Efficient Software Implementation for Finite Field Multiplication in Normal Basis

Peng Ning[1] and Yiqun Lisa Yin[2]

[1] Department of Computer Science
North Carolina State University, Raleigh, NC 27695, USA
ning@csc.ncsu.edu
[2] NTT Multimedia Communications Laboratories, Inc.
250 Cambridge Avenue, Palo Alto, CA 94306, USA
yiqun@nttmcl.com

Abstract. Finite field arithmetic is becoming increasingly important in today's computer systems, particularly for implementing cryptographic operations. Among various arithmetic operations, finite field multiplication is of particular interest since it is a major building block for elliptic curve cryptosystems. In this paper, we present new techniques for efficient software implementation of binary field multiplication in normal basis. Our techniques are more efficient in terms of both speed and memory compared with alternative approaches.

1 Introduction

Finite field arithmetic is becoming increasingly important in today's computer systems, particularly for implementing cryptographic operations. Among the more common finite fields in cryptography are odd-characteristic finite fields of degree 1 and even-characteristic finite fields of degree greater than 1. The latter is conventionally known as $GF(2^m)$ arithmetic or binary field arithmetic. $GF(2^m)$ arithmetic is further classified according to the choice of basis for representing elements of the finite field; two common choices are polynomial basis and normal basis.

Fast implementation techniques for $GF(2^m)$ arithmetic have been studied intensively in the past twenty years. Among various arithmetic operations, $GF(2^m)$ multiplication has attracted most of the attention since it is a major building block for implementing elliptic curve cryptosystems. Depending on the choice of basis, the mathematical formula for a $GF(2^m)$ multiplication can be quite different, thus making major differences in practical implementation. Currently, it seems that normal basis representation (especially optimal normal basis) offers the best performance in hardware [9–11], while in software polynomial basis representation is more efficient [2, 3, 8].

For interoperability, it is desirable to support both types of basis in software, which can be done either by implementing arithmetic in both bases or by implementing one basis together with basis conversion algorithms. Various basis

conversion techniques [4–6] have been proposed with performance tradeoffs. Because of the overhead of basis conversion, supporting both bases directly seems preferable than basis conversion for certain applications.

There has not been much study related to implementing normal basis multiplication in software, in contrast with the amount of work related to polynomial basis. The main difficulties for fast normal basis multiplication in software are due to the particular computation process: First, when multiplying two elements represented in normal basis according to the standard formula, the coefficients of their product need to be computed one bit at a time. Second, the computation of a given bit involves a series of "partial sums" which need to be computed sequentially in software, while this is easily parallelized in hardware.

In this paper, we present new techniques for efficient software implementation of normal basis multiplication, part of which were originally described in a patent application [13]. At the core of our method are a mathematical transformation and a novel way of doing precomputation, which significantly reduce both time and memory complexity.

To study the effectiveness of our techniques, we compare our approach with the best alternative one[1] developed by Rosing [14]. Our approach is much more efficient than his method in terms of both speed and memory. Speed wise, analysis and experimental results show that there is a significant speed up using our new techniques. Memory wise, the number of bytes stored is only $O(m)$ compared with $O(m^2)$ in [14]. This is especially useful for memory constraint devices – environments that elliptic curve cryptosystems seem more attractive than conventional public key cryptosystems such as RSA. Our techniques for field multiplication can also be combined with elliptic curve arithmetic to provide further speed up.

The rest of the paper is organized as follows. In Section 2, we provide some mathematical background, and in Section 3, we review the related work on normal basis multiplication for both software and hardware. In Section 4, we present our new multiplication techniques, and in Section 5, we summarize experimental results. Some further discussions on related issues are included in Section 6, and concluding remarks are given in Section 7.

2 Mathematical background

In this section, we first define some basic notations for finite field $GF(2^m)$ and its representation in normal basis. Then, we describe the multiplication formulas for both general normal basis and optimal normal basis.

Since we are considering software implementation, we will use w to denote the word size throughout the paper. For simplicity, we assume that $w|m$.

