High-Speed Software Multiplication in $\mathbb{F}_{2^m}$

Julio López *Ricardo Dahab*

Relatório Técnico IC–00-09

Maio de 2000

High-Speed Software Multiplication in $\mathbb{F}_{2^m}$

Julio López[*] Ricardo Dahab[†]

Institute of Computing
State University of Campinas
Campinas, 13081-970 São Paulo, Brazil
{julioher,rdahab}@dcc.unicamp.br

May 22, 2000

Abstract

In this paper we describe an efficient algorithm for multiplication in $\mathbb{F}_{2^m}$, where the field elements of $\mathbb{F}_{2^m}$ are represented in standard polynomial basis. The proposed algorithm can be used in practical software implementations of elliptic curve cryptography. Our timing results, on several platforms, show that the new method is significantly faster than the "shift-and-add" method.

Key words. Multiplication in $\mathbb{F}_{2^m}$, Polynomial Basis, Elliptic Curve Cryptography.

1 Introduction

Efficient algorithms for multiplication in $\mathbb{F}_{2^m}$ are required to implement cryptosystems such as the Diffie-Hellman and elliptic curve cryptosystems defined over $\mathbb{F}_{2^m}$. Efficient implementation of the field arithmetic in $\mathbb{F}_{2^m}$ depends greatly on the particular basis used for the finite field. Two common choices of bases for $\mathbb{F}_{2^m}$ are normal and polynomial. Normal bases seem more suitable for hardware implementations (see [1]).

In this paper we describe a technique for multiplication in the finite field $\mathbb{F}_{2^m}$, where the field elements are represented as binary polynomials modulo an irreducible binary polynomial of degree m. The proposed method is about 2-5 times faster than the standard multiplication, and is particularly useful for software implementation of elliptic curve cryptosystems over $\mathbb{F}_{2^m}$. It is based on the observation that Lim/Lee's method [6] (or comb method [7]), designed for exponentiation, can be modified to work in $\mathbb{F}_{2^m}$.

The remainder of this paper is organized as follows. In Section 2 we describe the finite field $\mathbb{F}_{2^m}$ using a polynomial basis, along with a description of the standard algorithm for multiplication in $\mathbb{F}_{2^m}$. A description of a simple version of Lee/Lim's method and two versions of the proposed method are described in Section 3. In Section 4, we present timing results on different computational platforms.

[*]Institute of Computing, State University of Campinas, 13081-970 Campinas, SP, Brazil, and Dept. of Computer Science, University of Valle, Cali, Colombia.

[†]Institute of Computing, State University of Campinas, 13081-970 Campinas, SP, Brazil. Research partially supported by a Pronex-Finep grant 107/97.

2 The finite field $\mathbb{F}_{2^m}$

2.1 Polynomial basis representation

In this section we describe the finite field $\mathbb{F}_{2^m}$, called a *characteristic two finite field* or a *binary finite field*, in terms of a *polynomial basis representation.* Let $f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$ (where $f_i \in \{0,1\}$, for $i = 0, \ldots, m-1$) be an irreducible polynomial of degree m over $\mathbb{F}_2$; polynomial $f(x)$ is called the *reduction polynomial.* A *polynomial basis* is specified by a reduction polynomial. In such a representation, the bit string $(a_{m-1} \ldots a_1 a_0)$ is taken to represent the polynomial

$$a_{m-1}x^{m-1} + \ldots + a_1 x^1 + a_0$$

over $\mathbb{F}_2$. Thus, the finite field $\mathbb{F}_{2^m}$ can be represented by the set of all polynomials of degree less than m over $\mathbb{F}_2$. That is,

$$\mathbb{F}_{2^m} = \{(a_{m-1} \ldots a_1 a_0) \mid a_i \in \{0,1\}\}.$$

The field arithmetic is implemented as polynomial arithmetic modulo $f(x)$. In this representation, addition and multiplication of $a = (a_{m-1} \ldots a_1 a_0)$ and $b = (b_{m-1} \ldots b_1 b_0)$ are performed as follows:

- *Addition:* $a + b = (c_{m-1} \ldots c_1 c_0)$, where $c_i = (a_i + b_i) \bmod 2$.
- *Multiplication:* $c = a \cdot b = (c_{m-1} \ldots c_1 c_0)$, where the polynomial $c(x) = \sum_{i=0}^{m-1} c_i x^i$ is the remainder of the division of polynomial $(\sum_{i=0}^{m-1} a_i x^i) \cdot (\sum_{i=0}^{m-1} b_i x^i)$ by $f(x)$. That is, $c = ab \bmod f$.

