Fast Normal Basis Multiplication Using General Purpose Processors

Arash Reyhani-Masoleh, *Member*, *IEEE*, and M. Anwar Hasan, *Senior Member*, *IEEE*

Abstract—For cryptographic applications, normal bases have received considerable attention, especially for hardware implementation. In this article, we consider fast software algorithms for normal basis multiplication over the extended binary field $\mathrm{GF}(2^m)$. 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 codesign of processors for very large finite fields.

Index Terms—Finite field multiplication, normal basis, software algorithms, ECDSA, composite fields.

1 Introduction

The extended binary finite field $\mathrm{GF}(2^m)$ 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 1,024 in the standard discrete log-based cryptography. Elliptic curve crypto-systems, which were proposed by Koblitz [15] and Miller [23] independently, use relatively smaller field sizes, but require a 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 $\mathrm{GF}(2^m)$ arithmetic operations are addition and multiplication. Compared to the former, the latter is a much more complicated and time-consuming operation. The complexity of $\mathrm{GF}(2^m)$ multiplication very much depends 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], [19], [10], [35]). Unlike hardware, so far software implementation of a $\mathrm{GF}(2^m)$ multiplier using normal bases has not been that popular. 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 paper, we consider algorithms for fast software normal basis (NB) 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. In view of this, a vector-level normal basis multiplication algorithm is presented 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 present implementation results of these schemes for the five fields recommended by NIST for NB multiplication in ECDSA (elliptic curve digital signature algorithm) [25], i.e., $m = 163$, 233, 283, 409, 571. For example, it takes $99\mu s$ and requires only 322 bytes of memory for a $GF(2^{163})$ multiplication using NB on Pentium III 533 MHz PC. 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. For example, it requires $114\mu s$ and 25 bytes of memory for multiplication over $GF(2^{299})$. Composite fields also offer a great deal of flexibility toward hardware-software codesign of very large finite field processors.

The organization of this article is as follows: In Section 2, the NB representation and its conventional multiplication algorithm are presented. This algorithm has been proposed by NIST for NB multiplication in ECDSA [25] where all

- *A. Reyhani-Masoleh is with the Centre for Applied Cryptographic Research, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. E-mail: areyhani@math.uwaterloo.ca.*
- *M.A. Hasan is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. E-mail: ahasan@ece.uwaterloo.ca.*

Manuscript received 26 July 2001; revised 16 Sept. 2002; accepted 24 Jan. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114607.

recommended values of m are prime. Then, in Section 3, we extend NIST's algorithm to a vector-level multiplication algorithm which uses the full width of the data-path of the processor. In Section 4, a more efficient algorithm for NB multiplication is presented. In this section, this algorithm is also compared with other algorithms. Then, we present an algorithm for multiplication over finite fields $GF(2^m)$ with composite values of m in Section 5. Finally, a few concluding remarks are given in Section 6.

2 Preliminaries

2.1 Normal Basis Representation

It is well-known that there exists a normal basis 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 $a_i \in GF(2)$, $0 \le i \le m-1$, is the ith coordinate of A. In this paper, this normal basis representation of A will be written in short as $A = (a_0, a_1, \cdots, a_{m-1})$. Now, consider the following matrix:

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

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 [23], we now give the definition of the complexity of an NB as follows.

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

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

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 1s, in* $\mathbf{M}_i$.

It is well-known that $C_N \ge 2m-1$ [23]. When $C_N = 2m-1$, the NB is called an optimal normal basis (ONB). Two types of ONBs were constructed by Mullin et al. [23]. Gao and Lenstra [7] 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 [4] 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 [4]. 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 [12].

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 [11] for an algorithm with t odd). The case of t even is of particular interest for implementing high speed cryptosystems 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. Let $A = (a_0, a_1, \cdots, a_{m-1})$ and $B = (b_0, b_1, \cdots, b_{m-1})$ be the elements of $GF(2^m)$, then the ith coordinate of the product $C = AB$ is computed as [13]:

$$c_i = \sum_{n=1}^{p-2} a_{F(n+1)+i} b_{F(p-n)+i}, \; 0 \le i \le m-1. \tag{4}$$

In (4), $p = tm+1$ and

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

where u is an integer of order $t \bmod p$. In order to realize (4), the following algorithm can be used [11] where c_i is computed in the inner loop of this algorithm. Note that, in the following algorithm, $A \ll i$ (resp. $A \gg i$) denotes the i-fold left (resp. right) cyclic shifts of the coordinates of A. The algorithm uses the fact that the $(i+j)$th coordinate of A (i.e., a_{i+j}) is equal to the jth coordinate of $A \ll i$. It also requires the input sequence $F(1), F(2), \cdots, F(p-1)$ to be precomputed using (5).

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 $C = (c_0, c_1, \cdots, c_{m-1}) := 0$
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.      $A \ll 1, B \ll 1$
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. Second, in line 4, the $\bmod 2$ *multiplication* of the coordinates, which is implemented by bit-level logical AND operation, is performed $m(p-2)$ times in total and the $\bmod 2$ *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 $\bmod 2$ multiplication and addition operations correspond to about $m(p-2)$ AND and $\frac{1}{4}m(p-2)$ XOR instructions,[1] respectively. It is worth mentioning here that, although each XOR/AND instruction of the processor is capable of working on 16 or 32-bit words (i.e., the processor's data-path is 16/32 bits wide), the above algorithm does not make use of the full data-path.

