Fast Software Exponentiation in GF(2^k) *

Ç. K. Koç and T. Acar
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon, 97331, USA
E-mail: {koc,acar}@ece.orst.edu

Abstract

We present a new algorithm for computing a^e where $a \in$ GF(2^k) and e is a positive integer. The proposed algorithm is more suitable for implementation in software, and relies on the Montgomery multiplication in GF(2^k). *The speed of the exponentiation algorithm largely depends on the availability of a fast method for multiplying two polynomials of length w defined over* GF(2). *The theoretical analysis and our experiments indicate that the proposed exponentiation method is at least 6 times faster than the exponentiation method using the standard multiplication when $w = 8$. Furthermore, the availability of a 32-bit* GF(2) *polynomial multiplication instruction on the underlying processor would make the new exponentiation algorithm up to 37 times faster.*

1. Introduction

The arithmetic operations in the Galois field GF(2^k) have several applications in coding theory, computer algebra, and cryptography. We are especially interested in cryptographic applications where k is very large. Examples of the cryptographic applications are the Diffie-Hellman key exchange algorithm [3] based on the discrete exponentiation and the elliptic curve cryptosystems [8, 14] over the field GF(2^k). The Diffie-Hellman algorithm requires computation of g^e, where g is a fixed primitive element of the field and e is an integer. In elliptic curves, the exponentiation operation is used to compute inverse of an element in GF(2^k), based on Fermat's identity $a^{-1} = a^{2^k-2}$ [6, 1]. In general, an arbitrary integer power of an element $a \in$ GF(2^k) can be computed using the binary method [7] which breaks the exponentiation operation into a series of squaring and multiplication operations in GF(2^k).

We present a new method for fast software implementation of the exponentiation operation in GF(2^k). The method is based on the Montgomery multiplication algorithm introduced in [9], and is applicable to any exponentiation method, whether it is the binary method or the m-ary method or any of the more advanced methods. The proposed method simply replaces the standard squaring and multiplication operations in GF(2^k) with the Montgomery squaring and multiplication operations. A small amount of pre- and postprocessing is also performed.

2. Montgomery Exponentiation

Let r be a special fixed element of GF(2^k). The selection of r will be made clear in the sequel. Also let a be an arbitrary element of GF(2^k). The Montgomery image of a under r is denoted as $\bar{a}$, and is defined as

$$\bar{a} = a \cdot r \ . \tag{1}$$

Given two Montgomery images $\bar{a}$ and $\bar{b}$, their Montgomery product '$\times$' is defined as

$$\bar{a} \times \bar{b} = \bar{a} \cdot \bar{b} \cdot r^{-1} \ . \tag{2}$$

The Montgomery product of $\bar{a}$ and $\bar{b}$ is equal to the Montgomery image of the product $a \cdot b$. This is easily proved as

$$\bar{c} = \bar{a} \times \bar{b} = \bar{a} \cdot \bar{b} \cdot r^{-1} = (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} = a \cdot b \cdot r = c \cdot r \ .$$

The proposed exponentiation algorithm is based on this property. Let e be an m-bit integer, where $e_i \in \{0, 1\}$ is the ith bit of e for $i = 0, 1, \ldots, m-1$. In order to compute $c = a^e$ for a given $a \in$ GF(2^k), we first compute the Montgomery images of 1 and a using standard multiplications. The exponentiation algorithm based on the binary method then proceeds to compute c using only the Montgomery squaring and multiplication operations.

*This research is supported in part by Intel Corporation.

1063-6889/97 $10.00 © 1997 IEEE

Algorithm for Montgomery Exponentiation

Step 1. $\bar{c} := 1 \cdot r$
Step 2. $\bar{a} := a \cdot r$
Step 3. for $i = m - 1$ downto 0 do
Step 4. $\quad \bar{c} := \bar{c} \times \bar{c}$
Step 5. $\quad$ if $e_i = 1$ then $\bar{c} := \bar{c} \times \bar{a}$
Step 6. $c := \bar{c} \times 1$