For efficiency reasons, the reduction polynomial can be selected as a *trinomial* $x^m + x^k + 1$, where $1 \le k \le m-1$ or a *pentanomial* $x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, where $1 < k_1 < k_2 < k_3 < m-1$. ANSI X9.62 [2] specifies several rules for choosing the reduction polynomial.

In software implementations, we partition the bit representation of a field element $a = (a_{m-1} \ldots a_1 a_0)$ into blocks of the same size. Let w be the word size of a computer (typical values are $w = 8, 16, 32, 64$), and s be the number of words required to pack a into words. That is, $s = \lceil m/w \rceil$. Then, we can write a as an sw-bit number consisting of s words, where each word is of length w. Thus, we can write

$$a = (A_{s-1} \ldots A_1 A_0),$$

where each A_i is of length w and

$$A_i = (a_{iw+w-1} \ldots a_{iw+1} a_{iw}) \in \mathbb{F}_{2^w}.$$

In polynomials terms,

$$a(x) = \sum_{i=0}^{s-1} A_i(x) x^{iw} = \sum_{i=0}^{s-1} \sum_{j=0}^{w-1} a_{iw+j} x^{iw+j}. \tag{1}$$

2.2 Recent methods for multiplication in $\mathbb{F}_{2^m}$

In recent years, several algorithms for software multiplication in $\mathbb{F}_{2^m}$ have been reported; however, we are interested in techniques that can be used when m is prime.[1] In Schroeppel *et al.* [10] various programming tricks are discussed for implementing the "shift-and-add" method, a basic algorithm for multiplication in $\mathbb{F}_{2^m}$. A slight variant of this method is described by De Win *et al.* [11]. In Koç [5], a word-level Montgomery multiplication algorithm in $\mathbb{F}_{2^m}$ is proposed. This method is significantly faster than the standard method whenever the multiplication of two words of size w, each one representing a polynomial in $\mathbb{F}_{2^w}$ can be performed in few cycles. Since this operation is not available in most general purpose processors, the alternative is to use table lookup. This approach requires, for example, 128 Kbytes for $w = 8$ and 16 Gbytes for $w = 16$, making it less attractive for practical applications. Another well known method for multiplication in $\mathbb{F}_{2^m}$ is that of Karatsuba (see for example [4]).

2.3 The "shift-and-add" method

In this section we describe the basic method for computing $c(x) = a(x) \cdot b(x) \bmod f(x)$ in $\mathbb{F}_{2^m}$. It is analogous to the binary method for exponentiation, with the square and multiplication operations being replaced by the SHIFT (multiplication of a field element by x) and field addition operations, respectively. Thus, the "shift-and-add" method processes the bits of polynomial $a(x)$ from left to right, and uses the following equation to perform $c = ab \bmod f$:

$$c(x) = x(\cdots x(xa_{m-1}b(x) + a_{m-2}b(x) \bmod f(x)) + \cdots) + a_0 b(x) \bmod f(x).$$

Assume that $a(x) = \sum_{i=0}^{s-1} A_i x^{wi}$, $b(x) = \sum_{i=0}^{s-1} B_i x^{wi}$, and $f(x) = \sum_{i=0}^{s-1} F_i x^{wi}$. Then the steps of the "shift-and-add" method are given below.

Algorithm 1: the "shift-and-add" method.

