Algorithm 5 in Table 8. In this table, μ is the number of instructions needed for one sub-field cyclic shift in each register and it is 4 in the C programming language.

Table 9 shows the number of instructions and memory requirements of Algorithm 5 for six different composite fields. These six fields are obtained by combining three m_1's and two m_2's. Algorithm 5 is also coded for these composite fields using the C programming language. The actual timing (in μs) of Algorithm 5 are also shown in Table 9.

In order to obtain parameters of the finite fields $GF(2^{m_1})$ and $GF(2^{m_2})$ used in Algorithm

Parameters					# Instructions				Memory		Actual timing
m	m_1	m_2	C_1	C_2	XOR	AND	Others	Total	Size in bits	# Accesses	(in μs)
299	13	23	45	45	3445	2093	16380	21918	198	176	114
377	"	29	"	57	4264	2639	20748	27651	228	218	150
391	17	23	81	45	6001	3519	27540	37060	310	238	188
437	19	"	117	"	7714	4370	34200	46284	400	278	249
493	17	29	81	57	7378	4437	34884	46699	340	292	242
551	19	"	117	"	9424	5510	43320	58254	430	338	309

Table 9: Cost of Algorithm 5 for certain composite fields ($\mu = 4$).

5, we have used Algorithm 4 for the values of m_1 and m_2 shown in Table 9. The results are presented in Tables 10 and 11. In Table 10, the outputs of Algorithm 4 are shown after arranging $w_{i,j}$'s in ascending order. Also, $w^{(1)}_{j,k}$'s are obtained from the outputs of Algorithm 4 after removing two same $w_{j,k}$ in each row of this table. In Table 11, the outputs of Algorithm 4 are equal to $w^{(2)}_{j,k}$'s. It is worth mentioning that $w^{(1)}_{j,k}$'s and $w^{(2)}_{j,k}$'s in these tables match with those presented in Table 2 of [10].

6 Conclusions

In this document, a number of software algorithms for normal basis multiplication over $GF(2^m)$ have been presented. Both Algorithms 2 and 3 make maximal use of the full width of the data-path of the processor on which the software is to be executed and they provide significant speed-ups compared to the conventional bit-level multiplication scheme (i.e., Algorithm 1). Algorithms 2 and 3 are particularly suitable if m is a prime. Such values of m are of importance, especially for designing high speed crypto-systems based on Koblitz curves and for protecting elliptic curve crypto-systems against the attack of Galbraith and Smart [4]. Both Algorithms 2 and 3 have been coded for software implementation using C, and our timing results show that Algorithm 3 is about 200% faster that Algorithm 2. These results are for those five Gaussian normal bases over the binary fields which NIST has described in their ECDSA document [13]. For the purpose of using NIST parameters, although we have presented our results for Gaussian normal bases, our algorithms are quite

$m_1(u,t)$	j	Outputs of Alg. 4	$w_{j,k}^{(1)}$	$\Delta w_{j,k}^{(1)}$	$h_j^{(1)}$
13(23,4)	1	0,3,4,5	0,3,4,5	0,3,1,1	4
	2	4,8,11,12	4,8,11,12	4,4,3,1	4
	3	1,7,8,12	1,7,8,12	1,6,1,4	4
	4	1,2,10,12	1,2,10,12	1,1,8,2	4
	5	1,7,8,10	1,7,8,10	1,6,1,2	4
	6	6,6,9,11	9,11	9,2	2
17(47,6)	1	0,3,9,11,13,13	0,3,9,11,	0,3,6,2	4
	2	6,7,8,9,12,16	6,7,8,9,12,16	6,1,1,1,3,4	6
	3	1,6,9,11,13,14	1,6,9,11,13,14	1,5,3,2,2,1	6
	4	4,4,5,5,7,15	7,15	7,8	2
	5	4,4,7,10,12,15	7,10,12,15	7,3,2,3	4
	6	2,3,7,9,14,15	2,3,7,9,14,15	2,1,4,2,5,1	6
	7	2,4,5,6,12,15	2,4,5,6,12,15	2,2,1,1,6,3	6
	8	2,9,10,11,14,16	2,9,10,11,14,16	2,7,1,1,3,2	6
19(7,10)	1	0,2,4,6,7,11,12,13,15,18	0,2,4,6,7,11,12,13,15,18	0,2,2,2,1,4,1,1,2,3	10
	2	1,2,2,9,11,11,13,13,14,15	1,9,14,15	1,8,5,1	4
	3	0,0,7,7,10,10,11,13,15,18	11,13,15,18	11,2,2,3	4
	4	1,5,6,7,8,10,13,15,16,16	1,5,6,7,8,10,13,15	1,4,1,1,1,2,3,2	8
	5	0,0,4,5,5,7,10,11,14,18	4,7,10,11,14,18	4,3,3,1,3,4	6
	6	1,4,7,8,8,9,10,14,15,18	1,4,7,9,10,14,15,18	1,3,3,2,1,4,1,3	8
	7	1,3,3,4,5,6,8,16,16,17	1,4,5,6,8,17	1,3,1,1,2,9	6
	8	4,6,6,7,9,10,10,11,13,16	4,7,9,11,13,16	4,3,2,2,2,3	6
	9	2,6,8,12,12,13,14,15,17,17	2,6,8,13,14,15	2,4,2,5,1,1	6

Table 10: Parameters of type t GNBs used in Algorithm 5.