The difference of the above algorithm from the binary method using the standard squaring and multiplication operations is that in Steps 4 and 5, respectively, we perform the Montgomery squaring and multiplication operations. Initially, we have $\bar{c} = 1 \cdot r$. When a Montgomery squaring (or multiplication) is performed, the multiplicative factor r remains in place since

$$\bar{c} \times \bar{c} = (c \cdot r) \cdot (c \cdot r) \cdot r^{-1} = (c \cdot c) \cdot r \ , \quad (3)$$

$$\bar{c} \times \bar{a} = (c \cdot r) \cdot (a \cdot r) \cdot r^{-1} = (c \cdot a) \cdot r \ . \quad (4)$$

We remove this multiplicative factor on $\bar{c}$ in Step 6 by performing a Montgomery multiplication as

$$\bar{c} \times 1 = (c \cdot r) \cdot 1 \cdot r^{-1} = c \ . \quad (5)$$

In order to perform the Montgomery squaring and multiplication operations, we use the algorithm introduced in [9]. This method is based on the polynomial representation of the elements of GF(2^k), and is particularly suitable for software implementation due to the fact that it proceeds in a word-level fashion.

3. Montgomery Multiplication

The elements of the field GF(2^k) can be represented in several different ways [12, 13, 11]. The Montgomery multiplication is based on the polynomial representation. According to this representation an element a of GF(2^k) is a polynomial of length k, written as $a(x) = \sum_{i=0}^{k-1} a_i x^i$. The coefficients $a_i \in$ GF(2) are often referred to as the bits of a, and the element a is also written as $a = (a_{k-1} a_{k-2} \cdots a_1 a_0)$.

In the word-level description of the algorithms, we partition these bits into blocks of w bits, where w is the wordsize of the computer. We assume that $k = sw$, and write a as an sw-bit number consisting of s blocks, where each block is of length w. If k is less than sw (and more than $(s-1)w$), then we pad the high-order bits of the most significant block with zero bits and take k as sw. Thus, we write a as $a = (A_{s-1} A_{s-2} \cdots A_1 A_0)$, where each A_i is of length w.

The addition of two elements a and b in GF(2^k) are performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in the field GF(2). This is equivalent to the bit-wise XOR operation on the vectors a and b. On the other hand, we need an irreducible polynomial of degree k in order to multiply two elements a and b in GF(2^k). Let $n(x)$ be an irreducible polynomial of degree k over the field GF(2). The product $c = a \cdot b$ in GF(2^k) is obtained by computing

$$c(x) = a(x)b(x) \bmod n(x) \ , \quad (6)$$

where $c(x)$ is a polynomial of length k, representing the element $c \in$ GF(2^k). Thus, the multiplication operation in the field GF(2^k) is accomplished by multiplying the corresponding polynomials modulo the irreducible polynomial $n(x)$.

The Montgomery product is defined as $a \cdot b \cdot r^{-1}$, where r is a special fixed element of GF(2^k). The selection of $r(x) = x^k$ turns out to be very useful in obtaining fast software implementations. Thus, r is the element of the field, represented by the polynomial $r(x) \bmod n(x)$. The Montgomery multiplication method requires that $r(x)$ and $n(x)$ be relatively prime. For this assumption to hold, it suffices that $n(x)$ be not divisible by x. Since $n(x)$ is an irreducible polynomial over the field GF(2), this will always be case. Since $r(x)$ and $n(x)$ are relatively prime, there exist two polynomials $r^{-1}(x)$ and $n'(x)$ with the property that

$$r(x)r^{-1}(x) + n(x)n'(x) = 1 \ , \quad (7)$$

where $r^{-1}(x)$ is the inverse of $r(x)$ modulo $n(x)$. The polynomials $r^{-1}(x)$ and $n'(x)$ can be computed using the extended Euclidean algorithm [11, 12]. The Montgomery product of a and b is defined as

$$c(x) = a(x)b(x)r^{-1}(x) \bmod n(x) \ , \quad (8)$$

which can be computed in 3 steps using:

Step 1. $t(x) := a(x)b(x)$
Step 2. $u(x) := t(x)n'(x) \bmod r(x)$
Step 3. $c(x) := [t(x) + u(x)n(x)]/r(x)$

