A Reduction Method for Multiplication in Finite Fields

Ç. K. Koç and A. Halbutoğulları
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Technical Report, August 1998

Abstract

We propose a new table lookup based reduction method for performing the modular reduction operation, which can be used to obtain fast software implementations of the finite field multiplication and squaring operations. The reduction algorithm has both left-to-right and right-to-left versions, which respectively improve the standard and Montgomery multiplication methods in $GF(2^k)$. We also treat in detail the case in which special irreducible polynomials, e.g., the trinomials and all-one-polynomials, are used to generate the field. Furthermore, we show that the proposed reduction method also works in the integer case for the right-to-left reduction algorithm if the modulus n is an odd number. However, the table lookup based left-to-right reduction algorithm in the integer case is generally inefficient.

Key Words: Elliptic curve cryptography, finite field arithmetic, table lookup.

1 Introduction

Recently, there has been a growing interest to develop software methods for implementing $GF(2^k)$ arithmetic operations, which find applications in elliptic curve cryptographic operations [6, 7, 8] and the Diffie-Hellman key exchange algorithm [1] based on the discrete exponentiation. The software implementation of the arithmetic operations in $GF(2^k)$ require that we design *word-level* algorithms for efficient implementation on current general purpose microprocessors. In this paper, we propose a new table lookup based reduction method for performing the modular reduction operation, which can be used to obtain fast software implementations of the finite field multiplication and squaring operations. The proposed modular reduction method has three important properties:

- The method works for an arbitrary generating polynomial $n(x)$. It does not assume any special structure in $n(x)$, for example, the generating polynomial need not be a trinomial, a sparse polynomial, or an all-one-polynomial. However, in such cases, the method simplifies to certain modular reduction methods existing in the literature.
- The method provides word-level algorithms, enabling efficient software implementations of finite field multiplication and squaring operations.
- The method has both left-to-right and right-to-left versions, which are useful for performing both the standard and the Montgomery multiplications in finite fields.

Furthermore, it turns out that the table lookup reduction method is also applicable to the integer case for the right-to-left reduction algorithm, giving a more efficient Montgomery multiplication algorithm. The left-to-right reduction algorithm in the integer case turns out to be inefficient.

2 Representation of Field Elements

The elements of the field $GF(2^k)$ can be represented in several different ways [4, 5, 3]. The polynomial representation seems to be useful and suitable for both hardware and software implementations. According to this representation an element a of $GF(2^k)$ is a polynomial of length k, i.e., of degree less than or equal to $k-1$, written as $a(x) = \sum_{i=0}^{k-1} a_i x^i$, where the coefficients $a_i \in GF(2)$. The element a is also represented as a k-dimensional vector $a = (a_{k-1}a_{k-2}\cdots a_1 a_0)$. Software implementations are often based on the word-level representations of the field elements. We assume that $k = sw$, and partition the sw-dimensional into w-bit blocks A_i such that $a = (A_{s-1}A_{s-2}\cdots A_1 A_0)$. These A_i blocks are also represented as polynomials of degree $w-1$ using the notation $A_i(x)$.

The addition of two elements a and b in $GF(2^k)$ is performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in $GF(2)$. This is equivalent to the bit-wise XOR operation on the vectors a and b. Assuming the processor can perform the XOR of two w-bit numbers in one cycle, the computation of $c := a + b$ requires s cycles.

In order to multiply two elements a and b in $GF(2^k)$, we need an irreducible polynomial $n(x)$ of degree k over the field $GF(2)$. The product $c = a \cdot b$ in $GF(2^k)$ is obtained by computing $c(x) = a(x)b(x) \bmod n(x)$, 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 first multiplying the input polynomials, and then performing a modular reduction using the generating polynomial $n(x)$.

On the other hand, the Montgomery product of two elements $a(x)$ and $b(x)$ is defined as the computation of $c(x) = a(x)b(x)r^{-1}(x) \bmod n(x)$, where $r(x)$ is a fixed element of the field [2]. It is established that when $r(x) = x^k \bmod n(x)$, we can obtain efficient software implementations of the Montgomery multiplication operation in $GF(2^k)$. In the sequel, we define the Montgomery multiplication in $GF(2^k)$ as the computation of $c(x) = x^{-k}a(x)b(x) \bmod n(x)$, given $a(x)$ and $b(x)$.