3 Vector-Level NB Multiplication

In this section, we discuss improvements to Algorithm 1. One crucial improvement is that most arithmetic operations

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

TABLE 1
Values of F of Type 2 GNB for $GF(2^5)$

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	-

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. The case of t odd is considered in Appendix A.

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., $F(n) = F(p-n)$, $1 \le n \le p-1$.*

Proof. In (5), 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 $0 \le i \le m-1$ and $0 \le j \le t-1$, 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,

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

□

From (5) and Lemma 1, one has $F(1) = F(p-1) = 0$. For $1 \le n \le p-2$, let us define

$$\Delta F(n) = F(n+1) - F(n) \mod m. \tag{6}$$

Now, we have the following corollary.

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

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

Proof. Using (6), one obtains $F(n+1) = \sum_{i=1}^{n} \Delta F(i)$. Applying Lemma 1 into (6), one can also write

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

which results in

$$\Delta F(p-n) = m - \Delta F(n-1),\ \ 2 \le n < \frac{p-1}{2},$$

$$\text{and } \Delta F\left(\frac{p-1}{2}\right) = 0.$$

□

Using Lemma 1 and (6), one can write (4) as follows:

$$c_i = \sum_{n=1}^{p-2} a_{F(n+1)+i} b_{F(n)+i},\ 0 \le i \le m-1, \tag{7}$$

$$= \sum_{n=1}^{p-2} a_{F(n)+\Delta F(n)+i} b_{F(n)+i},\ 0 \le i \le m-1. \tag{8}$$

For a particular GNB, the values of $\Delta F(n)$, $1 \le n \le p-2$, 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.

TABLE 2
Contents of Variables in Algorithm 2 for Multiplication of $A = (01110)$ and $B = (10101)$

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

We now state the vector-level algorithm for t even as follows:

Algorithm 2. (Vector-Level NB Multiplication for t even)

```
Input: $A,\ B \in GF(2^m)$, $\Delta F(n) \in [0, m-1]$, $1 \le n \le p-1$
Output: $C = AB$
1.  Initialize $S_A := A,\ S_B := B,\ C := 0$
2.  For $n = 1$ to $p-2$ {
3.     $S_A \lll \Delta F(n)$
4.     $R := S_A \odot S_B$
5.     $C := C + R$
6.     $S_B \lll \Delta F(n)$
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., $X \odot Y = (x_0y_0,\ x_1y_1,\ \cdots,\ x_{m-1}y_{m-1})$. The following example can be used to illustrate the operation of this algorithm.

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

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

Also, using (6), one can obtain the values of $\Delta F(n)$. 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. Here (and in the rest of the paper), 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 $(p-2)\lceil\frac{m}{\omega}\rceil = (tm-1)\lceil\frac{m}{\omega}\rceil$. Also, this algorithm needs $2(p-2)\lceil\frac{m}{\omega}\rceil = 2(tm-1)\lceil\frac{m}{\omega}\rceil$ cyclic shifts. We assume that an i-fold, $1 \le i < \omega$, 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.

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

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

For type II optimal normal bases, where $t=2$, the following remark can be made. However, for multiplication using type I optimal normal bases, one should use Algorithm 6 in Appendix A.

Remark 1. *For type II optimal normal bases, the dynamic instruction counts for Algorithm 2 is* $2(1+\rho)(2m-1)\lceil\frac{m}{\omega}\rceil$.

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

Although the previous algorithm utilizes the full data-path of the processor for NB multiplication and is very suitable and efficient for software implementation, below we present another algorithm which is even more efficient and requires fewer instructions and memory accesses. Also, the cost of this efficient 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 $v = \lceil\frac{m-1}{2}\rceil$. Then, one has the following result from [32].

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

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

where a_is and b_is are the NB coordinates of A and B, respectively. Also, indices and exponents are reduced $\bmod m$ *and*

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

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

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

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

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

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 ith coordinate is equal to its $(\frac{m}{2} + i \mod m)$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 1. *Let A and B be two elements of $GF(2^m)$ and C be their product. Then,*

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

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 $w_{j,k}$, $1 \le j \le v$, $1 \le k \le h_j$. Now, define

$$\Delta w_{j,k} \overset{\triangle}{=} w_{j,k} - w_{j,k-1},\ 1 \le j \le v,\ 1 \le k \le h_j,\ w_{j,0} = 0, \tag{15}$$

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. An efficient scheme to compute $\Delta w_{j,k}$s can be found in [29].

Algorithm 3. (ENB Multiplication for m Odd)

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

TABLE 3
Contents of Variables in Algorithm 3 for Multiplication of $A = (01110)$ and $B = (10101)$

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

```
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.    $S_A \ll 1$, $S_B \ll 1$
5.    $T_A := A \odot S_B$, $T_B := B \odot S_A$
6.    $R := T_A + T_B$
7.    For $k = 1$ to $h_j$ {
8.      $R \gg \Delta w_{j,k}$
9.      $C := C + R$
10.   }
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 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 [24], 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 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 can be seen in Algorithm 3, all $\Delta w_{j,k}$s have to be precomputed and it is done only once when the basis is chosen. In the above example, they are determined by calculating δ_js, which is essentially a multiplication process all by itself. For this multiplication, one can use either Algorithm 1 or Algorithm 2.

