Two Software Normal Basis Multiplication Algorithms for $GF(2^n)$

Haining Fan and Yiqi Dai

Abstract - In this paper, two different normal basis multiplication algorithms for software implementation are proposed over $GF(2^n)$. The first algorithm is suitable for high complexity normal bases and the second algorithm is fast for type-I optimal normal bases and low complexity normal bases.

Index Terms - Finite field, normal basis, multiplication algorithm.

1. Introduction

The arithmetic operations in $GF(2^n)$ play an important role in coding theory, computer algebra, and cryptosystems. Among the different types of field representations, normal basis (NB) has received considerable attention for efficient implementation. For portability as well as for price reasons, it is often advantageous to realize cryptographic algorithms in software. While many constructions of VLSI NB multipliers have been proposed recently, few software-efficient NB algorithms can be found in the open literature. In [1], a software algorithm for type-I optimal normal bases (ONB) was presented. It can be further improved if the symmetric property proposed in [2,3] is employed. In [3], Reyhani-Masoleh and Hasan proposed a word-level NB multiplication algorithm over $GF(2^n)$, and we denote it as the RH algorithm. The RH algorithm is designed for working with all NBs. A bit-level NB multiplication algorithm is presented in the IEEE standard P1363-2000 [4], but it is not efficient for software implementation. Ning and Yin presented a generalized version of this algorithm in [5], and we denote it as the NY algorithm. Although the NY algorithm is fast for ONBs, it is slow for nonoptimal normal bases.

Haining Fan and Yiqi Dai are with the Department of Computer Science, Tsinghua University, Beijing 100084, People's Republic of China.

E-mail: fhn99@mails.tsinghua.edu.cn

In this paper, two NB multiplication algorithms for software implementation are proposed. The first algorithm (Algorithm 1) is an improvement on the RH algorithm. The theoretical analysis shows that it is faster than the RH algorithm. But experimental results show that this is not true for a few $GF(2^n)$s, e.g. $GF(2^{359})$ and $GF(2^{491})$. The reason is that the size of lookup tables of Algorithm 1 is larger than that of the RH algorithm. The total number of cyclic shift operations (CSO) needed in the second algorithm (Algorithm 2) is a constant. For type-II ONBs, the NY algorithm is faster than Algorithm 2, but it is slower than Algorithm 2 for other NBs. Compared to Algorithm 2, Algorithm 1 is suitable for high complexity NBs. For example, our experimental results show that it is faster than Algorithm 2 in $GF(2^{283})$ (Type 6 Gaussian NB (GNB)) and $GF(2^{571})$ (Type 10 GNB).

We also compare these NB algorithms to the polynomial basis multiplication algorithm presented in [6], i.e., the finite field analogue of the Montgomery multiplication for integers. Our experimental results show that in some $GF(2^n)$s where ONBs or low complexity normal bases exist Algorithm 2 is faster than the Montgomery algorithm, for example, type 4 GNBs in $GF(2^{577})$, $GF(2^{673})$ and $GF(2^{739})$.

This paper is organized as follows: Section 2 introduces the RH algorithm and presents a method to reduce the number of cyclic shift operations. This method may be applied to all NB multiplication algorithms we discussed. Algorithm 1 and Algorithm 2 are presented in Section 3 and Section 4 respectively. Theoretic analyses and numerical results are presented in Section 5. Finally, concluding remarks are made in Section 6.

2. Preliminaries

Let γ be an element of $GF(2^n)$, for simplicity, denote γ^{2^i} by γ_i. Given a normal basis $N=\{\beta_0, \beta_1, \beta_2, ..., \beta_{n-1}\}$ of $GF(2^n)$ over $GF(2)$, a field element A can be represented by a binary vector $(a_0, a_1, ..., a_{n-1})$ with respect to this basis as $A = \sum_{i=0}^{n-1} a_i \beta_i$, where $a_i \in GF(2)$ and $i = 0,1,...,n-1$.