3 Reduction Algorithms

The proposed modular reduction methodology uses a table of the multiples of the generating polynomial $n(x)$, and performs a *word-level division*. We start with $n(x)$, which is a polynomial of degree k, and compute all multiples of $n(x)$ having degrees less than $k+w$. Consider the set of all polynomials over $GF(2)$ of length w and the set of all w-bit integers as

$$\begin{aligned} Q_w &= \{1, x, x+1, x^2, x^2+1, x^2+x, \ldots, x^{w-1}+x^{w-2}+\cdots+1\} \ , \\ I_w &= \{0, 1, 2, \ldots, 2^w - 1\} \ . \end{aligned}$$

Let $q_i(x)$ be the ith element of Q_w and $m_i(x) = q_i(x)n(x)$ for $i \in I_w$. The polynomial $m_i(x)$ is of degree less than $k+w$, which we represent as an $(s+1)$-word number

$$m_i(x) = (M_{i,s}M_{i,s-1}\cdots M_{i,1}M_{i,0}) \ . \tag{1}$$

We then construct the table T_1 containing 2^w rows, in which we store the polynomial $m_i(x)$ using its most significant word (w-bits) as the index, i.e.,

$$T_1(M_{i,s}) = (M_{i,s-1}\cdots M_{i,1}M_{i,0}) \ , \tag{2}$$

for $i \in I_w$. An important observation is that the most significant words $M_{i,s}$ for $i \in I_w$ span the set Q_w, in other words, they are all unique.

Theorem 1 *The most significant words $M_{i,s}$ are all unique for $i \in I_w$.*

Proof Assume $M_{i,s} = M_{j,s}$ for $i \neq j$. The polynomial

$$p(x) = m_i(x) + m_j(x)$$

is of length k since

$$\begin{aligned} p(x) &= (M_{i,s}M_{i,s-1} \cdots M_{i,1}M_{i,0}) + (M_{j,s}M_{j,s-1} \cdots M_{j,1}M_{j,0}) \\ &= (0P_{s-1} \cdots P_1P_0) \ . \end{aligned}$$

Furthermore, $p(x)$ is divisible by $n(x)$ since

$$p(x) = q_i(x)n(x) + q_j(x)n(x) = (q_i(x) + q_j(x))n(x) \ ,$$

which means $p(x)$ can only be the zero polynomial, i.e., $i = j$. □

The table T_1 will be used in the *left-to-right* algorithm for reducing polynomials modulo $n(x)$, i.e., we compute $p(x) \bmod n(x)$. In order to perform the Montgomery-type reduction, we use a *right-to-left* algorithm and obtain $x^{-k}p(x) \bmod n(x)$. This algorithm requires that we construct a table of multiples of $n(x)$ based on the least significant words. Similarly, we take the polynomial $m_i(x) = q_i(x)n(x)$ for $i \in I_w$, and construct the table T_2 containing 2^w rows. The table T_2 keeps the polynomial $m_i(x)$, where we use the least significant word $M_{i,0}$ as the index, i.e.,

$$T_2(M_{i,0}) = (M_{i,s} \cdots M_{i,2}M_{i,1}) \ , \tag{3}$$

The uniqueness of the least significant words $M_{i,0}$ depends on whether $n(x)$ is not divisible by x.

Theorem 2 *The least significant words $M_{i,0}$ are all unique for $i \in I_w$ if only if $n(x)$ is not divisible by x.*

Proof Assume $M_{i,0} = M_{j,0}$ for $i \neq j$. The polynomial

$$p(x) = x^{-w}(m_i(x) + m_j(x))$$

is of length k since

$$\begin{aligned} p(x) &= x^{-w}\left((M_{i,s}M_{i,s-1} \cdots M_{i,1}M_{i,0}) + (M_{j,s}M_{j,s-1} \cdots M_{j,1}M_{j,0})\right) \\ &= x^{-w}(P_s \cdots P_2P_10) \\ &= (P_s \cdots P_2P_1) \ . \end{aligned}$$