4.2 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 precomputed values of $\Delta w_{j,k}$. For software implementation of the above algorithm, one would heavily rely 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 $\sum_{j=1}^{v} h_j = \frac{C_N-1}{2}$ [18], the total number of XOR instructions is $\frac{1}{2}(C_N + m - 1)\lceil\frac{m}{\omega}\rceil$. Because of the $\odot$ operations in lines 1 and 5, one can also see that the above algorithm requires $m\lceil\frac{m}{\omega}\rceil$ AND instructions. We assume that each i-fold cyclic shift, $1 \le i \le m-1$, in lines 2, 4, and 8 needs $\rho\lceil\frac{m}{\omega}\rceil$ 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 $\frac{\rho}{2}(C_N + 2m - 1)\lceil\frac{m}{\omega}\rceil$. Based on the above discussion, we have the following.

Proposition 2. *The dynamic instruction count for Algorithm 3 is given by*

$$\#Instructions \approx \left(\frac{1+\rho}{2}C_N + \frac{3+2\rho}{2}m - \frac{2+\rho}{2}\right)\left\lceil\frac{m}{\omega}\right\rceil. \quad (16)$$

It is noted that the type I ONB only exists when m is even. For this type of bases, an algorithm and its instruction count are presented in Appendix B. For type II ONBs, where the value of m is odd, one can obtain the total instruction for Algorithm 3 by substituting $C_N = 2m - 1$ into (16). However, for the even values of m, where type II ONBs exist, one can use an algorithm presented in Appendix B to further reduce the instruction count. Now, we can state the following.

Remark 2. *The dynamic instruction count for ENB multiplication algorithm when the finite field is defined for type II optimal normal bases is at most* $((2.5 + 2\rho)m - (1.5 + \rho))\lceil\frac{m}{\omega}\rceil$.

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 precomputed $\Delta w_{j,k}$s.

Table 4 compares the number of dynamic instructions of the three algorithms we have described so far. As 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, this

TABLE 4
Generic Comparison of Multiplication Algorithms in Terms of Number of Instructions and Memory Requirements

Algorithms	# Instructions			Memory	
	XOR	AND	Others	Size in bits	# Accesses
Alg.1	$\frac{1}{4}m(tm-1)$	$m(tm-1)$	$2\rho m\lceil\frac{m}{\omega}\rceil$	$(tm-1)\lceil\log_2 m\rceil$	$2m(tm-1)$
Alg. 2	$(tm-1)\lceil\frac{m}{\omega}\rceil$	$(tm-1)\lceil\frac{m}{\omega}\rceil$	$2\rho(tm-1)\lceil\frac{m}{\omega}\rceil$	$\frac{tm}{2}\lceil\log_2 m\rceil$	tm
Alg. 3	$\frac{1}{2}(C_N+m-2)\lceil\frac{m}{\omega}\rceil$	$m\lceil\frac{m}{\omega}\rceil$	$\frac{\rho}{2}(C_N+2m-1)\lceil\frac{m}{\omega}\rceil$	$\frac{C_N-1}{2}\lceil\log_2 m\rceil$	$\frac{C_N-1}{2}$
Ratio of Alg. 2 to Alg. 3	$\approx\frac{2t}{t+1}$	$\approx t$	$\approx\frac{4t}{t+1}$	≈ 1	≈ 2

table gives memory sizes and numbers of memory accesses of these algorithms. The final row of this table gives approximate improvement factors of Algorithm 3 to Algorithm 2. A more detailed comparison of these two algorithms is given in Table 5 and for the five binary fields recommended by NIST for ECDSA (elliptic curve digital signature algorithm) [26].

These algorithms have been coded in software using the C programming language. Table 5 also shows timing (in μs) for these codes executed on Pentium III 533 MHz PC. The PC has 64 M bytes of RAM, 32 K bytes of L1 cache, and 512 K bytes of L2 cache. 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 $(C_N-1)\lceil\frac{m}{\omega}\rceil$ XOR instructions, $m\lceil\frac{m}{\omega}\rceil$ AND instructions, and $\rho(C_N+m-1)\lceil\frac{m}{\omega}\rceil$ other instructions. Thus, the dynamic instruction count would be $(\rho+1)(C_N+m-1)\lceil\frac{m}{\omega}\rceil$, which is about twice that in Algorithm 3 (see Proposition 2). In [32], one can find software implementation of the NB multiplication for two special cases, namely, two optimal normal bases. The method used in [32] is similar to that of the NB multiplication of [1].

Very recently, another NB multiplication algorithm suitable for software implementation has been proposed [24]. This algorithm is quite different from the ones discussed here. It uses much more memory, but, with respect to computational time, it is expected to be better than the algorithm presented here and it is possible to combine the work of [24] with that of this article to further improve the performance of NB multiplication in software.