For $1 \le i \le n-1$, let $\beta_0 \beta_i = \sum_{j=0}^{n-1} \phi_{i,j} \beta_j$ be the expansion of $\beta_0 \beta_i$ with respect to N, where

$\phi_{i,j} \in GF(2)$. Let $S_i = \{j \mid \phi_{i,j} = 1\}$ and $h_i=|S_i|$. We may rewrite S_i as $S_i = \{w_{i,1}, w_{i,2}, ..., w_{i,h_i}\}$, where

$$0 \le w_{i,1} < w_{i,2} < ... < w_{i,h_i} \le n-1. \tag{1}$$

Clearly, $\beta_0\beta_i = \sum_{k=1}^{h_i} \beta_{w_{i,k}}$.

Note that for a particular normal basis N the representation of $\beta_0\beta_i$ is fixed and so is $w_{i,k}$.

In [3] a vector-level normal basis multiplication algorithm over $GF(2^n)$ is presented. The algorithm includes two similar versions: one is for n odd and the other is for n even. In this paper, we assume that n is odd, unless otherwise stated.

Let $v=(n-1)/2$ and $<x>$ denote the non-negative residue of x mod n. $D=AB$ can be computed by the following identity [3]:

$$D = AB = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j \beta_i \beta_j = \sum_{i=0}^{n-1} a_i b_i \beta_{\langle i+1\rangle} + \sum_{i=1}^{n-1}\sum_{j=0}^{n-1} (a_{\langle i+j\rangle} b_j)(\beta_i\beta_0)^{2^j}$$

$$= \sum_{i=0}^{n-1} a_i b_i \beta_{\langle i+1\rangle} + \sum_{i=1}^{v}\sum_{k=1}^{h_i}\left[\sum_{j=0}^{n-1}\left(a_{\langle i+j\rangle} b_j + b_{\langle i+j\rangle} a_j\right)\beta_{\langle j+w_{i,k}\rangle}\right]. \tag{2}$$

Let $A_i = A^{2^i}$. If we define $B \,\&\, A_{n-i}$ as $B \,\&\, A_{n-i} = (a_i b_0, a_{\langle i+1\rangle} b_1, ..., a_{\langle i+n-1\rangle} b_{n-1})$ and treat it as the field element, then $\sum_{j=0}^{n-1} a_{\langle i+j\rangle} b_j \beta_{\langle j+w_{i,k}\rangle} = (B \,\&\, A_{n-i})_{w_{i,k}}$. So (2) may be rewritten as follows:

$$D = (B \,\&\, A)_1 + \sum_{i=1}^{v}\sum_{k=1}^{h_i}\left[(B \,\&\, A_{n-i})_{w_{i,k}} + (A \,\&\, B_{n-i})_{w_{i,k}}\right] \tag{3}$$

$$= (B \,\&\, A)_1 + \sum_{i=1}^{v}\sum_{k\in S_i}\left[(B \,\&\, A_{n-i}) + (A \,\&\, B_{n-i})\right]_k. \tag{4}$$

Furthermore, Let $R_i=(B \,\&\, A_{n-i})+(A \,\&\, B_{n-i})$, we have

$$D=(B \,\&\, A)_1 + \sum_{i=1}^{v}\sum_{k\in S_i}(R_i)_k. \tag{5}$$

Similarly, for even n, we set $v= n/2$ and have:

$$D=(B \,\&\, A)_1 + \sum_{i=1}^{v}\sum_{k\in S_i}(R_i)_k, \tag{6}$$

where R_i is defined as:

$$\begin{cases} R_i = (B \,\&\, A_{n-i}) + (A \,\&\, B_{n-i}) & \text{where } 1 \le i \le v-1 \\ R_v = A \,\&\, B_v \end{cases}. \tag{7}$$

Based on (5), the following multiplication algorithm was presented in [3]:

RH Multiplication Algorithm for *n* odd:

INPUT: A, B, S_i, where $1 \le i \le v$.

OUTPUT: D=AB.

S1: D := (A & B) >> 1;

S2: U_A := A; U_B := B;

S3: for i = 1 to v do {

S4: U_A := U_A << 1 ; U_B := U_B << 1;

S5: R := (A & U_B) $\oplus$ (B & U_A);

S6: for each $k \in S_i$ do D := D $\oplus$ (R >> k); }

S7: Output D ;

Notes: 1. A & B= $(a_0b_0, a_1b_1, ..., a_{n-1}b_{n-1})$.