Furthermore, $p(x)$ is divisible by $n(x)$ if $\gcd(x^w, n(x)) = 1$ since

$$p(x) = x^{-w}(q_i(x)n(x) + q_j(x)n(x)) = x^{-w}(q_i(x) + q_j(x))n(x) \ .$$

Therefore, $p(x)$ can only be the zero polynomial, i.e., $i = j$. The condition $\gcd(x^w, n(x)) = 1$ is satisfied if only if $n(x)$ is not divisible by x. □

The tables T_1 and T_2 are used to reduce a polynomial of length $sw+w$ to a polynomial of length sw using the irreducible polynomial $n(x)$. Let $p(x)$ be a polynomial of length $sw+w$, which is to be reduced, denoted as $p(x) = (P_s P_{s-1} \cdots P_1 P_0)$. The left-to-right algorithm computes $p(x)$ mod $n(x)$, while the right-to-left algorithm computes $x^{-w}p(x)$ mod $n(x)$. The resulting polynomial in both cases is of length sw.

Left-To-Right Reduction Algorithm: In order to reduce $p(x) = (P_s P_{s-1} \cdots P_1 P_0)$, we select the entry $(M_{s-1} M_{s-2} \cdots M_1 M_0)$ from the table T_1 using the index $P_s = M_s$. Since T_1 was constructed so that the element $(P_s M_{s-1} M_{s-2} \cdots M_1 M_0)$ resides in position P_s, we have

$$\begin{aligned} p(x) &:= (P_s P_{s-1} \cdots P_1 P_0) + (P_s M_{s-1} M_{s-2} \cdots M_1 M_0) \\ &:= (0 P'_{s-1} \cdots P'_1 P'_0) \ , \end{aligned}$$

where $P'_j = P_j + M_j$ for $j = 0, 1, \ldots, s-1$. We also discard the most significant w bits of the new $p(x)$. Since we add a multiple of $n(x)$ to $p(x)$, and obtain a polynomial of length sw, we effectively compute $p(x)$ mod $n(x)$, as required. We denote the above computation as

$$p(x) := p(x) + T_1(P_s) \ . \tag{4}$$

Right-To-Left Reduction Algorithm: In order to reduce $p(x) = (P_s P_{s-1} \cdots P_1 P_0)$, we select the entry $(M_s M_{s-1} \cdots M_1)$ from the table T_2 using the index $P_0 = M_0$. Since T_2 was constructed so that the element $(M_s M_{s-1} \cdots M_1 P_0)$ resides in position P_0, we have

$$\begin{aligned} p(x) &:= (P_s P_{s-1} \cdots P_1 P_0) + (M_s M_{s-1} \cdots M_1 P_0) \\ &:= (P'_s P'_{s-1} \cdots P'_1 0) \ , \end{aligned}$$

where $P'_j = P_j + M_j$ for $j = 1, 2, \ldots, s$. We then shift the new $p(x)$ w bits to the right, i.e., we multiply it by x^{-w}. Since we add a multiple of $n(x)$ to $p(x)$, and then, multiply it by x^{-w} in order to obtain a polynomial of length sw, we effectively compute $x^{-w}p(x)$ mod $n(x)$, as required. We will denote the above computation as

$$p(x) := x^{-w}(p(x) + T_2(P_0)) \ . \tag{5}$$

4 Standard Multiplication with Table Lookup Reduction

The standard multiplication algorithm computes $c(x) = a(x)b(x)$ mod $n(x)$ given $a(x)$, $b(x)$, and $n(x)$. In order to apply the table lookup reduction method, we first construct the table T_1 using the generating polynomial $n(x)$. The algorithm then proceeds by multiplying one word of $a(x)$ by the entire $b(x)$, which is followed by a table lookup reduction to reduce the partial product. We will call this algorithm STDMUL, whose steps are given below:

Algorithm STDMUL
Step 0: Construct T_1 using $n(x)$ and w
Step 1. $c(x) := 0$
Step 2. for $i = s-1$ downto 0 do
Step 3. $\quad c(x) := x^w c(x) + A_i(x)b(x)$
Step 4. $\quad c(x) := c(x) + T_1(C_s)$
Step 5. return $c(x)$