INPUT: $a = (A_{s-1} \ldots A_0)$, $b = (B_{s-1} \ldots B_0)$, and $f = (F_{s-1} \ldots F_0)$.
OUTPUT: $c = (C_{s-1} \ldots C_0) = a \cdot b \bmod f$.

```
1.  Set $k \leftarrow m - 1 - w(s-1)$,   $c \leftarrow 0$
2.  for $i$ from $s-1$ downto 0 do
        for $j$ from $k$ downto 0 do
            Set $c \leftarrow$ SHIFT($c$)
            if $a_{iw+j} = 1$ then $c \leftarrow c \oplus b$
            if $c_m = 1$ then $c \leftarrow c \oplus f$
        Set $k \leftarrow w - 1$
3.  return ($c$).
```

This algorithm requires $m - 1$ shift operations and m field additions on average, but the number of field additions can be reduced by selecting the reduction polynomial $f(x)$ as a

[1]Many standards that include elliptic curves defined over $\mathbb{F}_{2^m}$ recommend for security reasons, the use of binary finite fields with the property that m be prime.

trinomial or a pentanomial. Observe that in this algorithm, the multiplication step (the computation of $d(x) = a(x) \cdot b(x)$) and the reduction step (the computation of $c(x) = d(x) \bmod f(x)$) are integrated. Since for the proposed algorithm these steps are separated, we include Algorithm 2 for performing the reduction step. Assume that $f(x) = x^m + g(x)$, where the degree of polynomial $g(x)$ is less than $m - w$.

```
Algorithm 2: modular reduction.
INPUT: $a = (A_{n-1} \ldots A_{s-1} \ldots A_0)$, and $f = (F_{s-1} \ldots F_0)$.
OUTPUT: $c = (C_{s-1} \ldots C_0) = a \bmod f$
1.  for $i$ from $n-1$ downto $s$ do
        Set $d \leftarrow iw - m$
        Set $t \leftarrow A_i(x)x^d \cdot f(x) = \sum_{j=0}^{w-1} a_{iw+j}x^{d+j} \cdot f(x)$
        // $t = (T_i \ldots T_{i-s}0 \ldots 0)$, where $T_i = A_i$ //
        for $j$ from $i$ downto $i-s$ do
            Set $A_j \leftarrow A_j \oplus T_j$
2.  Set $t \leftarrow \sum_{j=0}^{sw-1-m} a_{m+j}x^j \cdot f(x)$
    // $t = (T_{s-1} \ldots T_0)$ //
3.  for $j$ from $s-1$ downto 0 do
        Set $A_j \leftarrow A_j \oplus T_j$
4.  return $(c \leftarrow (A_{s-1} \ldots A_0))$.
```

Algorithm 2 works by zeroing out the most significant word of $a(x)$ in each iteration of step 1. A chosen multiple of the reduction polynomial $f(x)$ is added to $a(x)$ which lowers the degree of $a(x)$ by w. This is possible because the degree of $g(x)$ is less than $m - w$. Finally, the leading $sw - m$ bits of A_{s-1} are cancelled in step 3 obtaining a polynomial of degree less than m. The number of XOR operations will depend on the weight of the reduction polynomial $f(x)$. For example, if $f(x)$ is a pentanomial then Algorithm 2 requires at most $8n$ XOR operations.

Remark 1. The use of standard programming tricks such as *separated name variables*, and *loop-unrolled code*, can be used to improve the performance of both Algorithms 1 and 2. See [10] for some suggested programming optimizations.

3 Proposed method

In this section we describe two versions of the new algorithm for multiplication in $\mathbb{F}_{2^m}$. The first version is a straightforward extension of Lim/Lee's method, which does not require extra temporary memory. The second version is based on a window technique. Before we describe the proposed algorithms, we discuss a simple version of Lim/Lee's method for exponentiation, using the terminology of additive groups; this will help us to understand the extension to $\mathbb{F}_{2^m}$.

In order to compute the "multiplication" $a \cdot g$ (the addition of g to itself a times) where a is an integer and g is an element of an additive group, the number a is divided into s words of size w. Then a can be written as

$$a = (A_{s-1} \ldots A_1 A_0) = \sum_{i=0}^{s-1} A_i 2^{wi},$$

where each $A_i, 0 \leq i < s$, has the binary representation $(a_{iw+w-1} \ldots a_{iw+1} a_{iw})_2$. Based on the binary representation $(u_{s-1} \ldots u_1 u_0)_2$ of u, $1 \leq u < 2^s$, and the group elements $2^{wi} \cdot g, 0 \leq i < s-1$, define the vector $P[u]$ of precomputations by the following equation:

$$P[u] = u_{s-1} 2^{w(s-1)} \cdot g + u_{s-2} 2^{w(s-2)} \cdot g + \cdots + u_1 2^w \cdot g + u_0 \cdot g.$$