2. $\oplus$ denotes the addition in $GF(2^n)$.

3. A<<i (resp. A>>i) denotes i-fold left (resp. right) CSO of the coordinates of A.

Obviously, the number of CSO in S4 is n-1. Before introducing the first algorithm, we show that this number can be further reduced for large values of n.

Let z be the full width of data-path of the general-purpose processor, e.g. z=32 for Pentium CPU. We assume that z<v.

Let DA_0= (a_0,a_1,...,a_{n-1},a_0,a_1,...,a_{v-1}) and DA_j = (a_j,...,a_{n-1},a_0,a_1,...,$a_{v-1}a_0$,...,a_{j-1}), i.e., DA_j is the j-fold left cyclic shifts of the (n+v)-bit vector DA_0, where 0<j<z .

In software implementation, DA is defined as a 2-dimensional array $DA[z][\lceil \frac{n+v}{z} \rceil]$, and each DA_j is stored in $\lceil \frac{n+v}{z} \rceil$ successive computer words. So for $1 \le i \le v$, A_{n-i} is stored in $\lceil \frac{n}{z} \rceil$ successive computer words starting from $DA[s][t]$ and ending at $DA[s][t+\lceil \frac{n}{z} \rceil -1]$, where t=[i/z], s=i && (z-1) and && denotes integer bit-wise AND. That is to say, the two indices of A_{n-i} can be computed at the cost of one binary shift (t = [i/z]) and one bit-wise AND. Moreover, the starting address of A_{n-i} may be calculated in the precomputation procedure. Similarly, arrays DB_j's are defined.

Clearly, the time complexity to compute DA_j's and DB_j's ($0 \le j < z$) is about $3z$ n-bit CSOs, and $3zn$ bits are needed to store these arrays.

3. Algorithm 1

Now we present the first algorithm. It is an improvement on the RH algorithm. The idea is based on the following observation on (5). For some $1 \le i_1 < ... < i_e \le v$, $S_{i_1} \cap ... \cap S_{i_e}$ may not be empty. Thus for each $k \in S_{i_1} \cap ... \cap S_{i_e}$, $\sum_{i=i_1,...,i_e} \sum_{k \in S_i} (R_i)_k$ can be computed by

$$\sum_{i=i_1,...,i_e} \sum_{k \in S_i} (R_i)_k = \left(\sum_{i=i_1,...,i_e} R_i \right)_k . \tag{8}$$

The saving of the k-fold cyclic shift operation is obvious. The left side of (8) needs e such operations, while the right side needs only 1.

The correctness of this method is based on the fact that we can interchange the order of summation in the identity (5). That is to say, since $0 \le k \le n-1$, we have

$$D=(B \,\&\, A)_1 + \sum_{i=1}^{v} \sum_{k \in S_i} (R_i)_k = (B \,\&\, A)_1 + \sum_{k=0}^{n-1} \left(\sum_{i:\ \text{such that } 1 \le i \le v \text{ and } k \in S_i} R_i \right)_k . \tag{9}$$

Similarly, for even n, we set $v= n/2$ and have:

$$D=(B \,\&\, A)_1 + \sum_{i=1}^{v} \sum_{k \in S_i} (R_i)_k = (B \,\&\, A)_1 + \sum_{k=0}^{n-1} \left(\sum_{i:\ \text{such that } 1 \le i \le v \text{ and } k \in S_i} R_i \right)_k . \tag{10}$$

where R_i is defined in (7).

Based on (9) and the method to compute DA_i and DB_i for $0<i<z$, we now present Algorithm 1 for odd values of n. For each $0 \le k \le n-1$, the following precomputation procedure is to find all i's such that $1 \le i \le v$ and $k \in S_i$.

Precomputation:

INPUT: n, S_i, where $1 \le i \le v$.

OUTPUT: e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$.