Some of the recently proposed *polynomial basis* multiplication algorithms, for example, [9], [17], create a look-up table on the fly based on one of the inputs (say A) and yield significant speed-ups by processing a group of bits of the other input (i.e., B) at a time. At this point, it is not clear whether such a group-level processing of B can be incorporated into our Algorithm 3. However, if m is a composite number, then one can essentially achieve a similar kind of group-level processing by performing computations in the subfields. 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 $\mathrm{GF}(2^m)$, 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 [26], [27]. When these fields are to be used for elliptic curve crypto-systems, one must choose m such that its cofactors

TABLE 5
Comparison of the Proposed Algorithms for Binary Fields Recommended by NIST for ECDSA Applications ($\omega = 32$, $\rho = 4$)

Parameters				Algorithm 2, Algorithm 3							
				# Instructions				Memory		Actual timing	
m	t	u	C_N	XOR	AND	Others/ρ	Total	Size	# Accesses	in μs	Ratio
163	4	149	645	3906, 2418	3906, 978	7812, 2910	39060, 15036	2608, 2576	652, 322	307, 99	3.1
233	2	466	465	3720, 2784	3720, 1864	7440, 3720	37200, 19528	1864, 1856	466, 232	346, 126	2.75
283	6	398	1677	15273, 8811	15273, 2547	30546, 10089	152730, 51714	7641, 7542	1698, 838	1005, 318	3.16
409	4	316	1629	21255, 13234	21255, 5317	42510, 15899	212550, 82147	7362, 7326	1636, 814	1466, 473	3.1
571	10	3421	5637	102762, 55854	102762, 10278	205524, 61002	1027620, 310140	28550, 28180	5710, 2818	8423, 2949	2.86

are large enough to resist the attack described by Galbraith and Smart [6], [34]. The composite fields which are not vulnerable against the known attacks for elliptic curve crypto-systems are presented in [20], [5].

5.1 Algorithm

Let us introduce the following lemma from [21] which is used for constructing a composite field normal basis.

Lemma 3 [21]. *Let* $\gcd(m_1, m_2) = 1$. *Let* $N_1 = \{\beta_1^{2^j} \mid 0 \leq j \leq m_1 - 1\}$ *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 (i.e., both m_1 and m_2 are prime). This is particularly important for 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 hundred only (571 as given in [25]). To avoid the attack of [6], one, however, may like to choose m such that it has no small factors such as 2, 3, 5, 7, 11 (see [20] for secure composite values of $m \in [160, 600]$). This basically makes one choose m as the product of two primes. Thus, in the following, we give all equations and algorithms for odd degrees. 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 subfields $GF(2^{m_1})$ 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 [28]:

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

where

$$y_{i,j} = (a_i + a_{i+j})(b_i + b_{i+j}),\ 1 \leq j \leq v_1,\ 0 \leq i \leq m_1 - 1,$$
$$v_1 = \frac{m_1 - 1}{2},\quad \beta_1^{2^j+1} = \sum_{k=1}^{h_j^{(1)}} \beta_1^{2^{w_{j,k}^{(1)}}}.$$

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

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

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 subfield coordinates of A *and* B.

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

Algorithm 4. (ECFNB Multiplication of $GF(2^{m_1 m_2})$ over $GF(2^{m_2})$

```
Input: $A,\ B \in GF(2^m)$, $\Delta w_{j,k}^{(1)} \in [0, m_1 - 1]$, $1 \leq j \leq v_1$,
$v_1 = \frac{m_1-1}{2}$, $1 \leq k \leq h_j^{(1)}$
Output: $C = AB$
1.  Initialize $C := A \otimes B$, $S_A := A$, $S_B := B$
2.  For $j = 1$ to $v_1$ {
3.      $S_A \ll m_2$, $S_B \ll m_2$
4.      $T_A := A + S_A$, $T_B := B + S_B$
5.      $R := T_A \otimes T_B$
6.      For $k = 1$ to $h_j^{(1)}$ {
7.          $R \gg m_2 \Delta w_{j,k}^{(1)}$
8.      $C := C + R$
9.      }
10. }
```

In lines 1 and 5 of Algorithm 4,

$$A \otimes B = (A_0 B_0,\ A_1 B_1, \cdots,\ A_{m_1-1} B_{m_1-1})$$

denotes parallel subfield multiplications of A and B. This subfield multiplication can be implemented with an extension of Algorithm 3 such that it produces m_1 subfield multiplications over $GF(2^{m_2})$. This is shown in Algorithm 5, where $A \triangleright i$ (resp. $A \triangleleft i$) $0 \leq i \leq m_2 - 1$, denotes an i-fold right (resp. left) subfield cyclic shift of all subfield elements of A, i.e., $A_0,\ A_1, \cdots,\ A_{m_1-1}$, respectively.