The operation in Step 4 of STDMUL is performed by first discarding the sth (the most significant) word of $c(x) = (C_s C_{s-1} \cdots C_1 C_0)$, and then by adding the s-word number $T_1(C_s) = (M_{s-1} M_{s-2} \cdots M_1 M_0)$ to the partial product $c(x)$ as

$$\begin{array}{cccccc} C_{s-1} & C_{i-2} & \cdots & C_1 & C_0 \\ M_{s-1} & M_{s-2} & \cdots & M_1 & M_0 \end{array}$$

Similarly, we perform the standard squaring operation using the table lookup reduction method. This algorithm is denoted as STDSQU, whose steps are given below. An important saving in this case is that the cross product terms disappear because the ground field is $GF(2)$. Since

$$a^2(x) = \sum_{i=0}^{k-1} a_i x^{2i} \;=\; a_{k-1} x^{2(k-1)} + a_{k-2} x^{2(k-2)} + \cdots + a_1 x^2 + a_0 \;, \tag{6}$$

the multiplication step (i.e., Step 3) in STDMUL can be skipped. The standard squaring algorithm, called STDSQU, starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$ given by

$$c(x) = (a_{k-1} \mathbf{0} a_{k-2} \mathbf{0} \cdots \mathbf{0} a_1 \mathbf{0} a_0) \;,$$

and then performs the reduction steps using the table T_1.

Algorithm STDSQU
Step 0: Construct T_1 using $n(x)$ and w
Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2. for $i = 2s-1$ downto s do
Step 3. $\quad c(x) := c(x) + T_1(C_i)$
Step 4. return $c(x)$

We perform the operation in Step 3 of STDSQU by first discarding the ith (the most significant) word of $c(x) = (C_i C_{i-1} \cdots C_1 C_0)$, and then by adding the s-word number

$$T_1(C_i) = (M_{s-1} M_{s-2} \cdots M_1 M_0)$$

to the partial product $c(x)$ by aligning M_{s-1} with C_{i-1} from the left:

$$\begin{array}{cccccccc} C_{i-1} & C_{i-2} & \cdots & C_{i-s+1} & C_{i-s} & \cdots & C_1 & C_0 \\ M_{s-1} & M_{s-2} & \cdots & M_1 & M_0 & & & \end{array}$$

Thus, each addition operation in Step 3 requires exactly s XOR operations.

5 An Example of Standard Multiplication

We take the field $GF(2^8)$ to illustrate the construction of the table T_1, and also give an example of the standard multiplication operation using the table lookup reduction method. We select the irreducible polynomial as

$$n(x) = x^8 + x^5 + x^3 + x^2 + 1 \;. \tag{7}$$

We also select $w = 4$, which gives $s = k/w = 8/4 = 2$. The table T_1 is constructed by taking a polynomial $q(x)$ from Q_4, multiplying it by $n(x)$ to obtain $m(x) = q(x)n(x)$, and then placing the least significant $s = 2$ words of $m(x)$ to T_1 using the most significant word as the index. The step-by-step construction of T_1 is shown in Table 1. The multiples of $n(x)$ do not necessarily come in an increasing order, however, we have a complete set of first words, and thus, we can use these values as their indices to store them in T_1. The table T_1 is shown in Table 1 in its unsorted form.

Table 1: The construction of T_1 and T_2 for $n(x) = (0001\ 0010\ 1101)$.

$q(x)$	$m(x)$
(0000)	(0000 0000 0000)
(0001)	(0001 0010 1101)
(0010)	(0010 0101 1010)
(0011)	(0011 0111 0111)
(0100)	(0100 1011 0100)
(0101)	(0101 1001 1001)
(0110)	(0110 1110 1110)
(0111)	(0111 1100 0011)
(1000)	(1001 0110 1000)
(1001)	(1000 0100 0101)
(1010)	(1011 0011 0010)
(1011)	(1010 0001 1111)
(1100)	(1101 1101 1100)
(1101)	(1100 1111 0001)
(1110)	(1111 1000 0110)
(1111)	(1110 1010 1011)

i	$T_1(i)$
(0000)	(0000 0000)
(0001)	(0010 1101)
(0010)	(0101 1010)
(0011)	(0111 0111)
(0100)	(1011 0100)
(0101)	(1001 1001)
(0110)	(1110 1110)
(0111)	(1100 0011)
(1001)	(0110 1000)
(1000)	(0100 0101)
(1011)	(0011 0010)
(1010)	(0001 1111)
(1101)	(1101 1100)
(1100)	(1111 0001)
(1110)	(1010 1011)
(1111)	(1000 0110)