The computation of $c(x)$ involves standard multiplications, a modulo $r(x)$ multiplication, and a division by $r(x)$. The modular multiplication and division operations in Steps 2 and 3 are intrinsically fast operations since $r(x) = x^k$. The remainder operation in modular multiplication using the modulus x^k is accomplished by simply ignoring the terms which have powers of x larger than and equal to k. Similarly, division of an arbitrary polynomial by x^k is accomplished by shifting the polynomial to the right by k places. A drawback in computing $c(x)$ is the precomputation of $n'(x)$ required in Step 2. However, it turns out that the computation of $n'(x)$ can be completely avoided if the coefficients of $a(x)$ are scanned one bit at a time. Furthermore, the word-level algorithm requires the computation of only

the least significant word $N'_0(x)$ instead of the whole $n'(x)$. In order to explain this, we note that the Montgomery product can be written as

$$c(x) = x^{-k}a(x)b(x) = x^{-k}\sum_{i=0}^{k-1} a_i x^i b(x) \pmod{n(x)}.$$

The product

$$t(x) = (a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0)b(x)$$

can be computed by starting from the most significant digit, and then proceeding to the least significant as

$$\begin{array}{l} t(x) := 0 \\ \text{for } i = k-1 \text{ to } 0 \\ \quad t(x) := t(x) + a_i b(x) \\ \quad t(x) := xt(x) \end{array}$$

The shift factor x^{-k} in $x^{-k}a(x)b(x)$ reverses the direction of summation. Since

$$\begin{aligned} &x^{-k}(a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0) = \\ &\quad a_{k-1}x^{-1} + a_{k-2}x^{-2} + \cdots + a_1 x^{-k+1} + a_0 x^{-k} \ , \end{aligned}$$

we start processing the coefficients of $a(x)$ from the least significant, and obtain the following bit-level algorithm in order to compute $t(x) = a(x)b(x)x^{-k}$.

$$\begin{array}{l} t(x) := 0 \\ \text{for } i = 0 \text{ to } k-1 \\ \quad t(x) := t(x) + a_i b(x) \\ \quad t(x) := t(x)/x \end{array}$$

The above algorithm computes the product $t(x) = x^{-k}a(x)b(x)$, however, we are interested in computing $c(x) = x^{-k}a(x)b(x) \bmod n(x)$. Following the analogy to the integer algorithm, we achieve this computation by adding $n(x)$ to $c(x)$ if c_0 is 1, making the new $c(x)$ divisible by x since $n_0 = 1$. If c_0 is already 0 after the addition step, we do not add $n(x)$ to it. Therefore, we are computing $c(x) := c(x) + c_0 n(x)$ after the addition step. After this computation, $c(x)$ will always be divisible by x. We can compute $c(x) := c(x)x^{-1} \bmod n(x)$ by dividing $c(x)$ by x since $c(x) = xu(x)$ implies $c(x) = xu(x)x^{-1} = u(x) \bmod n(x)$. The bit-level algorithm is given below:

Bit-Level Algorithm for Montgomery Multiplication

Step 1.	$c(x) := 0$
Step 2.	for $i = 0$ to $k-1$ do
Step 3.	$\quad c(x) := c(x) + a_i b(x)$
Step 4.	$\quad c(x) := c(x) + c_0 n(x)$
Step 5.	$\quad c(x) := c(x)/x$

The bit-level algorithm for the Montgomery multiplication given above is generalized to the word-level algorithm by proceeding word by word, where the word-size is $w \geq 2$ and $k = sw$. Recall that $A_i(x)$ represents the ith word of the polynomial $a(x)$. The addition step is performed by multiplying $A_i(x)$ by $b(x)$ for $i = 0, 1, \ldots, s-1$. We then need to multiply the partial product $c(x)$ by x^{-w} modulo $n(x)$. In order to perform this step using division, we add a multiple of $n(x)$ to $c(x)$ so that the least significant w coefficients of $c(x)$ will be zero, i.e., $c(x)$ will be divisible by x^w. Thus, if $c(x) \neq 0 \bmod x^w$, then we find $M(x)$ (which is a polynomial of length w) such that $c(x) + M(x)n(x) = 0 \pmod{x^w}$. Let $C_0(x)$ and $N_0(x)$ be the least significant words of $c(x)$ and $n(x)$, respectively. We calculate $M(x)$ as

$$M(x) = C_0(x)N_0^{-1}(x) \bmod x^w \ .$$

We note that $N_0^{-1}(x) \bmod x^w$ is equal to $N'_0(x)$ since the property (7) implies that

$$\begin{aligned} x^{sw}x^{-sw} + n(x)n'(x) &= 1 \pmod{x^w} \\ N_0(x)N'_0(x) &= 1 \pmod{x^w} \end{aligned}$$