Algorithm 5. (Parallel Subfield Multiplication over $GF(2^{m_2})$

```
Input: $A,\ B \in GF(2^m)$, $\Delta w_{j,k}^{(2)} \in [0, m_2 - 1]$, $1 \leq j \leq v_2$,
$1 \leq k \leq h_j^{(2)}$, $v_2 = \frac{m_2-1}{2}$
Output: $C = A \otimes B$
1.  Initialize $C := A \odot B$, $S_A := A$, $S_B := B$
2.  $C \triangleright 1$
3.  For $j = 1$ to $v_2$ {
4.      $S_A \triangleleft 1$, $S_B \triangleleft 1$
5.      $T_A := A \odot S_B$, $T_B := B \odot S_A$
6.      $R := T_A + T_B$
7.      For $k = 1$ to $h_j^{(2)}$ {
8.          $R \triangleright \Delta w_{j,k}^{(2)}$
9.          $C := C + R$
10.     }
11. }
```

5.2 Cost

In order to obtain the cost of Algorithm 4, we need to evaluate the cost of Algorithm 5, 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 5 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 4 also depends on how subfield elements, each of m_2 bits, are

TABLE 6
Cost of Algorithm 4

# 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	$\frac{\rho m_1}{4}(m_1+1)(C_2+2m_2-1)$
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 7
Cost of Algorithm 4 for Certain Composite Fields ($\rho = 4$)

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 8
Comparison of Algorithm 4 of This Paper with Algorithm 6 of [3] for $m_1 = 2^e$ and $m_2 = n, \gcd(2, n) = 1$

	Algorithm 6 of [3]	Algorithm 4 of this article
# of $GF(2^n)$ multiplications	4^e	$\frac{4^e+2^e}{2}$
# of $GF(2^n)$ additions	$2\cdot 4^e - 2\cdot 2^e$	$4^e + 2^{e-1}(C_1-3)$

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 cryptosystems 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 4 are almost free of cost (or at best register renaming). Based on this assumption, we give the dynamic instruction counts of Algorithm 4 in Table 6. In this table, ρ is the number of instructions needed for one subfield cyclic shift in each register and it is four in the C programming language.

Table 7 shows the number of instructions and memory requirements of Algorithm 4 for six different composite fields. These six fields are obtained by combining three m_1s and two m_2s. Based on Table 1 of [20], these fields are not vulnerable against the known attacks. Algorithm 4 is also coded for these composite fields using the C programming language. The actual timing (in μs) of Algorithm 4 executed on Pentium III 533 MHz PC is also shown in Table 7. In order to obtain parameters of the finite fields $GF(2^{m_1})$ and $GF(2^{m_2})$ used in Algorithm 4, we have used Table 10 and Table 11 of [29]. The results are presented in Table 9 of Appendix C.

5.3 Comparison and Comments

5.3.1 Normal Basis Multipliers

In order to compare the cost of Algorithm 4 with that of Algorithm 6 of [3] which uses $m = n \cdot 2^e$, one can let $m_1 = 2^e$ and $m_2 = n,\ \gcd(2,\ n) = 1$. The condition that $\gcd(2,\ n) = 1$ enables us to use Algorithm 4; however, it is not needed for Algorithm 6 of [3]. In [3], the complexity of $GF(2^m)$ multiplication is given in terms of the number of subfield, i.e., $GF(2^n)$, operations. Table 8 compares the number of subfield operations required in Algorithm 4 and that in [3]. In this table, our composite field multiplication algorithm requires $\frac{m_1(m_1+1)}{2} = \frac{4^e+2^e}{2}$ and $\frac{m_1}{2}(C_1+2m_1-3) = 4^e + 2^{e-1}(C_1-3)$ subfield multiplications and subfield additions, respectively. Since the subfield multiplication is more costly than the subfield addition, our proposed algorithm has about half of the computational complexity of the algorithm presented in [3].

5.3.2 Multipliers Based on Polynomial and Dual Bases

In addition to normal bases, in the literature at least two other types of bases, namely, polynomial and dual bases, have received considerable attention for implementation of multipliers over $GF(2^m)$ [33], [16], [8], [15]. Multiplication algorithms that use these two types of bases usually have regular structures and can easily take advantage of fixed irreducible polynomials, which define the fields. In addition, they can also be made to operate in digit serial fashion to provide improved performance (perhaps at the cost of some memory [9]). As a result, dual and/or polynomial basis multipliers can be faster than a normal basis multiplier (e.g., Algorithm 4). For the purpose of illustration, below we mention three pertinent multipliers where the values of m are composite.

Schroeppel et al. in [33] obtained $112.6\mu s$ and $7.1\mu s$ for implementing a multiplication over $GF(2^{155})$ on Sun SPARC IPC at 25 MHz and DEC Alpha 3000 at 175 MHz, respectively. They used polynomial basis representation for their operation. Guajardo and Paar in [8] have used the Karatsuba-Ofman algorithm for field multiplication over $GF((2^{16})^{11})$. Their 176×176 bit multiplication uses polynomial basis representation and requires $38.56\mu s$ on the DEC Alpha 3000 platform at 175 MHz. Also, Lee and Lim have presented a $GF(2^m)$ multiplication algorithm in [16] using a special type of dual basis called circular dual basis. The circular dual basis of $GF(2^m)$ exists only for the values of m where a type I optimal normal basis exists. Their multiplication method has been implemented on Pentium 133 MHz and requires $50\mu s$ for the multiplication in $GF(2^{178})$.