S1: for $k = 0$ to n-1 do e_k :=0;

S2: for $i = 1$ to v do {

S3: for each $k \in S_i$ do {

S4: $m[k][e_k] := i$;

S5: $e_k := e_k+1$; } }

This procedure outputs e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$. e_k is the total number of i's such that $1 \le i \le v$ and $k \in S_i$, and $m[k][0]$ to $m[k][e_k\text{-}1]$ store these i's, i.e., $k \in S_{m[k][j]}$ for $0 \le j \le e_k - 1$.

Multiplication Algorithm 1 for *n* Odd:

INPUT: A, B, e_k and $m[k][j]$, where $0 \le k \le n-1$ and $0 \le j \le e_k - 1$.

OUTPUT: D=AB.

S1: Compute DA_i and DB_i for $0 \le i < z$;

S2: $D := A_1 \,\&\, B_1$;

S3: for $i = 1$ to v do $R[i] := (B \,\&\, A_{n-i}) \oplus (A \,\&\, B_{n-i})$;

S4: for $k = 0$ to n-1 do

S5: if $e_k > 0$ then{

S6: $C := R[m[k][0]]$;

S7: for $j = 1$ to e_k-1 do $C := C \oplus R[m[k][j]]$;

S8: $D := D \oplus (C >> k)$; }

S9: Output D ;

Notes: Algorithm 1 may be implemented without computing arrays DA and DB. In this implementation, each $R[i]$ is computed using statements similar to S4 and S5 of the RH algorithm [5], and the total number of CSO becomes at most $2n$. Experimental results indicate that arrays DA and DB speedup Algorithm 1 by no more than 10% for $GF(2^n)$s listed in Table 4 of Section 5. For example, Algorithm 1 without computing arrays DA and DB performs one multiplication operation in 1566 μs over $GF(2^{571})$. This result is still better than the following Algorithm 2.

4. Algorithm 2

From the definition of *A&B* we know that

$(A \& B)_i = (A \& B)^{2^i} = (a_{n-i}b_{n-i}, a_{\langle n-i+1\rangle}b_{\langle n-i+1\rangle}, \ldots, a_{n-1}b_{n-1}, a_0b_0, \ldots, a_{\langle n-i-1\rangle}b_{\langle n-i-1\rangle}) = (A_i \& B_i)$.

Thus (3) can be rewritten as:

$$D = (B_1 \& A_1) + \sum_{i=1}^{v}\sum_{k=1}^{h_i}\left[\left(B_{w_{i,k}} \& A_{\langle w_{i,k}-i\rangle}\right) + \left(A_{w_{i,k}} \& B_{\langle w_{i,k}-i\rangle}\right)\right]. \qquad (11)$$

Since $\sum_{i=1}^{v} h_i = (C_N - 1)/2$, D can be computed by (11) at the cost of about C_N AND (&) and C_N XOR ($\oplus$) operations if A_i's and B_i's are available, where $0<i<n$ and C_N denotes the complexity of the normal basis N.

Now we show that (11) can be further improved. From (1) we know that $0 \le w_{i,k} \le n-1$ and $w_{i,s} \ne w_{i,t}$ for a given i, where $1 \le i \le v$ and $1 \le s \ne t \le h_i$. Thus (11) can be rewritten as:

$$D = (B_1 \& A_1) + \sum_{w=0}^{n-1} \sum_{\substack{i:\text{ such that } 1\le i\le v \text{ and} \\ w=w_{i,k} \text{ for some } k,\text{ where } 1\le k\le h_i}} \left[\left(B_w \& A_{\langle w-i\rangle}\right) + \left(A_w \& B_{\langle w-i\rangle}\right)\right]$$

$$= (B_1 \& A_1) + \sum_{w=0}^{n-1}\left(\left(A_w \& \left(\sum_{\substack{i:\text{ such that } 1\le i\le v \text{ and} \\ w=w_{i,k} \text{ for some } k,\text{ where } 1\le k\le h_i}} B_{\langle w-i\rangle}\right)\right) + \left(B_w \& \left(\sum_{\substack{i:\text{ such that } 1\le i\le v \text{ and} \\ w=w_{i,k} \text{ for some } k,\text{ where } 1\le k\le h_i}} A_{\langle w-i\rangle}\right)\right)\right). \qquad (12)$$

Obviously, the number of & operations in (12) is $2n+1$. Thus (12) is faster than (11) for nonoptimal normal bases.