Then the multiplication $a \cdot g = \sum_{i=0}^{s-1} A_i 2^{wi} \cdot g$, can be computed as

$$a \cdot g = \sum_{j=0}^{w-1} 2^j (\sum_{i=0}^{s-1} a_{iw+j} 2^{wi} \cdot g) = \sum_{j=0}^{w-1} 2^j P[I_j], \tag{2}$$

where $I_j = (a_{(s-1)w+j} \ldots a_{w+j} a_j)_2$. A detailed algorithm for computing $a \cdot g$ using the Lim/Lee's precomputation technique is given in Algorithm 3.

Algorithm 3: Lim/Lee's algorithm.

INPUT: $a = \sum_{i=0}^{s-1} A_i 2^{wi}, A_i = (a_{iw+w-1} \ldots a_{iw})_2, 0 \leq i < s$, and g.
OUTPUT: $r = a \cdot g$

```
// Precomputation //
1.  for $u$ from 0 downto $2^s - 1$ do
        Set $u \leftarrow (u_{s-1} \ldots u_1 u_0)_2$
        Set $P[u] \leftarrow \sum_{i=0}^{s-1} u_i 2^{wi} \cdot g$
// Main Computation //
2.  Set $r \leftarrow 0$
3.  for $j$ from $w-1$ downto 0 do
        Set $r \leftarrow r + r$
        Set $u \leftarrow (a_{(s-1)w+j} \ldots a_{w+j} a_j)_2$
        Set $r \leftarrow r + P[u]$
4.  return ($r$).
```

Algorithm 3 performs well in situations where the group element g is known in advance, since the calculation of the precomputation step can be made off-line. A faster version of this algorithm, with more precomputations, is discussed in [6].

Next we explain the extension of Algorithm 3 to the finite field $\mathbb{F}_{2^m}$. Let a and b be two polynomials in $\mathbb{F}_{2^m}$. Assume that a can be represented as $a = (A_{s-1} \ldots A_0)$. By replacing 2 by x and $2^w \cdot g$ by $x^w b(x)$ in (2), we obtain the following formal expression for the product $a(x)b(x)$:

$$a(x)b(x) = \sum_{j=0}^{w-1} x^j (\sum_{i=0}^{s-1} a_{iw+j} x^{wi}) b(x).$$

It is easy to verify that indeed the above formula for $a(x)b(x)$ is correct. Then an algorithm, analogue of Algorithm 3, can be derived for computing $ab \bmod f$ when b is a polynomial known in advance. By observing that the operation $x^{wi}b(x)$ is virtually free (it consists of an arrangement of the words representing b), the precomputation of the $2^s - 1$ polynomials: $P[u] = \sum_{i=0}^{s-1} u_i x^{wi}, 1 < u < 2^s, u = (u_{s-1} \dots u_0)_2$, can be made online. This eliminates the need of storing $2^s - 1$ polynomials, and the resulting algorithm is faster than Algorithm 1, even when b is not a fixed polynomial. The details of this method are given in Algorithm 4.

Algorithm 4: basic proposed method.

INPUT: $a = (A_{s-1} \dots A_0)$, $b = (B_{s-1} \dots B_0)$, and $f = (F_{s-1} \dots F_0)$.
OUTPUT: $c = (C_{s-1} \dots C_0) = ab \bmod f$

```
1.  Set $T_i \leftarrow 0$;  $i = 0, \dots, 2s-1$
2.  for $j$ from $w-1$ downto 0 do
       for $i$ from 0 to $s-1$ do
          if $a_{iw+j} \neq 0$ then
             for $k$ from 0 to $s-1$ do
                Set $T_{k+i} \leftarrow T_{k+i} \oplus B_k$
       if $j \neq 0$ then $T \leftarrow xT$ // shift $T$//
3.  Set $c \leftarrow T$ mod $f$ // Use Algorithm 2 //
4.  return ($c$).
```