[1] The recent result in [15] has better performance than the Rosing's method for certain choices of m. However, our approach remains the fastest among the known methods. Please see section 3.

2.1 Finite field $GF(2^m)$ and normal basis representation

The finite field $GF(2^m)$ is the set of all 2^m possible 0-1 strings of length m, with certain rules for field addition and multiplication. The finite field $GF(2^m)$ have various basis representations including normal basis representation.

A binary polynomial is a polynomial with coefficients in $GF(2)$. A binary polynomial is irreducible if it is not the product of two binary polynomials of smaller degrees. For simplicity, we will refer to such a polynomial an irreducible polynomial. Irreducible polynomials exist for every degree m and can be found efficiently.

Let $g(x)$ be an irreducible polynomial of degree m. If β is a root of $g(x)$, then the m distinct roots of $g(x)$ in $GF(2^m)$ is given by $B = (\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}})$. If the elements of B are linearly independent, then $g(x)$ is called a normal polynomial and B is called a normal basis for $GF(2^m)$ over $GF(2)$. Normal polynomials exist for every degree m. For certain choices of m, $x^m + x^{m-1} + x^{m-2} + \cdots + x + 1$ is a normal polynomial. Given any element $a \in GF(2^m)$, one can write

$$a = \sum_{i=0}^{m-1} a_i \beta^{2^i}, \text{ where } a_i \in \{0, 1\}.$$

2.2 Multiplication with general normal basis

In normal basis, field multiplication is usually carried out using a multiplication matrix, which is an m-by-m matrix M with each entry $M_{ij} \in GF(2)$. Details on how to compute matrix M from $g(x)$ can be found in [4]. The complexity of M, denoted by C_m, is defined to be the number 1's in M. It is well known that $C_m \geq 2m - 1$.

Let $a = (a_0, a_1, ..., a_{m-1})$ and $b = (b_0, b_1, ..., b_{m-1})$ be two elements represented in normal basis, and let $c = (c_0, c_1, ..., c_{m-1})$ be their product. Then each coefficient c_k is computed as follows[2].

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

In a straightforward software implementation of formula (1), a, b, and each column of M are all stored in m/w computer words. A matrix-vector multiplication Mb^T can be carried out with $(m/2)(m/w)$ word operations on average, and hence the total number of word operations for computing c is about $m(m/2)(m/w) = m^3/2w$. Note that the computation time is independent of the complexity C_m.

If we spell out formula (1), we obtain the following equation for c_k.

$$c_k = \bigoplus_{i=0}^{m-1} \left[a_{k+i} \cdot \left(\bigoplus_{j=0}^{m-1} M_{ij} \cdot b_{j+k} \right) \right]. \tag{2}$$

[2] Throughout the paper, the additions "+" in the subscripts are understood as additions modulo the degree m, unless otherwise specified.

In formula (2), essentially the same expression is used for each coefficient c_k. More specifically, given the expression for c_k, one can just increase the subscripts of a and b by one (modulo m) to obtain the expression for c_{k+1}. Formula (2) will be useful in later discussions.

2.3 Multiplication with optimal normal basis

An optimal normal basis (ONB) [1, 12] is a normal basis which has the lowest complexity. That is, $C_m = 2m - 1$. Optimal normal bases only exist for certain degree m. In the range [150, 600], there are only 101 degrees for which ONB exists.

There are two kinds of normal basis called type I ONB and type II ONB. They differ in the mathematical formulas which define them. The matrix M has the form that the first row has a single non-zero entry, and the rest of each row has exactly two non-zero entries. So the matrix M can be stored more compactly using two tables $t1[i]$ and $t2[i]$, which are the indices of the two non-zero entries in row i of M. Using $t1$ and $t2$, formula (2) can be rewritten as follows.

$$c_k = (a_k \cdot b_{t1[0]+k}) \oplus \left(\bigoplus_{i=1}^{m-1} \left[a_{k+i} \cdot \left(b_{t1[i]+k} \oplus b_{t2[i]+k} \right) \right] \right). \tag{3}$$

3 Related Work

3.1 Hardware

In formula (1), when a new bit c_k needs to be computed, the coefficients of both a and b are rotated to the left by one bit. This fact is useful for efficient hardware implementation of normal basis multiplication [9–11], since the same circuit that represents M can be repeatedly used and each coefficient can be computed in one clock cycle.