Similarly, for even n, we set $v = n/2$ and have:

$$D = (B_1 \& A_1) + \sum_{i=1}^{v-1}\sum_{k=1}^{h_i}\left[\left(A_{w_{i,k}} \& B_{\langle w_{i,k}-i\rangle}\right) + \left(B_{w_{i,k}} \& A_{\langle w_{i,k}-i\rangle}\right)\right] + \sum_{k=1}^{h_v}\left(A_{w_{v,k}} \& B_{\langle w_{v,k}-v\rangle}\right).$$

$$= (B_1 \& A_1) + \sum_{w=0}^{n-1}\left(\left(A_w \& \left(\sum_{\substack{i:\text{ such that } 1\le i\le v \text{ and} \\ w=w_{i,k} \text{ for some } k,\text{ where } 1\le k\le h_i}} B_{\langle w-i\rangle}\right)\right) \oplus \left(B_w \& \left(\sum_{\substack{i:\text{ such that } 1\le i\le v-1 \text{ and} \\ w=w_{i,k} \text{ for some } k,\text{ where } 1\le k\le h_i}} A_{\langle w-i\rangle}\right)\right)\right). \qquad (13)$$

Especially, for type-I ONBs, it is well known that $\beta_0\beta_v = 1 \in GF(2)$ and $\beta_0\beta_i = \beta_j$ for some $0 \le j \le n-1$, where $1 \le i \le v-1$. Thus the formula (13) can be further simplified if the Hamming weight method of [1] is used, i.e., we have

$$D=\left(B_1\ \&\ A_1\right)+\sum_{i=1}^{v-1}\left(\left(B_{w_{i,1}}\ \&\ A_{\langle w_{i,1}-i\rangle}\right)+\left(A_{w_{i,1}}\ \&\ B_{\langle w_{i,1}-i\rangle}\right)\right)+\text{HammingWeight}\left(B\ \&\ A_v\right). \qquad (14)$$

A_w=$A_{n-(n-w)}$ can be computed by the method introduced in Section 2. Here the array *DA* is defined as:

DA_0 = (a_0,a_1,...,a_{n-1},a_0,a_1,...,a_{n-1}) and

DA_j = (a_j,...,a_{n-1},a_0,a_1,...,$a_{n-1}$$a_0$,...,$a_{j-1}$), where $1\le j<z$,

i.e., they are 2*n*-bit vectors. The array *DB* is defined in the similar way.

So the time complexity to compute DA_j and DB_j is about 4*z* *n*-bit cyclic shift operations, and 4*zn* bits are needed to store these arrays, where $0\le j<z$.

Based on (12), we now present the second algorithm for odd values of *n*.

For each $0\le w\le n-1$, the following precomputation procedure is to find all *i*'s such that $1\le i\le v$ and w=$w_{i,k}$ for some *k*, where $1\le k\le h_i$.

Precomputation:

INPUT: *n*, S_i, where $1\le i\le v$.

OUTPUT: e_w and *m*[*w*][*j*], where $0\le w\le n-1$ and $0\le j\le e_w-1$.

S1: for *w* = 0 to *n*-1 do e_w:=0;

S2: for *i* = 1 to *v* do {

S3: for each $w\in S_i$ do {

S4: *m*[*w*][e_w] := *i* ;

S5: e_w := e_w+1; } }

This procedure outputs e_w and *m*[*w*][*j*], where $0\le w\le n-1$ and $0\le j\le e_w-1$. e_w is the total number of *i*'s such that $1\le i\le v$ and w=$w_{i,k}$, and *m*[*w*][0] to *m*[*w*][e_w-1] store these *i*'s, i.e., $w\in S_{m[w][j]}$ for $0\le j\le e_w-1$.

Multiplication Algorithm 2 for *n* Odd:

INPUT: *A*, *B*, e_w and *m*[*w*][*j*], where $0\le w\le n-1$ and $0\le j\le e_w-1$.

OUTPUT: *D*=*AB*.