The word-level algorithm for the Montgomery multiplication is obtained as

Word-Level Algorithm for Montgomery Multiplication

Step 1.	$c(x) := 0$
Step 2.	for $i = 0$ to $s-1$ do
Step 3.	$\quad c(x) := c(x) + A_i(x)b(x)$
Step 4.	$\quad M(x) := C_0(x)N'_0(x) \pmod{x^w}$
Step 5.	$\quad c(x) := c(x) + M(x)n(x)$
Step 6.	$\quad c(x) := c(x)/x^w$

4. Montgomery Squaring

The computation of the Montgomery squaring can be optimized due to the fact that cross terms disappear because they come in pairs and the ground field is GF(2). Therefore, we skip the multiplication steps, and obtain

$$\begin{aligned} c(x) &= a^2(x) \\ &= a_{k-1}x^{2(k-1)} + a_{k-2}x^{2(k-2)} + \cdots + a_1 x^2 + a_0 \\ &= (a_{k-1}0a_{k-2}0\cdots 0a_1 0a_0) \ . \end{aligned}$$

The Montgomery squaring algorithm starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$, and then reduces $c(x)$ by computing $c(x) := c(x)x^{-k} \bmod n(x)$. The steps of the word-level algorithm are given below:

Word-Level Algorithm for Montgomery Squaring
Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2. for $i = 0$ to $s-1$ do
Step 3. $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 4. $c(x) := c(x) + M(x)n(x)$
Step 5. $c(x) := c(x)/x^w$

5. Computation of the Inverse

The word-level algorithm requires the computation of the w-length polynomial $N_0'(x)$ instead of the entire polynomial $n'(x)$ which is of length $k = sw$. It turns out that the algorithm introduced in [4] for computing n_0' in the integer case can also be generalized to the polynomial case. The inversion algorithm is based on the observation that the polynomial $N_0(x)$ and its inverse satisfy

$$N_0(x)N_0^{-1}(x) = 1 \pmod{x^i} \qquad (9)$$

for $i = 1, 2, \ldots, w$. In order to compute $N_0'(x)$, we start with $N_0'(x) = 1$, and proceed as

Algorithm for Inversion
Step 1. $N_0'(x) := 1$
Step 2. for $i = 2$ to w
Step 3. $t(x) := N_0(x)N_0'(x) \bmod x^i$
Step 4. if $t(x) \neq 1$ then $N_0'(x) := N_0'(x) + x^{i-1}$

6. Analysis

In this section, we give a rigorous analysis of the Montgomery exponentiation algorithm in GF(2^k) by calculating the number of word-level operations. The word-level GF(2) polynomial addition is simply the bitwise XOR operation which is a readily available instruction on most general purpose microprocessors and signal processors. The word-level GF(2) polynomial multiplication operation receives two 1-word (w-bit) polynomials $a(x)$ and $b(x)$ defined over the field GF(2), and computes the 2-word ($2w$-bit) polynomial $c(x) = a(x)b(x)$. For example, given $a = (1101)$ and $b = (1010)$, this operation computes c as

$$\begin{aligned} a(x)b(x) &= (x^3 + x^2 + 1)(x^3 + x) \\ &= x^6 + x^5 + x^4 + x \\ &= (0111\ 0010)\,. \end{aligned}$$

The implementation of this operation, which we call MULGF2, can be performed in 3 distinctly different ways:

- An instruction on the processor.
- Table lookup approach.
- Emulation using SHIFT and XOR operations.

In the first approach, the processor has a special MULGF2 instruction as defined above. The availability of an instruction to perform this operation would definitely be the fastest method. However, none of the general purpose processors contains an instruction to perform this operation.