The idea of window methods [4, pp. 66] for exponentiation can be extended to Algorithm 4 to obtain a more efficient algorithm, provided that extra temporary memory is available. For example, if we define the precomputed vector $P_{16}[u]$ for $0 \leq u < 16$, using the equation

$$P_{16}[u](x) = (u_3x^3 + u_2x^2 + u_1x + u_0)b(x),$$

where $u = (u_3 \dots u_0)_2$, then the product $a(x)b(x)$ can be computed as

$$\begin{aligned} a(x)b(x) &= \sum_{i=0}^{s-1} \sum_{j=0}^{w-1} a_{iw+j} x^{iw+j} b(x) \\ &= \sum_{j=0}^{w-1} x^j \sum_{i=0}^{s-1} a_{iw+j} x^{iw} b(x) \\ &= \sum_{j=0}^{w/4-1} x^{4j} \sum_{i=0}^{s-1} (a_{iw+j+3}x^3 + \cdots + a_{iw+j+1}x + a_{iw+j}) x^{iw} b(x) \end{aligned}$$

$$= \sum_{j=0}^{w/4-1} x^{4j} (\sum_{i=0}^{s-1} x^{wi} P_{16}[u_{i,j}](x)), \text{ where } u_{i,j} = (a_{iw+j+3} \dots a_{iw+j})_2.$$

Based on the above formula for ab, we derived an algorithm that processes simultaneously four bits of each word of a and trades in each iteration four multiplications by x for one multiplication by x^4. This method is described in Algorithm 5.

Algorithm 5: fast proposed method.

INPUT: $a = (A_{s-1} \dots A_0)$, $b = (B_{s-1} \dots B_0)$, and $f = (F_{s-1} \dots F_0)$.
OUTPUT: $c = (C_{s-1} \dots C_0) = ab \bmod f$.

```
1.  for $j$ from 0 to 15 do
        Set $P_{16}[j] \leftarrow (j_3x^3 + \cdots + j_0)b(x), j = (j_3j_2j_1j_0)_2$
2.  Set $T_i \leftarrow 0$;  $i = 0, \dots, 2s-1$
3.  for $j$ from $w/4-1$ downto 0 do
        for $i$ from 0 to $s-1$ do
            Set $u_{i,j} \leftarrow A_i/2^{4j} \bmod 16$
            for $k$ from 0 to $s-1$ do
                Set $T_{k+i} \leftarrow T_{k+i} \oplus P_{16}[u_{i,j}][k]$
        if $j \neq 0$ then $T \leftarrow x^4T$
4.  Set $c \leftarrow T$ mod $f$ // Use Algorithm 2 //
5.  return ($c$).
```

Remark 2. When b is known in advance, Algorithm 5 can be modified to work with a larger window size. If we process eight bits at the same time, then we need 256 field elements of precomputations. By observing that $\sum_{i=0}^{7} a_i x^i b(x) = \sum_{j=0}^{3} a_j x^j b(x) + \sum_{j=0}^{3} a_{4+j} x^j x^4 b(x)$, we reduce the precomputation to 32 field elements at the expense of doing more XOR operations.

3.1 Performance comparison

Let us compare the performance of Algorithms 4 and 5. We calculate the number of XOR operations and SHIFT operations required in each algorithm. We assume that the reduction polynomial is a pentanomial, so the total number of XOR operations required by Algorithm 2 is at most $8(2s-1)$. Therefore, Algorithm 4 requires $2(w-1)$ SHIFT operations and $sm/2 + 8(2s-1)$ XOR operations on average. Similarly, Algorithm 5 requires $3 + 2(w/4-1)$ SHIFT[2] operations and $s(11 + m/4) + 8(2s-1)$ XOR operations on average. Thus, the time saved in Algorithm 5 is at the expense of using 16 field elements of temporary memory. In Table 1 we compared the number of operations required by Algorithms 1, 4 and 5, for the particular case $m = 163$, $w = 32$, $s = 6$, and the pentanomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

[2]We are assuming that multiplying a polynomial by x^4 is comparable in speed to multiplying a polynomial by x.

Algorithms	XOR	SHIFT
Algorithm 1	81*6+ 81*2 = 648	162
Algorithm 4	81*6 + 42 = 528	62
Algorithm 5	52*6 + 42 = 354	17

Table 1: Number of operations for Algorithms 1, 4 and 5.