i	$T_2(i)$
(0000)	(0000 0000)
(1101)	(0001 0010)
(1010)	(0010 0101)
(0111)	(0011 0111)
(0100)	(0100 1011)
(1001)	(0101 1001)
(1110)	(0110 1110)
(0011)	(0111 1100)
(1000)	(1001 0110)
(0101)	(1000 0100)
(0010)	(1011 0011)
(1111)	(1010 0001)
(1100)	(1101 1101)
(0001)	(1100 1111)
(0110)	(1111 1000)
(1011)	(1110 1010)

Furthermore, we also give the table T_2 in Table 1, which is to be used by the right-to-left algorithm for computing the Montgomery multiplication and squaring operations in $GF(2^8)$. The table T_2 is constructed by placing the polynomial $m(x)$ in T_2 using its least significant word as the index. As an example for STDMUL, we take

$$\begin{aligned} a(x) &= x^7 + x^6 + x^4 + x^3 + x + 1 \ , \\ b(x) &= x^7 + x^5 + x^3 + x^2 + x \ . \end{aligned}$$

We have $a = (A_1A_0) = (1101\ 1011)$ and $b = (B_1B_0) = (1010\ 1110)$. The algorithm starts with $c = 0$ and then performs the following steps to find the result:

$$\begin{aligned} i = 1 \quad \text{Step 3:} \quad c(x) &:= c(x)x^4 + A_1(x)b(x) = (C_2C_1C_0) \\ &= 0 + (1101)(1010\ 1110) = (0111\ 0110\ 0110) \\ \text{Step 4:} \quad c(x) &:= c(x) + T_1(C_2) = (C_1C_0) + T_1(C_2) \\ &= (0110\ 0110) + (1100\ 0011) = (1010\ 0101) \end{aligned}$$

$$\begin{aligned} i = 0 \quad \text{Step 3:} \quad c(x) &:= c(x)x^4 + A_0(x)b(x) = (C_2C_1C_0) \\ &= (1010\ 0101\ 0000) + (1011)(1010\ 1110) = (1110\ 1101\ 0010) \\ \text{Step 4:} \quad c(x) &:= c(x) + T_1(C_2) = (C_1C_0) + T_1(C_2) \\ &= (1010\ 1011) + (1101\ 0010) = (0111\ 1001) \end{aligned}$$

Therefore, the result is found as $c(x) = (0111\ 1001) = x^6 + x^5 + x^4 + x^3 + 1$.

6 Montgomery Multiplication with Table Lookup Reduction

The Montgomery multiplication of two elements $a(x)$ and $b(x)$ is defined as the product

$$x^{-k}a(x)b(x) \bmod n(x) \ . \tag{8}$$

Note that the inverse element $x^{-k} = (x^k)^{-1}$ exists since $n(x)$ is an irreducible polynomial, and thus $\gcd(x^k, n(x)) = 1$. The details of the Montgomery multiplication algorithm in $GF(2^k)$ and its properties are found in [2]. The algorithm requires that we compute $N_0'(x)$ in advance which is the least significant word of the polynomial $n'(x)$ defined as

$$n'(x) = -n(x)^{-1} \bmod x^k \ . \tag{9}$$

An algorithm for computing $N_0'(x)$ is also described in [2]. However, it turns out that the Montgomery multiplication method using the table lookup reduction algorithm does not require the computation of $N_0'(x)$. This is not surprising since the table T_2 keeps all 2^w multiples of $n(x)$, and therefore, there is no need to separately compute $N_0'(x)$.

The steps of the Montgomery multiplication algorithm using the table lookup reduction method are given below. The algorithm, which we call MONMUL, is based on the right-to-left reduction algorithm described in §3.