S1: Compute DA_i and DB_i for $0 \le i < z$;

S2: $D := A_1$ & B_1;

S3: for $w = 0$ to n-1 do

S4: if $e_w > 0$ then{

S5: $UA := A_{<w-m[w][0]>}$; $UB := B_{<w-m[w][0]>}$;

S6: for $j = 1$ to e_w-1 do { $UA := UA \oplus A_{<w-m[w][j]>}$; $UB := UB \oplus B_{<w-m[w][j]>}$; }

S7: $D := D \oplus (B_w \,\&\, UA) \oplus (A_w \,\&\, UB)$; }

S8: Output D ;

Notes: For each $0 \le w \le n-1$, A_w=$A_{n-(n-w)}$ is stored in $\lceil \frac{n}{z} \rceil$ successive computer words starting from $DA[s][t]$ and ending at $DA[s][t+\lceil \frac{n}{z} \rceil -1]$, where $t = [(n-w)/z]$, $s = (n-w)$ && $(z-1)$ and && denotes integer bit-wise AND. In our implementation, these address computations are performed in the precomputation procedure, and starting addresses of A_w, $A_{<w-m[w][j]>}$, B_w and $B_{<w-m[w][j]>}$ are stored sequentially in a 1-dimensional array for $0 \le w \le n-1$.

5. Analysis and Comparison

We implement these algorithms in ANSI C using Microsoft Visual C++ 6.0 complier and test them on two computers:

1. An IBM ThinkPad 770X notebook with a 300 MHz Pentium II CPU running Windows NT 4.0.
2. A PC compatible computer with a 450 MHz Pentium III CPU running Windows 2000.

Both experimental results serve to validate our conclusions based primarily on theoretical considerations. Timings listed in this paper are obtained on the first computer.

We call the RH algorithm using precomputation tables *DA* and *DB* the improved RH algorithm. The time complexity of the original RH algorithm is determined in [3]. It needs n n-bit AND operations and $\frac{n-1}{2}+\frac{C_N-1}{2}=(C_N+n-2)/2$ n-bit XOR operations. The number of n-bit CSO of the original RH algorithm is equal to $(C_N+2n-1)/2$. While the original and improved RH algorithms require the same number of XOR and AND operations, the improved algorithm requires

$\frac{(C_N+1)}{2}+3z$ CSO. Thus it is faster than the original RH algorithm when n>3z+1.

Now we determine the time complexity of Algorithm 1. Since $\sum_{k=0}^{n-1} e_k = \sum_{i=1}^{v} h_i = (C_N-1)/2$, the total number of XOR in line S7 and S8 is $(C_N-1)/2$. Thus the total number of XOR operations in Algorithm 1 is $\frac{n-1}{2}+\frac{C_N-1}{2}=(C_N+n-2)/2$. Obviously, Algorithm 1 requires n AND operations. Thus the two RH algorithms and Algorithm 1 require the same number of XOR and AND operations.

It is well known that $C_N \geq 2n-1$, thus the total number of CSO in the improved RH algorithm is at least n+3z. Since e_k may be zero for some k's, one can see that the total number of CSO in Algorithm 1 is at most n+3z. Obviously, Algorithm 1 is faster than the improved RH algorithm for nonoptimal normal bases.

For $0 \leq i \leq n-1$, let $\beta_0\beta_i = \sum_{j=0}^{n-1} \phi_{i,j}\beta_j$ be the expansion of $\beta_0\beta_i$ with respect to the normal basis generated by β, where $\phi_{i,j} \in GF(2)$. The following matrix was defined in [7].

$$T_0 = \left(\phi_{i,j}\right)_{\substack{0\leq i\leq n-1,\\ 0\leq j\leq n-1}} \tag{15}$$

For type-II ONB, *t*he matrix T_0 defined for the type-II optimal normal basis is symmetric. So we know that the probability of e_k=0 is 0.25 for type-II ONBs in Algorithm 1. Our experiments show that for 100<n<1000, where n is odd and type-II ONB exists in $GF(2^n)$, the minimal, average and maximal percentages that e_k=0 are 22.9%, 25.0% and 27.2% respectively. That is to say, the average total number of CSO in Algorithm 1 is (3n/4)+3z. So Algorithm 1 should be faster than the improved RH algorithm for type-II ONBs. But numerical results show that this is only true for small values of n, e.g. 131, 233 and 293. The reason is that the size of temporary variables of Algorithm 1 is larger than that of the improved RH algorithm. In fact, while about 3zn/8 Bytes are needed to store arrays *DA* and *DB* in both algorithms, approximately n^2/16 Bytes are needed to store R_i's ($1 \leq i \leq v$) in statement S3 of Algorithm 1. This large array increases the transfer from the memory to the CPU cash in Algorithm 1 and makes it slow for large values of n.