4 Timing results

This section presents running timings for the proposed algorithms and the "shift-and-add" method on the following platforms: a 233 MHz Pentium MMX, a 400 MHz Pentium II, a 450 MHz Sun UltraSparc workstation and a 10 MHz Intel 386 processor (RIM interactive pager [3]). The implementation was written entirely in C, and the compilers used were `gcc` for the workstation Sun and the Pentium MMX, and Microsoft Visual C++ (version 6.0) for the other architectures. All algorithms were implemented with a comparable level of programming optimizations.

Tables 2 and 3 show timings to perform a multiplication in $\mathbb{F}_{2^{163}}$ using Algorithms 1, 4 and 5.[3] From Table 2, Algorithm 4 performs 45% to 49% faster than Algorithm 1, and the best speed up was obtained on the UltraSparc machine. In Table 3 the performances of the fast version of the proposed algorithm (Algorithm 5) and the standard method are compared. We observed a significant improvement: Algorithm 5 is about 3.0 to 5.5 times faster than the standard method.

	Pentium 233 MHz	UltraSparc 450 MHz
Algorithm 1	31.27	10.97
Algorithm 4	17.07	5.55

Table 2: Timings (in microseconds) of the "shift-and-add" method and Algorithm 4 for multiplication in $\mathbb{F}_{2^{163}}$.

4.1 Applications

The most important application of this work is in software implementations of elliptic curve cryptography over $\mathbb{F}_{2^m}$. Our timings on different architectures have shown that Algorithm 5 is significantly faster than the standard method in modern workstations as well as in wireless devices such as the RIM pager (a hand-held device with an Intel processor running at 10 MHz [3]).

[3]Recently, NIST has recommended elliptic curves over $\mathbb{F}_{2^{163}}$ for US federal government use [9].

	RIM 10 MHz	Pentium 233 MHz	Pentium II 400 MHz	UltraSparc 450 MHz
Algorithm 1	4,848	31.27	16.48	10.97
Algorithm 5	1,515	10.20	2.97	2.52

Table 3: Timings (in microseconds) of the "shift-and-add" method and Algorithm 5 for multiplication in $\mathbb{F}_{2^{163}}$.

5 Conclusions

There are several techniques that can be used for speeding up the computation of $c = ab \bmod f$ in $\mathbb{F}_{2^m}$. In this paper we have shown a technique based on Lim/Lee's method for exponentiations. It turns out that our software implementation of the optimized version (Algorithm 5), on different platforms, proved to be significantly faster than the "shift-and-add" method, making it useful for software implementations of elliptic curve cryptography in different computational environments.

6 Acknowledgments

The authors wish to thank Guido Araújo, Cláudio Lucchesi, Alfred Menezes, Daniel Panario and Routo Terada for many helpful comments and suggestions.

References

[1] G. B. Agnew, R. C. Mullin and S. A. Vanstone, "An implementation of elliptic curve cryptosystems over $\mathbb{F}_{2^{155}}$", *IEEE journal on selected areas in communications*, **11**, pp. 804-813, 1993.

[2] ANSI X9.62, "The elliptic curve digital signature algorithm (ECDSA)", American Bankers Association, 1999.

[3] Blackberry, `http://www.blackberry.net`

[4] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[5] C. K. Koç and T. Acar, "Montgomery multiplication in $GF(2^k)$", *Designs, Codes and Cryptography*, **14**, pp. 57-69, 1998.

[6] C. H. Lim and P. J. Lee, "More flexible exponentiation with precomputation", In *Advances in Cryptography-CRYPTO'94*, pp. 95-107, Springer-Verlag, 1994.

[7] A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

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

[9] National Institute of Standards and Technology, "Digital signature standard", FIPS Publication 186-2, February 2000. Available at `http://csrc.nist.gov/fips`

[10] R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast key exchange with elliptic curve systems", University of Arizona, C. S., Tech. report 95-03, 1995.

[11] E. De Win, A. Bosselaers, S. Vanderberghe, P. De Gersem and J. Vandewalle, "A fast software implementation for arithmetic operations in $GF(2^n)$," *Advances in Cryptology, Proc. Asiacrypt'96*, LNCS **1163**, pp. 65-76, Springer-Verlag, 1996.