In each of the three cases above, the field dimension has a small prime factor (i.e., 5 for [33], 2 for both [8] and [16]). The algorithm of [33] does not make use of the small factor 5, but yields a high speed multiplier by carefully choosing the irreducible polynomial (namely, $x^{155} + x^{62} + 1$) to define the field. In our implementation, we have restricted the smallest factor to be 13 or more. A smaller factor is expected to improve the timing results reported in Table 7 (perhaps at the expense of reduced level of security for certain fields). Also, for $m = 178$, one can use type 1 optimal normal basis to speed up the multiplication for which an algorithm is included in Appendix B.

As can be seen from the above discussions, various authors have reported the timing results of their multipliers using different computing platforms and parameters. A fair comparison solely based on timing results is therefore difficult. Nevertheless, software implementation of NB multiplication is important. Such an implementation is crucial in the point halving and add algorithm for elliptic curve point multiplication [13]. It is also very useful for the Frobenius endomorphism used with Koblitz curve-based crypto-systems. Thus, the work presented in this article is a step forward in this direction.

6 Conclusions

In this paper, 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 [6]. Both Algorithms 2 and 3 have been coded for software implementation using C and our timing results show that Algorithm 3 is about 200 percent faster than Algorithm 2. These results are for those five Gaussian normal bases over the binary fields which NIST has described in their ECDSA document [25]. For the purpose of using NIST parameters, although we have presented our results for Gaussian normal bases, our algorithms are quite 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 [6] 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 4) for normal basis multiplication for $GF(2^m)$ over $GF(2^{m_2})$. Our results show that, for similar values of m, Algorithm 4 can be much more efficient than Algorithm 3. For example, the actual timing of Algorithm 3 is 318 microseconds for $GF(2^{283})$, whereas the timing of Algorithm 4 is 114 microseconds for $GF(2^{299})$. Composite fields also provide an added flexibility to hardware-software codesign of finite field processors. For example, Algorithm 5, which is *called* by Algorithm 4 a total of $\frac{m_1+1}{2}$ times, can be implemented in hardware for small values of m_2 and Algorithm 4 can be embedded in a microcontroller, which would give us a high speed, yet quite flexible, normal basis multiplier over very large fields.

Appendix A
Vector-Level NBV 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, requires the input sequence $F(1),\ F(2),\ \cdots,\ F(p-1)$ to be precomputed using (5), i.e.,

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

where $p = tm + 1$ and u is an integer of order $t \bmod p$. Since p should be prime, tm has to be an even integer. Also, t is odd, which implies that m should be even. The ith coordinate of the product $C = AB$ can be obtained as [26]:

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

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 (6), i.e., $\Delta F(n) = F(n+1) - F(n) \bmod m$, where $F(1) = 0$. Let us introduce

TABLE 9
(a) Parameters of Type t GNBs Used in Algorithm 4
(b) Parameters of Type 2 GNBs Used in Algorithm 5 ($t = 2,\ h_j^{(2)} = 2$)

$m_1(u,t)$	j	$\Delta w_{j,k}^{(1)}$	$h_j^{(1)}$
13(23,4)	1	0,3,1,1	4
	2	4,4,3,1	4
	3	1,6,1,4	4
	4	1,1,8,2	4
	5	1,6,1,2	4
	6	9,2	2
17(47,6)	1	0,3,6,2	4
	2	6,1,1,1,3,4	6
	3	1,5,3,2,2,1	6
	4	7,8	2
	5	7,3,2,3	4
	6	2,1,4,2,5,1	6
	7	2,2,1,1,6,3	6
	8	2,7,1,1,3,2	6
19(7,10)	1	0,2,2,2,1,4,1,1,2,3	10
	2	1,8,5,1	4
	3	11,2,2,3	4
	4	1,4,1,1,1,2,3,2	8
	5	4,3,3,1,3,4	6
	6	1,3,3,2,1,4,1,3	8
	7	1,3,1,1,2,9	6
	8	4,3,2,2,2,3	6
	9	2,4,2,5,1,1	6

(a)

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

(b)

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

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 6. (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: $C = AB$
1.  Initialize $S_A := A,\ S_B := B,\ C := 0,\ f := 0$
2.    $S_B \ll \frac{m}{2}$
3.    $R := S_A \odot S_B,\ R = (r_0,\ r_1, \cdots,\ r_{m-1})$
4.  For $i = 0$ to $m-1$ {
5.    $f := f + r_i$
6.  }
7.  $S_B := B$
8.  For $n = 1$ to $p-2$ {
9.    $S_A \ll \Delta F(n)$
10.     $S_B \ll \Delta F'(n)$
11.     $R := S_A \odot S_B$
12.     $C := C + R$
13. }
14. If $f$ is 1, $C := C + (1, 1, \cdots, 1, 1)$
```

Proposition 3. *The average dynamic instruction count for Algorithm 6 is given by*

$$\#Instructions \approx$$
$$(1+\rho)(2tm-1)\left\lceil\frac{m}{\omega}\right\rceil + \left\lceil\log_2\left\lceil\frac{m}{\omega}\right\rceil\right\rceil + 3\lceil\log_2\omega\rceil + \frac{\lceil\frac{m}{\omega}\rceil}{2}.$$

Remark 3. *For type I optimal normal bases, the average dynamic instruction counts for Algorithm 6 is*

$$(1+\rho)(2m-1)\left\lceil\frac{m}{\omega}\right\rceil + \left\lceil\log_2\left\lceil\frac{m}{\omega}\right\rceil\right\rceil + 3\lceil\log_2\omega\rceil + \frac{\lceil\frac{m}{\omega}\rceil}{2}.$$

APPENDIX B
EFFICIENT NB MULTIPLICATION ALGORITHM FOR m EVEN

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