Algorithm MONMUL
Step 0: Construct T_2 using $n(x)$ and w
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. $\quad c(x) := x^{-w}(c(x) + T_2(C_0))$
Step 5. return $c(x)$

At the end of Step 1, we have an $(s+1)$-word number $C = (C_s C_{s-1} \cdots C_1 C_0)$. We discard the 0th (the least significant) word C_0, and then add the s-word number $T_2(C_0) = (M_s M_{s-1} \cdots M_2 M_1)$ to the partial product $c(x)$ by aligning M_1 with C_1 from the right:

$$\begin{array}{ccccc} C_s & C_{s-1} & \cdots & C_2 & C_1 \\ M_s & M_{s-1} & \cdots & M_2 & M_1 \end{array}$$

Thus, the 1-word shift operation denoted as the multiplication by x^{-w} is implicitly performed. Each addition operation in Step 4 requires exactly s XOR operations. Similarly, the Montgomery squaring method computes

$$c(x) = x^{-k} a^2(x) \bmod n(x) \ . \tag{10}$$

In order to compute $c(x)$, we first obtain $a^2(x)$ using the property (6), and then reduce the result with the help of the right-to-left reduction algorithm, as seen below.

Algorithm MONSQU
Step 0: Construct T_2 using $n(x)$ and w
Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2. for $i = 0$ to $s - 1$ do
Step 3. $\quad c(x) := x^{-w}(c(x) + T_2(C_0))$
Step 4. return $c(x)$

When Step 1 completes, we have an $(2s-1)$-word number $C = (C_{2s-2} C_{2s-3} \cdots C_1 C_0)$. In the beginning of the ith step, the number $c(x)$ is an $(2s-i-1)$-word number $C = (C_{2s-i-2} C_{2s-i-3} \cdots C_1 C_0)$. We perform the operation in Step 3 of MONSQU by first discarding the 0th (the least significant)

word of $c(x)$, and then by adding the s-word number $T_2(C_0) = (M_s M_{s-1} \cdots M_1)$ to the partial product $c(x)$ by aligning M_1 with C_1 from right, as follows:

$$\begin{array}{cccccccc} C_{2s-i-2} & C_{2s-i-1} & \cdots & C_s & C_{s-1} & \cdots & C_2 & C_1 \\ & & & M_s & M_{s-1} & \cdots & M_2 & M_1 \end{array}$$

As in the case for the Montgomery multiplication, the 1-word shift operation, i.e., the multiplication by x^{-w}, is implicitly performed, and each addition operation in Step 3 requires exactly s XOR operations.

7 An Example of Montgomery Multiplication

We take the field $GF(2^8)$ and the same irreducible polynomial $n(x)$ as the one exemplified in §5. The construction of the table T_2 was already shown in §5. In this case, we compute the Montgomery product of the elements

$$\begin{aligned} a(x) &= x^7 + x^6 + x^4 + x^3 + x + 1 \ , \\ b(x) &= x^7 + x^5 + x^3 + x^2 + x \ . \end{aligned}$$

Therefore, we will be computing $c(x) = x^{-8} a(x) b(x) \bmod n(x)$. The right-to-left reduction algorithm for computing the product $c(x)$ starts with $a(x) = (A_1 A_0) = (1101\ 1011)$, $b(x) = (B_1 B_0) = (10101110)$, $c(x) = 0$, and performs the following steps:

$$\begin{array}{llllll} i = 0 & \text{Step 3:} & c(x) & := & c(x) + A_0(x) b(x) = (C_2 C_1 C_0) \\ & & & = & 0 + (1011)(1010\ 1110) = (0100\ 1000\ 0010) \\ & \text{Step 4:} & c(x) & := & x^{-4}(c(x) + T_2(C_0)) = (C_2 C_1) + T_2(C_0) \\ & & & = & (0100\ 1000) + (1011\ 0011) = (1111\ 1011) \\ i = 1 & \text{Step 3:} & c(x) & := & c(x) + A_1(x) b(x) = (C_2 C_1 C_0) \\ & & & = & (1111\ 1011) + (1101)(1010\ 1110) = (0111\ 1001\ 1101) \\ & \text{Step 4:} & c(x) & := & (x^{-4} c(x) + T_2(C_0)) = (C_2 C_1) + T_2(C_0) \\ & & & = & (0111\ 1001) + (0001\ 0010) = (0110\ 1011) \end{array}$$

The product is found as $c(x) = (0110\ 1011) = x^6 + x^5 + x^3 + x + 1$.