Even though the sequence of operations for each coefficient is easily parallelized in hardware, it is quite difficult to mimic the same technique in a software implementation since these operations are inherently sequential in software.

3.2 Software

Fast software implementation techniques for normal basis multiplication have been centered around optimal normal basis. In [7], a method for type I optimal normal basis was considered. The idea is to use polynomial-basis-like multiplication and take advantage of the special form of the irreducible polynomials for type I ONB. Their method does not seem to extend to type II ONB or other normal basis.

In [14], Rosing presented an efficient method for ONB multiplication. The main idea is that the partial sum $a_{k+i} \cdot \left(b_{t1[i]+k} \oplus b_{t2[i]+k} \right)$ in formula (3) can be computed simultaneously for different coefficients (different subscript i) using

word operations in software. To do this, some preprocessing of b is necessary. At a high level, Rosing's method can be summarized as follows.

- *Precomputation*: compute and store m rotations of b.
- *Main loop*: for each $i = 1, 2, ..., m-1$, compute the partial sum $a_{k+i} \cdot \left(b_{t1[i]+k} \oplus b_{t2[i]+k}\right)$. As a special case, when $i = 0$, the partial sum $a_k \cdot b_{t1[0]+k}$ is computed.

In the main loop, each partial sum is computed in $O(m/w)$ operations, for a total of $O(m^2/w)$ operation. For the precomputation, the number of operations for computing all rotations of b is also $O(m^2/w)$, and the total number of bytes stored is $m^2/8$. Note that for precomputation, both time and memory grow quadratically as m increases.

Our approach does share a feature similar to Rosing's method: Our approach also computes multiple bits of the (partial) result simultaneously using word operations. However, our approach employs a very different precomputation technique. As we will show in section 5, our technique reduces the time and memory complexity for precomputation from quadratic to linear, yielding a much more efficient algorithm than Rosing's method.

Reyhani-Masoleh et al. recently proposed a series of fast normal basis multiplication algorithms based on some mathematical transformations [15]. According to their timing result [15], our approach is about twice as fast as their most efficient algorithm. For example, the running time for $m = 299$ (ONB) reported in [15] is 114 μs on Pentium III 533 MHz, which can be scaled to 101 μs on Pentium III 600 MHz (our platform). In comparison, our implementation takes 42.36 μs on Pentium III 600 MHz. Note that it is possible to combine our techniques with those in [15]; however, we do not cover this topic here but consider it as possible future work.

4 Our Techniques

In this section, we present our techniques for an efficient software implementation of normal basis multiplication. We begin with a basic method for general normal basis. Then, we present a simple yet effective improvement to the basic method. Finally, we discuss how our approach can be applied to ONB to provide much better performance.

4.1 The Basic Method

At the core of our method is a new way of doing the precomputation, which significantly reduces both time and memory complexity. First, we define the quantities that need to be precomputed. For $i = 0, 1, ..., m-1$, let

$$A[i] = (a_i, a_{i+1}, ..., a_{i+w-1}),$$
$$B[i] = (b_i, b_{i+1}, ..., b_{i+w-1}),$$
$$C[i] = (c_i, c_{i+1}, ..., c_{i+w-1}).$$

In other words, each $A[i]$ has length w and they are the successive blocks of a in a wrap-around fashion, and similarly for $B[i]$ and $C[i]$. It is easy to see that $c = (C[0], C[w], C[2w], \cdots, C[(\frac{m}{w} - 1)w])$. Now we can rewrite formula (2) using A, B, C as follows:

For t from 0 to $(m/w - 1)$

$$C[wt] = \bigoplus_{i=0}^{m-1} \left[A[(i + wt) \bmod m] \cdot \left(\bigoplus_{j=0}^{m-1} M[i, j] B[(j + wt) \bmod m] \right) \right]. \quad (4)$$

We can see that the total number of equations in formula (4) is m/w, and one equation in formula (4) corresponds to w consecutive equations in formula (1) or formula (2). In software implementation of formula (4), "$\cdot$" can be computed as a bit-wise AND operation between two words, and "$\bigoplus$" can be computed as a bit-wise Exclusive-OR operation between two words. Therefore, during the computation process, the quantities involved are only $A[i]$'s and $B[i]$'s which have already been precomputed, and the operations involved are only word operations.

The following gives a straightforward implementation of formula (4) in C.

Algorithm 1.

```
precompute arrays A and B;
for (t = 0; t < (m/w); t++) {
    C[w*t] = 0;
    for (i = 0; i < m; i++) {
        temp = 0;
        for (j = 0; j < m; j++)
            if (M[i,j]==1) temp ^= B[(j+w*t)%m];
        C[w*t] ^= A[(i+w*t)%m] & temp;
    }
}
```

The total number of word operations for the main loop is $O(C_m \cdot m/w)$. The number of operations for precomputing the arrays A and B is $O(m)$, and the total number of precomputed bytes is $2 \times m \times (w/8) = w \cdot m/4$, which is $8m$ for typical PC implementation. Note that both time and memory complexity is linear in m for the precomputation phase.

4.2 Further Optimization

We can further speed up the basic method in Section 4.1 by precomputing and storing the arrays A and B in a clever way to avoid all the modulo m computation for indexing in the main loop.

To achieve this, we first extend the definition of array A and B as follows: For $i = 0, ...m - 1$, we define[3]

$$A[i + m] = A[i] = (a_i, a_{i+1}, ..., a_{i+w-1}),$$
$$B[i + m] = B[i] = (b_i, b_{i+1}, ..., b_{i+w-1}).$$

[3] Here, the addition in $A[i + m]$ and $B[i + m]$ is a real addition without modulo m.

We precompute array A and B, each of which consists of $2m$ elements of length w:

$$A[0], A[1], ..., A[m-1], A[m], A[m+1], ..., A[2m-1],$$
$$B[0], B[1], ..., B[m-1], B[m], B[m+1], ..., B[2m-1].$$

Given A and B, we can improve the C code in Section 4.1:

Algorithm 2.

```
precompute arrays A and B;
for (k = 0; k < m; k += w) {
    C[k] = 0;
    for (i = 0; i < m; i++) {
        temp = 0;
        for (j = 0; j < m; j++)
            if (M[i,j]==1) temp ∧= B[j];
        C[k] ∧= A[i] & temp;
    }
    A += w; B += w;
}
```

The idea in the above code is the following: When computing $C[0]$, we use word 0 through $m-1$ in array A and B (that is, the first m words). When computing $C[w]$, we use word w through $w+m-1$ in array A and B, which is accomplished by two easy pointer jumping. Similarly, we can compute $C[2w], ..., C[m-w]$.

This way, the arrays A and B are accessed sequentially within the main loop, significantly improving the speed at some cost of the memory. The number of operations for precomputing A and B remains the same, and the total number of precomputed bytes is $w \cdot m/2$, which is $16m$ for typical PC implementation.

4.3 Applying the Techniques to ONB

Algorithm 2 in the preceding section can be simplified using the fact that the inner loop j no longer exists for ONB, since it only involves one or two elements of B. Here we assume the non-zero entry of the first row of M is stored in `t1[0]` and the two non-zero entries of row i ($0 < i < m$) are stored in `t1[i]` and `t2[i]`, respectively (see section 2.3).