```
12. $S_A \ll 1$, $S_B \ll 1$
13. $T_A := A \odot S_B$, $T_B := B \odot S_A$
14. $R := T_A + T_B$
15. For $k = 1$ to $\frac{h_v}{2}$ {
16.   $R \gg \Delta w_{v,k}$
17.   $C := C + R$
18. }
```

One of the special cases of Algorithm 7 is multiplication using a type I optimal normal basis where m is always even. For such an NB, there exists $\delta_j = \beta^{2^{w_j}}$, $1 \le j \le v-1$, $v = \frac{m}{2}$, and $\delta_v = 1$ [32], i.e., $\Delta w_{j,1} = w_j$, $1 \le j \le v-1$. This simplifies Algorithm 7 as follows:

Algorithm 7a. (ENB multiplication for type I ONB)

```
Input: $A,\ B \in GF(2^m), w_j,\ 1 \le j \le v-1$,
Output: $C = AB$
1. Initialize $C := A \odot B$, $S_A := A$, $S_B := B$, $f := 0$
2. $C \gg 1$
3. For $j = 1$ to $v-1$ {
4.   $S_A \ll 1$, $S_B \ll 1$
5.   $T_A := A \odot S_B$, $T_B := B \odot S_A$
6.   $R := T_A + T_B$
7.   $R \gg w_j$
8.   $C := C + R$
9.   }
10. }
11.   $S_A \ll 1$
12.   $R := S_A \odot B$, $R = (r_0,\ r_1, \cdots,\ r_{m-1})$
13. For $i = 0$ to $m-1$ {
14.   $f := f + r_i$
15. }
16. If $f$ is 1, $C := C + (1, 1, \cdots, 1, 1)$
```

Proposition 4. *The average dynamic instruction count for Algorithm 7a is given by*

$$\#Instructions \approx \left(\frac{m}{2}(3\rho+4) - (\rho+1.5)\right)\left\lceil\frac{m}{\omega}\right\rceil + \left\lceil\log_2\left\lceil\frac{m}{\omega}\right\rceil\right\rceil + 3\lceil\log_2\omega\rceil.$$

Appendix C
Parameters Used in Algorithms 4 and 5

Parameters used in Algorithms 4 and 5 are shown in Table 9.

Acknowledgments

The authors would like to thank Z. Zhang for his help with implementing the algorithms and getting their timing results. The authors would also like to thank the reviewers for their comments. A preliminary version of this article appeared in [31].

References

[1] G.B. Agnew, R.C. Mullin, I.M. Onyszchuk, and S.A. Vanstone, "An Implementation for a Fast Public-Key Cryptosystem," *J. Cryptology,* vol. 3, pp. 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 Comm.,* vol. 11, no. 5, pp. 804-813, June 1993.

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

[4] D.W. Ash, I.F. Blake, and S.A. Vanstone, "Low Complexity Normal Bases," *Discrete Applied Math.,* vol. 25, pp. 191-210, 1989.

[5] M. Ciet and J.-J. Quisquater, F. Sica, "A Secure Family of Composite Finite Fields Suitable for Fast Implementation of Elliptic Curve Cryptography," *Proc. Indocrypt 2001,* pp. 108-116, Dec. 2001.

[6] S.D. Galbraith and N. Smart, "A Cryptographic Application of Weil Descent," *Proc. Seventh IMA Conf. Cryptography and Coding,* pp. 191-200, 1999.

[7] S. Gao and H.W. Lenstra Jr., "Optimal Normal Bases," *Designs, Codes and Cryptography,* vol. 2, pp. 315-323, 1992.

[8] J. Guajardo and C. Paar, *Efficient Algorithms for Elliptic Curve Cryptosystems,* pp. 342-356. Springer-Verlag, 1997.

[9] M.A. Hasan, "Look-Up Table-Based Large Finite Field Multiplication in Memory Constrained Cryptosystems," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 749-758, July 2000.

[10] M.A. Hasan, M.Z. Wang, and V.K. Bhargava, "A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields," *IEEE Trans. Computers,* vol. 42, no. 10, pp. 1278-1280, Oct. 1993.

[11] IEEE Std 1363-2000, "IEEE Standard Specifications for Public-Key Cryptography," Jan. 2000.

[12] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *Int'l J. Information Security,* vol. 1, pp. 36-63, 2001.

[13] E. Knudsen, "Elliptic Scalar Multiplication Using Point Halving," *Proc. ASIACRYPT 1999,* pp. 135-149, 1999.

[14] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Computation,* vol. 48, pp. 203-209, 1987.

[15] C.K. Koc and T. Acar, "Montgomery Multiplication in GF(2^k)," *Designs, Codes, and Cryptography,* vol. 14, pp. 57-69, 1998.

[16] C. Lee and J. Lim, "A New Aspect of Dual Basis for Efficient Field Arithmetic," *Proc. Int'l Workshop Practice and Theory in Public Key Cryptography (PKC '99),* pp. 12-28, 1999.

[17] J. Lopez and R. Dahab, "High Speed Software Multiplication in F_{2^m}," *Proc. Indocrypt 2000,* pp. 203-212, 2000.

[18] C.-C. Lu, "A Search of Minimal Key Functions for Normal Basis Multipliers," *IEEE Trans. Computers,* vol. 46, no. 5, pp. 588-592, May 1997.

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