A simple method for implementing the table lookup approach is to use two tables, one for computing the higher (H) and the other for computing the lower (L) bits of the product. The tables are addressed using the bits of the operands, and thus, each table is of size $2^w \times 2^w \times w$ bits. We obtain the values H and L in two table read operations. However, we note that these tables are different from the tables in [5, 2], which are used to implement GF(2^w) multiplications. Here we are using the tables to multiply two $(w-1)$-degree polynomials over GF(2) to obtain the polynomial of degree $2(w-1)$.

In the emulation approach, two w-bit polynomials A and B are multiplied using shift, rotate, and xor operations. The 2-word product is accumulated in two words H and L as follows:

```
H := 0
L := 0
for j=w-1 downto 0 do
    L := SHL(L,1)
    H := RCL(H,1)
    if BIT(B,j)=1 then L := L XOR A
```

Here SHL and RCL correspond to the left shift and left rotate with carry instructions. This algorithm computes the 2-word result using a total of $2w$ SHIFT (or ROTATE) and w XOR operations. The emulation approach is usually slower than the table lookup approach, particularly for $w \geq 8$.

In order to compare the exponentiation algorithms using the standard and the Montgomery multiplications, we count the number of word-level operations required by these algorithms. We perform this analysis by fixing the exponentiation method as the binary method, and taking m as the number of bits in the exponent e. In our analysis, we do not consider certain processor features, e.g., special bit-level instructions (test jth bit), conditional executions (delay slots in conditional branches), and conditional data movement instructions. Also, loop count loop overhead, pointer arithmetic, etc., are ignored.

The standard and Montgomery exponentiation algorithms are given in Figure 1 below. The Montgomery exponentiation algorithm relies on the subroutines for computing the inverse, the Montgomery squaring and

multiplications (in Steps 4 and 5), and a single standard multiplication (in Step 2). We do not need to perform a multiplication in Step 1. The standard exponentiation algorithm, on the other hand, requires only the standard squaring and multiplication subroutines.

Figure 1. Exponentiation Algorithms.

Montgomery Exponentiation

Step 1. $\bar{c} := 1 \cdot r$
Step 2. $\bar{a} := a \cdot r$
Step 3. for $i = m - 1$ downto 0 do
Step 4. $\quad \bar{c} := \bar{c} \cdot \bar{c} \cdot r^{-1}$
Step 5. $\quad$ if $e_i = 1$ then $\bar{c} := \bar{c} \cdot \bar{a} \cdot r^{-1}$
Step 6. $c = \bar{c} \cdot 1 \cdot r^{-1}$

Standard Exponentiation

Step 1. $c := 1$
Step 2. for $i = m - 1$ downto 0 do
Step 3. $\quad c := c \cdot c$
Step 4. $\quad$ if $e_i = 1$ then $c := c \cdot a$

The detailed analyses of the word-level Montgomery and standard multiplication algorithms are given in [9]. Similar analyses can also be given for the word-level Montgomery and standard squaring algorithms. The number of word-level operations required by these algorithms are summarized in Table 1.

Table 1. Operation counts for the multiplication and squaring algorithms.

Operation	Montgomery MUL	Montgomery SQU
MULGF2	$2s^2 + s$	$s^2 + s$
XOR/AND	$4s^2$	$2s^2 + (2w+1)s$
SHIFT	-	$(2w+1)s$
Operation	**Standard MUL**	**Standard SQU**
MULGF2	s^2	-
XOR/AND	$(\frac{3w}{2} + 3)s^2 + \frac{w}{2}s$	$\frac{9w}{4}s^2 + (2w + \frac{3}{2})s$
SHIFT	$2(w+1)s^2 + (w+1)s$	$3ws^2 + (3w+1)s$

On the other hand, the inversion algorithm given in Section 5 requires $(w-1)$ MULGF2, $(w-1)$ AND, and $(w-1)$ SHIFT operations in Step 3. Assuming the least significant coefficient of $t(x)$ is equal 0 with probability 1/2, we obtain the number of XOR and SHIFT operations in Step 4 as $(w-1)/2$ and $(w-1)/2$, respectively. Using these values and Table 1, we summarize the operation counts of the exponentiation algorithms in Table 2.