$m_2 = 23$ $(u = 46)$			$m_2 = 29$ $(u = 58)$		
j	$w_{j,k}^{(2)}$	$\Delta w_{j,k}^{(2)}$	j	$w_{j,k}^{(2)}$	$\Delta w_{j,k}^{(2)}$
1	0,19	0,19	1	0,21	0,21
2	9,19	9,10	2	6,21	6,15
3	12,15	12,3	3	13,18	13,5
4	5,6	5,1	4	11,27	11,16
5	4,13	4,9	5	17,20	17,3
6	4,16	4,12	6	2,22	2,20
7	13,21	13,8	7	13,25	13,12
8	11,18	11,7	8	9,10	9,1
9	2,20	2,18	9	8,14	8,6
10	15,17	15,2	10	8,26	8,18
11	8,14	8,6	11	4,14	4,10
			12	17,24	17,7
			13	3,7	3,4
			14	9,11	9,2

Table 11: Parameters of type 2 GNBs used in Algorithm 6 ($t = 2,\ h_j^{(2)} = 2$).

generic and can be used for any normal bases of $GF(2^m)$ over $GF(2)$.

We have also considered composite fields with $m = m_1 \cdot m_2$. To avoid the attack of [4] on elliptic curve crypto-systems defined over these composite fields, we choose both m_1 and m_2 to be prime. We have presented an algorithm (i.e., Algorithm 5) for normal basis multiplication for $GF(2^m)$ over $GF(2^{m_2})$. Our results show that for similar values of m, Algorithm 5 can be much more efficient than Algorithm 3. For example, the actual timing of Algorithm 3 is 318 micro-seconds for $GF(2^{283})$ whereas the timing of Algorithm 5 is 114 micro-seconds for $GF(2^{299})$. Composite fields also provide an added flexibility to hardware-software co-design of finite field processors. For example, Algorithm 6 which is *called* by Algorithm 5 a total of $\frac{m_1+1}{2}$ times, can be implemented in hardware for small values of m_2, and Algorithm 5 can be embedded in a micro-controller which would give us a high speed, yet quite flexible, normal basis multiplier over very large fields.

Acknowledgment

The authors would like to thank Z. Zhang for his help with implementing the algorithms and getting their timing results.

Appendices

A Vector-Level NB Multiplication for t Odd

In this appendix, we discuss the vector-level NB multiplication algorithm when t is odd. This algorithm which is similar to Algorithm 2 on page 7, requires the input sequence $F(1)$, $F(2)$, $\cdots$, $F(p-1)$ to be pre-computed using (4), i.e.,

$$F(2^i u^j \mod p) = i,\ 0 \leq i \leq m-1,\ 0 \leq j < t,$$

where $p = tm + 1$ and u is an integer of order $t \mod p$. Since p should be prime, tm has to be an even integer. Also, t is odd which implies that m should be even. The i-th coordinate of the product $C = AB$ can be obtained as [13]

$$c_i = f + \sum_{n=1}^{p-2} a_{F(n+1)+i} b_{F(p-n)+i},\ 0 \le i \le m-1, \tag{20}$$

where

$$f = \sum_{n=1}^{\frac{m}{2}} (a_{n+i-1} b_{\frac{m}{2}+n+i-1} + a_{\frac{m}{2}+n+i-1} b_{n+i-1}). \tag{21}$$

In (21), $f \in GF(2)$ is independent of i. For $1 \le n \le p-2$, let us use the definition (5) on page 6, i.e., $\Delta F(n) = F(n+1) - F(n) \mod m$, where $F(1) = 0$. Let us introduce

$$\Delta F'(n) = F(p-n) - F(p-n+1) \mod m,$$

where $F(p) = 0$. It is easy to see that $\Delta F'(n) = -\Delta F(p-n+1) \mod m$. We now state the vector-level algorithm for t odd as follows.

Algorithm 7. (Vector-Level NB Multiplication for t odd)

Input: $A,\ B \in GF(2^m),\ \Delta F(n),\ \Delta F'(n) \in [0, m-1],\ 1 \le n \le p-1$

Output: $C = AB$

Initialize $S_A := A,\ S_B := B,\ C := 0,\ f := 0$

  $S_B \ll \frac{m}{2}$

  $R := S_A \odot S_B,\ R = (r_0,\ r_1, \cdots,\ r_{m-1})$

For $i = 0$ to $m-1$ {

  $f := f + r_i$

}

$S_B := B$

For $n = 1$ to $p-2$ {

  $S_A \ll \Delta F(n)$

  $S_B \ll \Delta F'(n)$

  $R := S_A \odot S_B$

  $C := C + R$

}

If f is 1, $C := C + (1, 1, \cdots, 1, 1)$

B Efficient NB Multiplication Algorithm for m Even

Algorithm 8. (ENB Multiplication for m even)