Since computation of starting addresses of A_w, $A_{<w-m[w][j]>}$, B_w and $B_{<w-m[w][j]>}$ in Algorithm 2 may be performed in the precomputation procedure, it is easy to determine the time complexity of Algorithm 2, i.e. $4z$ CSO, $2n$ AND and C_N XOR operations.

Table 1 compares the time complexity of these NB algorithms for nonoptimal normal bases in $GF(2^n)$ where n is odd.

TABLE 1: Comparison of NB multiplication algorithms for nonoptimal normal bases.

	RH algorithm	Improved RH alg.	Algorithm 1	Algorithm 2
XOR	$(C_N+n-2)/2$	$(C_N+n-2)/2$	$(C_N+n-2)/2$	C_N
AND	n	n	n	$2n$
CSO	$(C_N+2n-1)/2$	$\frac{(C_N+1)}{2}+3z$	$< n+3z$	$4z$

We assume that the general-purpose processor can perform 1 n-bit XOR or AND using 1 n-bit operation. As defined in [3], we also assume that 1 CSO needs ρ n-bit operations. Our experiments and [3] show that the value of ρ is typically 4 for the C programming language if only simple logical instructions, such as AND, SHIFT and OR are used to emulated a k-fold cyclic shift. When $\rho=4$ and $z=32$, we may deduce the following condition that Algorithm 1 is faster than Algorithm 2: $C_N > 7n - 256$. Thus for high complexity NBs, Algorithm 1 is theoretically the fastest one among these NB algorithms. The experimental results listed in Table 4 confirm this conclusion.

Now we compare Algorithm 2 to the NY algorithm for type-I ONBs.

A bit-level NB multiplication algorithm was presented in the IEEE standard P1363-2000 [4]. Ning and Yin proposed a generalized version of this algorithm in [5]. The NY algorithm is very fast for ONBs, but it is slow for nonoptimal normal bases. The difference between Algorithm 2 and the NY algorithm is that a different multiplication matrix is used, i.e., Algorithm 2 uses the matrix T_0 defined in (15) and the NY algorithm uses the matrix M defined in Annex 6.3 of [4].

For type-I ONB, formula (14) requires about n XOR, n AND, $4z$ CSO, and 1 calculation of the

Hamming weight. The Hamming weight of A can be computed by a lookup table. For example, if we create a table with 2^8 entries on a 32-bit computer, our experimental results show that the cost to compute A's Hamming weight is no more than 4 times that of a field addition operation for n=162, 418 and 562.

Since no description of the precomputation procedure was presented in [5] (part of the NY algorithm was described in a patent application), we assume that the method introduced in Section 2 is used to perform this precomputation procedure. For the NY algorithm, DA_j and DB_j are defined as $z\lceil \frac{n}{z} \rceil$ -bit vector, thus the total number of CSO is about $2z$. Based on this assumption, it is easy to see that the fastest NY algorithm, Algorithm 4 of [5], requires about $2n$ n-bit XOR, n n-bit AND and $2z$ n-bit CSO. Thus the theoretical analysis shows that formula (14) is faster than the NY algorithm for n>260 if we assume that ρ =4 and z=32. Our experiments confirm this conclusion. Table 2 compares the time complexity of formula (14) and the fastest NY Algorithm for type-I ONBs. Timings of some type-I ONBs are listed in Table 3.

TABLE 2: Comparison of Formula (14) and the fastest NY Algorithm for type-I ONBs.