[20] M. Maurer, A. Menezes, and E. Teske, "Analysis of the GHS Weil Descent Attack on the ECDLP over Characteristic Two Finite Fields of Composite Degree," *Proc. Indocrypt 2001,* pp. 195-213, Dec. 2001.

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

[22] V.S. Miller, "Use of Elliptic Curves in Cryptography," *Proc. Crypto '85,* pp. 417-426, 1986.

[23] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson, "Optimal Normal Bases in $GF(p^n)$," *Discrete Applied Math.,* vol. 22, pp. 149-161, 1988/1989.

[24] P. Ning and Y.L. Yin, "Efficient Software Implementation for Finite Field Multiplication in Normal Basis," *Proc. Information and Commu. Security (ICICS 2001),* pp. 177-181, Nov. 2001.

[25] Nat'l Inst. of Standards and Technology, *Digital Signature Standard,* FIPS Publication 186-2, 2000.

[26] S. Oh, C.H. Kim, J. Lim, and D.H. Cheon, "Efficient Normal Basis Multipliers in Composite Fields," *IEEE Trans. Computers,* vol. 49, no. 10, pp. 1133-1138, Oct. 2000.

[27] C. Paar, P. Fleishmann, and P. Soria-Rodriguez, "Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents," *IEEE Trans. Computers,* vol. 48, no. 10, pp. 1025-1034, Oct. 1999.

[28] A. Reyhani-Masoleh and M.A. Hasan, "On Efficient Normal Basis Multiplication," *Proc. Indocrypt 2000,* pp. 213-224, Dec. 2000.

[29] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors," Technical Report CORR 2001-25, Dept. of C & O, Univ. of Waterloo, Canada, Apr. 2001.

[30] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors," *Proc. Selected Areas in Cryptography (SAC 2001),* pp. 230-244, Aug. 2001.

[31] A. Reyhani-Masoleh and M.A. Hasan, "A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$," *IEEE Trans. Computers,* vol. 51, no. 5, pp. 511-520, May 2002.

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

[33] R. Schroeppel, H. Orman, S.W. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," *Proc. CRYPTO '95,* pp. 43-56, 1995.

[34] N.P. Smart, "How Secure Are Elliptic Curves over Composite Extension Fields?" *Proc. Eurocrypt 2001,* pp. 30-39, 2001.

[35] B. Sunar and C.K. Koc, "An Efficient Optimal Normal Basis Type II Multiplier," *IEEE Trans. Computers,* vol. 50, no. 1, pp. 83-88, Jan. 2001.

Arash Reyhani-Masoleh received the BSc degree from Iran University of Science and Technology in 1989, the MSc degree from the University of Tehran in 1991, both with the first rank in electrical and electronic engineering, and the PhD degree in electrical and computer engineering from the University of Waterloo, Canada, in 2001. From 1991 to 1997, he was with the Department of Electrical Engineering, Iran University of Science and Technology. Since June 2001, he has been a postdoctoral fellow with the Centre for Applied Cryptographic Research, University of Waterloo. His current research interests include algorithms and VLSI architectures for computations in finite fields, fault-tolerant computing, and error-control coding. He was awarded an NSERC (Natural Sciences and Engineering Research Council of Canada) postdoctoral fellowship in 2002. He is a member of the IEEE and the IEEE Computer Society.

M. Anwar Hasan received the BSc degree in electrical and electronic engineering, the MSc degree in computer engineering, both from the Bangladesh University of Engineering and Technology, in 1986 and 1988, respectively, and the PhD degree in electrical engineering from the University of Victoria in 1992. Since 1993, he has been with the Department of Electrical and Computer Engineering, University of Waterloo, Canada, where he is now a professor. At the University of Waterloo, he is also a member of the Centre for Applied Cryptographic Research and the Center for Wireless Communications. His current research interests include algorithms and architectures for computations in Galois fields, data security and reliability, and digital communication networks. From January to December of 1999, he was on sabbatical with Motorola Labs., Schaumburg, Illinois. He is a recipient of the Raihan Memorial Gold Medal. At the University of Victoria, he was awarded the President's Research Scholarship four times. He has served on the program and executive committees of several conferences and, currently, he is an associate editor of the *IEEE Transactions of Computers*. He is a senior member of the IEEE, a member of the IEEE Computer Society, and a licensed professional engineer of Ontario.