Input: $A,\ B \in GF(2^m),\ \Delta w_{j,k},\ 1 \le j \le v,\ 1 \le k \le h_j$

Output: $C = AB$

Initialize $C := A \odot B,\ S_A := A,\ S_B := B$

$C \gg 1$

For $j = 1$ to $v - 1$ {

  $S_A \ll 1,\ S_B \ll 1$

  $T_A := A \odot S_B,\ T_B := B \odot S_A$

  $R := T_A + T_B$

  For $k = 1$ to h_j {

    $R \gg \Delta w_{j,k}$

    $C := C + R$

  }

}

$S_A \ll 1,\ S_B \ll 1$

$T_A := A \odot S_B,\ T_B := B \odot S_A$

$R := T_A + T_B$

For $k = 1$ to $\frac{h_v}{2}$ {

  $R \gg \Delta w_{v,k}$

  $C := C + R$

}

C Algorithm for Computing $w_{j,k}$ for t Odd

The NB coordinates of $\delta_j = \beta \cdot \beta^{2^j},\ 1 \le j \le \frac{m}{2}$, can be obtained by letting $A = \beta = (1, 0, 0, \cdots 0)$ and $B = \beta^{2^j} = (\underbrace{0, 0, \cdots 0}_{j}, 1, 0, \cdots 0)$ in Algorithm 7. Using (21), one has

$$f = \begin{cases} 0, & 1 \le j < \frac{m}{2} \\ 1, & j = \frac{m}{2}. \end{cases} \tag{22}$$

Using equations (20) and (22) and following the discussion related to Algorithm 4 on page 13, one can obtain the following.

Algorithm 9. (*Finding the Coordinates of* δ_j, $1 \le j \le \frac{m}{2}$.)

Input: $F(n) \in [0,\ m-1]$ for $1 \le n \le p-1$

Output: $w_{j,k},\ h_j$

```
For j = 1 to m/2 {
Initialize l := 1, h'_j := 0
  For n = 1 to p-2 {
    If (F(p-n) - F(n+1) mod m = j) then{
      w'_{j,l} = j - F(p-n) mod m
      l := l+1, h'_j := h'_j + 1
    }
  }
}
 h_j = h'_j, 1 <= j < m/2, h_{m/2} = m - h'_{m/2}
 w_{j,k} = w'_{j,l}, 1 <= j < m/2,
 w_{m/2,k} ∈ {0, 1, ..., m-1} - {w'_{m/2,l} : 1 <= l <= h'_{m/2}}
```

Similar to Algorithm 4, the $w_{j,k}$'s from this algorithm should be sorted at the end.

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] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. "An Implementation of Elliptic Curve Cryptosystems Over $F_{2^{155}}$". *IEEE J. Selected Areas in Communications*, 11(5):804–813, June 1993.

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

[4] S. D. Galbraith and N. Smart. A Cryptographic Application of Weil Descent. In *Proceedings of the Seventh IMA Conf. on Cryptography and Coding, LNCS 1764*, pages 191–200. Springer-Verlag, 1999.

[5] S. Gao and Jr. H. W. Lenstra. "Optimal Normal Bases". *Designs, Codes and Cryptography*, 2:315–323, 1992.

[6] M. A. Hasan. Look-up Table-Based Large Finite Field Multiplication in Memory Constrained Cryptosystems. *IEEE Trans. Comput.*, 49:749–758, July 2000.

[7] M. A. Hasan, M. Z. Wang, and V. K. Bhargava. "A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields". *IEEE Transactions on Computers*, 42(10):1278–1280, Oct. 1993.

[8] D. Johnson and A. Menezes. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". *Technical Report CORR 99-34, Dept. of C & O, University of Waterloo, Canada*, August 23 1999. Updated: Feb. 24, 2000.

[9] J. Lopez and R. Dahab. High Speed Software Multiplication in F_{2^m}. In *Proceedings of Indocrypt 2000*, pages 203–212. LNCS 1977, Springer, 2000.

[10] Chung-Chin Lu. "A Search of Minimal Key Functions for Normal Basis Multipliers". *IEEE Transactions on Computers*, 46(5):588–592, May 1997.

[11] A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullin, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. Kluwer Academic Publishers, 1993.

[12] 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.

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

[14] S. Oh, C. H. Kim, J. Lim, and D. H. Cheon. "Efficient Normal Basis Multipliers in Composite Fields". *IEEE Transactions on Computers*, 49(10):1133–1138, Oct. 2000.

[15] C. Paar, P. Fleishmann, and P. Soria-Rodriguez. "Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents". *IEEE Transactions on Computers*, 48(10):1025–1034, Oct. 1999.

[16] A. Reyhani-Masoleh and M. A. Hasan. "A Reduced Redundancy Massey-Omura Parallel Multiplier over $GF(2^m)$". In 20^{th} *Biennial Symposium on Communications*, pages 308–312, Kingston, Ontario, Canada, May 2000.

[17] A. Reyhani-Masoleh and M. A. Hasan. "On Efficient Normal Basis Multiplication". In *LNCS 1977 as Proceedings of Indocrypt 2000*, pages 213–224, Calcutta, India, December 2000. Springer Verlag.

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

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