Algorithm 3.

```
precompute arrays A and B;
for (k = 0; k < m; k += w) {
    temp = A[0] & B[t1[0]];
    for (i = 0; i < m; i++)
        temp ∧= A[i] & (B[t1[i]] ∧ B[t2[i]]);
    C[k] = temp;
    A += w; B += w;
}
```

The implementation for type I ONB can be further improved by taking advantage of the special form of its multiplication matrix. For type I ONB, the non-zero entry of the first row of M is always in column $m/2$, and one of the two non-zero entries of row i is in column $m/2 + i \bmod m$ [14]. Thus, we can compute one non-zero entry row i of M, say `t1[i]`, as $i + m/2 \bmod m$.

This fact can be combined with the precomputation to reduce one table lookup in the inner loop of Algorithm 3. The idea is to further extend the array B by $m/2$ words such that for $i = 2m, 2m+1, \cdots, 2m+m/2-1$, $B[i] = B[i-m]$. Then $B[t1[i]] = B[i+m/2 \bmod m]$ in the inner loop can be replaced by $B[i+m/2]$ (without involving the mod operation), and thus we can use another pointer $D = B + m/2$ and further replace $B[i + m/2]$ with $D[i]$. As a result, the above code can be improved as follows.

Algorithm 4.

```
precompute arrays A and B;
D = B + m/2;
for (k = 0; k < m; k += w) {
    temp = A[0] & D[0];
    for (i = 1; i < m; i ++)
        temp ∧= A[i] & (D[i] ∧ B[t2[i]]);
    C[k] = temp;
    A += w; B += w; D += w;
}
```

5 Performance Results

To evaluate the performance of our methods, we performed a series of experiments for both type I and type II ONB on a Pentium III 600 PC running Windows 2000 Professional. The programs were written in C, and the timing results were computed by averaging the timing for 100,000 multiplications of random field elements. The rest of this section gives the performance data. In particular, our methods are compared with Rosing's method in terms of timings and memory requirements.

In FIPS 186-2 [16], NIST recommenced 10 finite fields: 5 prime fields and 5 binary fields. The lengths of the fields were chosen so that the corresponding elliptic curve cryptographic (ECC) systems would have comparable security to symmetric ciphers of key lengths 80, 112, 128, 192, 256. Since ONB does not exist for every field length m, we choose field lengths that are closest to the NIST recommenced field lengths. Table 1 lists the specific field lengths.

Table 2 shows the timings of type I ONB multiplications for the dimensions in table 1. Compared with Rosing's method, our general method (Algorithm 3) reduces the execution time for type I ONB multiplication by about 70%, while our enhanced method (Algorithm 4) further reduces the time by about 5%. As a result, the execution time of type I ONB multiplication is reduced by about 75%. Table 3 shows the timings of type II ONB multiplications for the dimensions listed in table 1. Though Algorithm 4 cannot be applied to type II ONB,

Table 1. NIST recommenced lengths of binary finite field for ECC.

Symmetric cipher key length	Algorithms	Dimension m of $GF(2^m)$	Type I ONB	Type II ONB
80	Skipjack	163	162	158
112	triple-DES	233	226	233
128	AES-128	283	292	281
192	AES-192	409	418	410
256	AES-256	571	562	575

Table 2. Timings for multiplication with Type I ONB (μs).

Dimension m	Rosing	Algorithm 3	Time Reduced	Algorithm 4	Time Reduced
162	55.48	17.5	68.46%	14.62	73.65%
226	92.74	28.05	69.75%	23.53	74.63%
292	137.39	41.55	69.76%	34.35	75.00%
418	257.57	76.6	70.26%	62.19	75.86%
562	426.21	125.2	70.62%	98.64	76.86%

Algorithm 3 still reduces the execution time by about 70% compared with Rosing's method. It is not difficult to conclude that our methods significantly reduce the time required for multiplication. Figure 1 also shows the overall timings for all three methods, where the dimension m ranges from 150 to 600.