8 Analysis of the Algorithms

In this section, we analyze the standard and Montgomery multiplication algorithms by calculating the size of the lookup tables and counting the total number of the table read and the word-level $GF(2)$ addition and multiplication operations.

First we start the word-level $GF(2)$ operations. The word-level addition is simply the bit-wise `XOR` operation on a pair of 1-word binary numbers, which is a readily available instruction on most general purpose microprocessors and signal processors. The word-level multiplication operation receives two 1-word (w-bit) polynomials $A(x)$ and $B(x)$ defined over $GF(2)$, and computes the 2-word polynomial $C(x) = A(x)B(x)$. The degree of the product polynomial $C(x)$ is $2(w-1)$. For example, given $A = (1101)$ and $B = (1010)$, this operation computes C as

$$A(x)B(x) \ = \ (x^3 + x^2 + 1)(x^3 + x) \ = \ x^6 + x^5 + x^4 + x \ = \ (0111\ 0010) \ .$$

The implementation of this operation, which we call MULGF2 as in [2], can be performed in three different ways: 1) An instruction implemented on the processor, 2) The table lookup method, and 3) The emulation using the XOR and SHIFT operations. The details of the analysis of these methods can be found in [2]. The fastest is the first one, while the slowest is the last one. In this paper, we will simply count the number of MULGF2 operations, and assume that they are implemented using any of the above methods. 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, the total size of these tables is of size $2 \times 2^w \times 2^w \times w$ bits. We store the values H and L in two table reads. Other approaches are also possible.

The other operation to consider is the table read operation from T_1 and T_2. We will denote this operation using TABLEREAD, and count the total number of TABLEREAD operations. In regards to the sizes of these tables, we note that the table T_1 (or T_2) has 2^w rows, each of which contains a polynomial of length k. This implies that the size of the tables is $2^w \times k$ bits. The space requirements for the tables T_1 (or T_2) for performing the TABLEREAD operation are exemplified in Table 2.

Table 2: The size of the table in bytes for the TABLEREAD operation.

k	$w = 4$	$w = 8$	$w = 10$	$w = 16$
160	320	5,120	20,480	1,310,720
256	512	8,192	32,768	2,097,152
512	1,024	16,384	65,536	4,194,304
1024	2,048	32,768	131,072	8,388,608

For example, if $w = 8$ and $k = 160$, the size of the table is $2^8 \times 20 = 5,120$ bytes, which is quite reasonable. However, the table size becomes excessive as we increase the wordsize. For a fixed field size k, we can decide about the wordsize w given the memory capacity of the computer system.

We give the steps of the algorithms STDMUL and MONMUL in detail in Table 3, together with the number of TABLEREAD, MULGF2, and XOR operations.

The total number of operations for STDMUL, STDSQU, MONMUL, MONSQU are summarized in Table 4.

Furthermore, in Table 5, we give the coefficients of the highest term s^2 in the operation counts of the multiplication algorithms using the proposed table lookup reduction method, comparing them to those algorithms which do not utilize the table lookup reduction method. The detailed timing requirements of the algorithms without the table lookup reduction method are given in [2].

Table 3: The operation counts for the STDMUL and MONMUL algorithms.

STDMUL	TABLEREAD	MULGF2	XOR
`for i=0 to s do`	-	-	-
`  C[i]:=0`	-	-	-
`for i=s-1 downto 0 do`	-	-	-
`  P:=0`	-	-	-
`  for j=s-1 downto 0 do`	-	-	-
`    (H,L):=MULGF2(A[i],B[j])`	-	s^2	-
`    C[j+1]:=C[j] XOR H XOR P`	-	-	$2s^2$
`    P:=L`	-	-	-
`  C[0]:=P`	-	-	-
`  for j=0 to s-1 do`	-	-	-
`    C[j]:=C[j] XOR T[C[s]][j]`	s	-	s^2
MONMUL			
`for i=0 to s do`	-	-	-
`  C[i] := 0`	-	-	-
`for i=0 to s-1 do`	-	-	-
`  P := 0`	-	-	-
`  for j=0 to s-1 do`	-	-	-
`    (H,L):=MULGF2(A[i],B[j])`	-	s^2	-
`    C[j]:=C[j] XOR L XOR P`	-	-	$2s^2$
`    P := H`	-	-	-
`  C[s] := P`	-	-	-
`  for j=0 to s-1 do`	-	-	-
`    C[j] := C[j+1] XOR T[C[0]][j]`	s	-	s^2

Table 4: The operation counts for the algorithms.