	XOR	AND	CSO
Formula (14)	n+4	n	$4z$
NY Algorithm	$2n$	n	$2z$

TABLE 3: Timing for some type-I ONBs (μs)

	$GF(2^{162})$	$GF(2^{226})$	$GF(2^{292})$	$GF(2^{418})$	$GF(2^{562})$
Formula (14)	31	49	67	122	195
NY Algorithm 4	31	49	69	136	216

We now compare these NB algorithms to the polynomial basis multiplication algorithm presented in [6], i.e., the finite field analogue of the Montgomery multiplication for integers. Since this method is significantly faster than the standard polynomial basis multiplication algorithm of [9], we only consider the Montgomery multiplication algorithm. For simplicity, we implement the multiplication algorithm in $GF(2^k)$ instead of $GF(2^n)$, where $k = w\lceil \frac{n}{w} \rceil$. From [6] we know that the case w=8 results in the fastest implementation on modern 32-bit computers. So we also select w=8, and employ the table lookup approach, which is shown to be the best choice to perform word-level multiplications [6].

Experimental results are listed in Table 4. The 5 binary fields recommended by NIST for ECDSA applications are $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$ and $GF(2^{571})$.

TABLE 4: Timing for some $GF(2^n)$s (μs)

n	Type	Original RH algorithm	Improved RH algorithm	Algorithm 1	Algorithm 2	NY algorithm	Montgomery algorithm
131	2	70	64	57	39	29	38
233	2	180	153	140	98	56	125
359	2	350	282	283	199	118	309
491	2	590	472	493	321	226	593
163	4	164	153	123	112	2500	57
277	4	373	333	260	236	11970	180
409	4	671	609	506	441	95840	421
577	4	1231	1070	993	825	278900	844
673	4	1593	1382	1320	1071	438900	1167
739	4	1872	1600	1578	1277	576600	1390
283	6	516	479	318	339	12490	190
503	6	1326	1212	914	865	177200	622
751	6	2726	2483	1989	1841	616100	1476
599	8	2241	2097	1444	1524	300300	896
571	10	2454	2317	1481	1684	258600	817
563	14	3308	3185	1782	2198	251300	801

Table 4 shows that for some $GF(2^n)$s where type 4 GNBs exist, Algorithm 2 is faster than the Montgomery algorithm. For $GF(2^{409})$, Algorithm 2 is slightly slower than the Montgomery algorithm. So for applications where many squaring operations are needed, e.g. exponentiation, Algorithm 2 is a better choice.

6. Conclusions

We have presented two normal basis multiplication algorithms in $GF(2^n)$. Algorithm 1 is suitable for high complexity NBs and Algorithm 2 is fast in $GF(2^n)$ where type-I ONBs or low complexity NBs exist.

7. Acknowledgments

We would like to thank people who had read this paper and given helpful comments.

References

[1] Haining Fan, "Simple Multiplication Algorithm for a Class of $GF(2^n)$," *Electronics Letters*, vol. 32, No.7, pp.636-637, 1996.

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

[3] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors," *Technical Report CORR 2001-25, Dept. of C&O, University of Waterloo, Canada*, April 19, 2001.

[4] IEEE P1363-2000. "Standard Specifications for Public Key Cryptography," August 2000.

[5] P. Ning and Y.L. Yin, "Efficient Software Implementation for Finite Field Multiplication in Normal Basis," In *LNCS 2229 as Proceedings of 3rd International Conference on Information and Communications Security* (*ICICS*) *2001*, pp.177-188, Xian, China, 2001. Springer Verlag.

[6] C. Koc and T. Acer, "Montgomery multiplication in $GF(2^k)$," *Design,Codes and Cryptography*, vol. 14, No.1, pp.57-69, Apr. 1998.

[7] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson, "Optimal normal bases in $GF(p^n)$," *Discrete Applied Mathematics*, vol. 22, pp.149-161, 1988/89.

[8] S. Gao and Jr. H.W. Lenstra, "Optimal Normal Bases," *Design,Codes and Cryptography*, 2:315-323, 1992.

[9] J. Lopez and R. Dahab. "High-Speed software multiplication in $F(2^m)$," *Technical report, IC-00-09*, May 2000. Available at http://www.dcc.unicamp.br/ic-main/publications-e.html.