Our methods not only save the execution time of ONB multiplications, but also reduce the memory requirements compared with Rosing's method. To save the precomputed rotations of one operand, Rosing's method requires a temporary array having m entries, each of which keeps one rotation of the operand. Thus, Rosing's method requires $m^2/8$ bytes. In contrast, our general method (Algorithm 3) needs $2m \cdot w/8$ bytes for each operand, and therefore totally requires $m \cdot w/2$ bytes. Our enhanced method for type I ONB (Algorithm 4) needs additional $m \cdot w/16$ bytes, and thus requires $9m \cdot w/16$ bytes in total. As shown in table 4, Algorithm 3 reduces the memory requirement up to 77%, and Algorithm 4 reduces the memory requirement up to 74% for the dimensions in table 1.

6 Discussions

Normal basis vs. polynomial basis For interoperability, it is desirable to support both bases in software, which can be done either by implementing both bases directly or by implementing one basis together with basis conversion algorithms.

Various software implementation techniques for polynomial basis have been proposed in recent years. The fastest method is described in [8], and a good survey is given in [3]. Some of the techniques can be very efficient when special

Table 3. Timings for multiplication with Type II ONB (μs).

Dimension m	Rosing	Algorithm 3	Time Reduced
158	41.26	14.12	66.78%
233	97.44	28.64	70.61%
281	124.68	36.75	70.52%
410	240.05	70.1	70.80%
575	433.42	127.59	70.56%

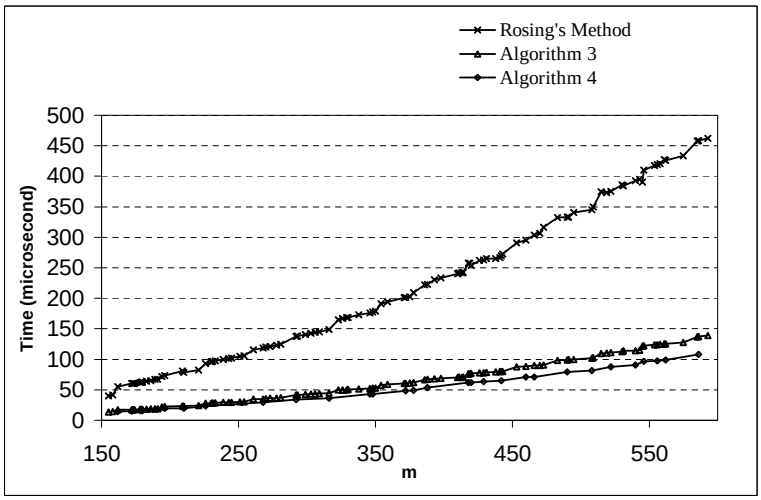

Fig. 1. Timings for all three methods with m ranging from 150 to 600.

irreducible polynomials are used. So the question is whether it is enough to just support polynomial basis in software together with basis conversion.

The general method for basis conversion involves storing a conversion matrix W and computing a matrix-vector multiplication for each conversion [4]. The size of W is $m^2/8$ bytes, which can be quite large for memory constraint devices. For example, when $m = 512$, the memory requirement is over $32K$ bytes (and $64K$ bytes if both W and W^{-1} are stored for efficient conversion in both directions). Methods for storage efficient basis conversion were proposed in [5, 6]. Such methods only need to store $O(m)$ bytes and then each conversion takes about $O(m)$ field multiplication. The extra $O(m)$ multiplication can be a slowdown factor for some implementation. We note that all the basis conversion methods assume that both bases are known *before* the communication, and certain quantities related to both bases (e.g., W) are already precomputed and stored. While this is easily done for certain applications, it may not be the case

Table 4. Memory requirements for precomputation: number of precomputed bytes ($w = 32$).