Table 2. Operation counts for the Montgomery and the standard exponentiation.

Montgomery EXP		Standard EXP	
Step	MULGF2	Step	MULGF2
Inv	$w-1$	3	-
2	s^2	4	$\frac{m}{2}s^2$
4	$ms^2 + ms$		
5	$ms^2 + \frac{m}{2}s$		
6	$2s^2 + s$		

Montgomery Exponentiation		
Step	XOR/AND/OR	SHIFT
Inv	$\frac{3(w-1)}{2}$	$\frac{3(w-1)}{2}$
2	$3(\frac{w}{2}+1)s^2 + \frac{w}{2}s$	$2(w+1)s^2 + (w+1)s$
4	$2ms^2 + (2w+1)ms$	$(2w+1)ms$
5	$2ms^2$	-
6	$4s^2$	-
Standard Exponentiation		
3	$\frac{9w}{4}ms^2 + (2w + \frac{3}{2})ms$	$3wms^2 + (3w+1)ms$
4	$(\frac{3w}{4} + \frac{3}{2})ms^2 + \frac{w}{4}ms$	$(w+1)ms^2 + \frac{w+1}{2}ms$

In Table 3, we summarize the total number of operations required by the Montgomery and standard exponentiation algorithms for $w = 8, 16, 32$. Table 4 gives the maximum speedup of the proposed exponentiation method assuming the word-level operations XOR/AND/OR and SHIFT take nearly the same amount of time. The emulation cost of MULGF2 is $2w$ SHIFT and w XOR operations in the emulation case. The cost of MULGF2 instruction is assumed equal to those of SHIFT and XOR operations in the instruction case.

7. Implementation Results

We have implemented the Montgomery and standard exponentiation algorithms in C, and obtained timings on a 100-MHz Intel 486DX4 processor running the NextStep 3.3 operating system. We executed the exponentiation programs several hundred times and obtained the average timings for each k. The modulus polynomial $n(x)$ is generated randomly for $k = 64$, 128, 256, 512, 1024, 1536, 2048. The exponent is an m-bit integer with equal number of 0 and 1 bits.

The multiplication operation MULGF2 was implemented using three approaches. In the first approach,

Table 3. Comparing the Montgomery and standard exponentiation algorithms.

MULGF2	w	Standard
Emulation	8	$70.5ms^2 + 49ms$
"	16	$138.5ms^2 + 95ms$
"	32	$274.5ms^2 + 187ms$
Instruction	8	$59ms^2 + 49ms$
"	16	$115ms^2 + 95ms$
"	32	$227ms^2 + 187ms$
MULGF2	w	**Montgomery**
Emulation	8	$(52m + 109)s^2 + (70m + 37)s + 189$
"	16	$(100m + 209)s^2 + (138m + 73)s + 765$
"	32	$(196m + 409)s^2 + (274m + 145)s + 3069$
Instruction	8	$(6m + 40)s^2 + (35.5m + 14)s + 28$
"	16	$(6m + 68)s^2 + (67.5m + 26)s + 60$
"	32	$(6m + 124)s^2 + (131.5m + 50)s + 124$

Table 4. Estimated speedup values of Montgomery exponentiation.

MULGF2→	Emulation			Instruction		
w	8	16	32	8	16	32
Speedup	1.36	1.39	1.40	9.83	19.17	37.83

we used the emulation algorithm given in the previous section.

In the second approach, a lookup table is used for $w = 8$, as described before. For $w = 8$, each of the tables is of size 64 Kilobytes, which is reasonable. However, for $w = 16$, the table size increases to $2^{16} \times 2^{16} \times 16$ bits, which gives 8 Gigabytes. Therefore, we have implemented the table lookup MULGF2 operation only for $w = 8$.

For $w = 16$ and $w = 32$, we implement the MULGF2 operation using a *hybrid approach*: 8-bit tables coupled with emulation to obtain the 16-bit or 32-bit result. For example, 16-bit multiplication using two 8-bit tables is computed as shown below.

```
a1 := SHR(a,8)
a0 := a AND 0xff
b1 := SHR(b,8)
b0 := b AND 0xff
L := TableL[a0][b0] XOR SHR(TableH[a0][b0]
     XOR TableL[a1][b0]
     XOR TableL[a0][b1],8)
H := TableH[a1][b0] XOR TableH[a0][b1]
     XOR TableL[a1][b1]
     XOR SHR(TableH[a1][b1],8)
```

where TableL and TableH are the 8-bit tables giving the low and high order 8-bits of an 8-by-8 bit GF(2) polynomial multiplication. The 32-bit hybrid multiplication algorithm also uses these 8-bit tables.

The experimental speedup values obtained are given in Table 5. These speedup values are obtained by dividing the time elapsed for standard exponentiation by the time elapsed for Montgomery exponentiation. Montgomery exponentiation time includes computation of $N_0'(x)$, precomputation of $\bar{a}$ and $\bar{c}$, and final computation by 1 to obtain c.

Table 5. Experimental speedup values of Montgomery exponentiation for $m = 128$.

$w \rightarrow$	8		16		32	
k	Tab8	Emu	Hyb8	Emu	Hyb8	Emu
64	6.32	4.10	6.75	5.00	5.33	3.61
128	4.85	3.79	4.51	4.20	4.00	3.25
256	4.95	3.49	4.40	3.60	3.03	2.79
512	5.66	3.83	3.96	3.35	2.88	2.66
1024	5.97	4.04	4.22	3.70	2.83	2.44
1536	6.00	3.95	4.58	3.51	2.69	2.44
2048	6.05	3.76	4.62	3.89	2.56	2.30

8. Conclusions

As the theoretical results summarized in Tables 3 and 4 the experimental data in Table 5 indicate, the Montgomery exponentiation algorithm is about 6 times faster than the standard exponentiation for $w = 8$. The table lookup approach for $w \geq 16$ seems unrealistic due to the size of the tables. An efficient way to implement the MULGF2 operation is to add an instruction to the processor to perform this multiplication. The availability of such an instruction would yield more speedup than the table lookup approach, because memory accesses would be eliminated which are required in the table lookup approach. For example, the availability of a 32-bit MULGF2 instruction would make the Montgomery exponentiation about 37 times faster than the standard exponentiation, as seen in Table 4.

The crucial part of the proposed exponentiation algorithm is the Montgomery multiplication in GF(2^k) introduced in [9]. The computation of the Montgomery multiplication in GF(2^k) is similar to the one for modular arithmetic. A review of the Montgomery multiplication algorithms for modular arithmetic is given in [10]. We are currently analyzing these algorithms, and comparing their time and space requirements for performing the Montgomery multiplication operation in GF(2^k). Another possible avenue of research is to compare the proposed exponentiation method to the one which uses trinomials and the normal basis. The squaring operation in the normal basis is trivial, however, the software implementation of the multiplication is more complicated.

References

[1] G. B. Agnew, T. Beth, R. C. Mullin, and S. A. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptology*, 6(1):3–13, 1993.

[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 *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. New York, NY: Springer-Verlag, 1996.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, Nov. 1976.

[4] S. R. Dussé and B. S. Kaliski, Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. New York, NY: Springer-Verlag, 1990.

[5] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 163–173. New York, NY: Springer-Verlag, 1992.

[6] T. Itoh, O. Teachai, and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF2^t using normal bases. *J. Soc. Electron. Comm. (Japan)*, 44:31–36, 1986.

[7] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[8] N. Koblitz. *A Course in Number Theory and Cryptography*. New York, NY: Springer-Verlag, Second edition, 1994.

[9] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). In *Proceedings of Third Annual Workshop on Selected Areas in Cryptography*, pages 95–106, Queen's University, Kingston, Ontario, Canada, August 15–16 1996.

[10] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[11] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. New York, NY: Cambridge University Press, 1994.

[12] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Boston, MA: Kluwer Academic Publishers, 1987.

[13] A. J. Menezes, editor. *Applications of Finite Fields*. Boston, MA: Kluwer Academic Publishers, 1993.

[14] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Boston, MA: Kluwer Academic Publishers, 1993.