Dimension m	Rosing's Method ($m^2/8$)	Algorithm 3 ($w \cdot m/2$)	Memory Reduced	Algorithm 4 for Type I ONB ($9w \cdot m/16$)	Memory Reduced
158	3.05K	2.47K	18.99%	-	-
162	3.20K	2.53K	20.99%	2.85K	11.11%
226	6.23K	3.53K	43.36%	3.97K	36.28%
233	6.63K	3.64K	45.06%	-	-
281	9.64K	4.39K	54.45%	-	-
292	10.41K	4.56K	56.16%	5.13K	50.68%
410	20.52K	6.41K	68.78%	-	-
418	21.33K	6.53K	69.38%	7.35K	65.55%
562	38.56K	8.78K	77.22%	9.88K	74.38%
575	40.36K	8.98K	77.74%	-	-

for other applications. In general, computing these quantities on the fly can be time consuming, which adds more complexity to basis conversion. Therefore, due to the overhead of basis conversion, sometimes supporting both bases directly seems preferable than basis conversion.

Using the new techniques in ECC arithmetic Our techniques for field multiplication can be combined with elliptic curve arithmetic to provide further speed up. Since some field elements are repeatedly used in ECC operations, we do not have to perform precomputation for these elements after the first time they are involved in a multiplication. For example, using projective coordinates, we need 15 finite field multiplication for point addition. Using our method, each multiplication needs to precompute one array for each of the two operands. So we totally need to precompute 30 arrays. By storing some of the precomputed results, the number of precomputed arrays can be reduced to 20. This can be used to further reduce the time for point additions. For example, the precomputation of each operand for $GF(2^{162})$ multiplication takes about 20% of the total multiplication time. Thus, we can save another 13% for each point addition reusing the precomputed results.

7 Conclusions

In this paper, we studied efficient software implementation for $GF(2^m)$ multiplication in normal basis. We presented new techniques for normal basis multiplication. In particular, our methods were optimized for both type I and type II ONB. Our techniques are more efficient in terms of both speed and memory compared with alternative approaches.

References

1. D. Ash, I. Blake, and S.A. Vanstone. *Low Complexity Normal Basis*. Discrete Applied Mathematics, Vol. 25, 1989.
2. E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. *A fast software implementation for arithmetic operations in* $GF(2^n)$. In Proc. *Asiacrypt'96*, 1996.
3. D. Hankerson, J.L. Hernandez, and A. Menesez. *Software Implementation of Elliptic Curve Cryptography over Binary Fields*. In Proc. *CHES'2000*, August 2000.
4. IEEE P1363-2000. *Standard Specifications for Public Key Cryptography*. August 2000.
5. B. Kaliski and M. Liskov. *Efficient Finite Field Basis Conversion Involving a Dual Basis*. In Proc. *CHES'99*, August 1999.
6. B. Kaliski and Y.L. Yin. *Storage Efficient Finite Field Basis Conversion*. In Proc. *SAC'98*, August 1998.
7. R.J. Lambert and A. Vadekar. *Method and apparatus for finite field multiplication*. US Patent 6,049,815, April 2000.
8. 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`.
9. J. L. Massey and J. K. Omura. *Computational method and apparatus for finite field arithmetic*. U.S. Patent 4,587,627, May 1986.
10. R.C. Mullin. *Multiple Bit Multiplier*. U.S. Patent 5,787,028, July 1998.
11. R.C. Mullin, I.M. Onyszchuk, and S.A. Vanstone. *Computational Method and Apparatus for Finite Field Multiplication*. U.S. Patent 4,745,568, May 1988.
12. R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R. Wilson *Optimal Normal Basis in* $GF(p^m)$. Discrete Applied Mathematics, Vol. 22, 1988/1989.
13. P. Ning and Y. L. Yin. *Efficient Software Implementation for Finite Field Multiplication in Normal Basis*. Pending US Patent Application. Provisional patent application filed in December 1997.
14. M. Rosing. *Implementing Elliptic Curve Cryptography*. Manning Publications Co., 1999.
15. A. Reyhani-Masoleh, M. A. Hasan. *Fast Normal Basis Multiplication Using General Purpose Processors*. To appear in the 8th Workshop on Selected Areas in Cryptography (SAC 2001). August 2001.
16. National Institute of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186-2, February 2000.