	TABLEREAD	MULGF2	XOR	SHIFT
STDMUL	s	s^2	$3s^2$	-
STDSQU	s	-	s^2	-
MONMUL	s	s^2	$3s^2$	-
MONSQU	s	-	s^2	-

Table 5: The operation count orders for the algorithms.

Using this method	MULGF2	XOR	SHIFT
STDMUL	1	3	0
STDSQU	0	1	0
MONMUL	1	3	0
MONSQU	0	1	0
Using the method in [2]			
STDMUL	1	$\frac{3w}{2}+3$	$2(w+1)$
STDSQU	0	$\frac{9w}{4}$	$3w$
MONMUL	2	4	0
MONSQU	1	2	0

9 Special Irreducible Polynomials

The presented table lookup reduction method and the resulting multiplication algorithms work for an arbitrary generating polynomial. The tables T_1 (or T_2) are constructed and used without assuming a special structure in $n(x)$. This is an important property of the table lookup reduction method, making it applicable for any value of k and for arbitrary generating polynomials. However, it may be possible to avoid the construction of the table or to reduce the size of it or to reduce the time taken by the algorithm when the generating polynomial has a special structure. In this section we consider several different forms irreducible polynomials for generating the field $GF(2^k)$.

We start with the case in which the generating polynomial $n(x)$ is of the form

$$n(x) = x^k + a_j x^j + a_{j-1} x^{j-1} + \ldots + a_1 x + a_0 \ ,$$

where $j = tw$ and $t < s$ is an integer. This implies that x^k is the only nonzero term among the $(s-t)w$ most significant terms of $n(x)$, and thus, there is no need to prepare the whole table T_1 as the multiples of $n(x)$ will have all zeros in the most significant $s-t$ words, except the most significant word which is used as an index. It suffices to store the least significant t words of the multiples of $n(x)$ to reduce the partial product. Therefore, the size of the table T_1 will be $2^w \times t \times w$ instead of $2^w \times s \times w$, i.e., the table size reduces linearly depending on the value of t. Furthermore, we only add the t least significant words of the operands during the addition operation since we know the remaining $s-t$ words are zero. Therefore, Step 4 of STDMUL is performed by first discarding the sth (the most significant) word of $c(x) = (C_s C_{s-1} \cdots C_1 C_0)$ and then by adding the t-word number $T_1(C_s) = (M_{t-1} M_{t-2} \cdots M_1 M_0)$ to the partial product as

$$\begin{array}{cccccccc} C_{s-1} & C_{s-2} & \cdots & C_t & C_{t-1} & \cdots & C_1 & C_0 \\ & & & & M_{t-1} & \cdots & M_1 & M_0 \end{array}$$

The most significant $s-t$ words are not added since $(M_{s-1} M_{s-2} \cdots M_t)$ is known to be all zero. This operation requires t word-level XOR operations in the reduction step (Step 4) instead of s.

9.1 Trinomials

When the irreducible polynomial is a trinomial of the form

$$n(x) = x^k + x^j + 1 \ ,$$

where $1 \leq j < k$, then, it turns out that there is no need to prepare the table T_1 (or T_2). We can use the the most significant (or the least significant) word of the partial product in order to reduce it. There are different approaches, depending on the value of j. If j is an integer multiple of the word size w, i.e., $j = tw$, we can use the word-level shifts of the partial product. If j is not an integer multiple of w, i.e., $j = tw + u$ for some $1 \leq u < w$, then, certain bit-level operations will need to be performed.

For $j = tw$, the left-to-right algorithm reduces the $(s+1)$-word partial product $c(x) = (C_s C_{s-1} \cdots C_1 C_0)$ by adding C_s multiple of the irreducible polynomial $n(x)$ to it as

$$c(x) := c(x) + (x^{sw} + x^{tw} + 1) C_s \ .$$

This implies that we need to add C_s to C_s, C_t, and C_0 in Step 4 of the algorithm STDMUL. However, we do not perform the first addition $C_s + C_s = 0$, as follows:

$$\begin{array}{ccccccc} C_{s-1} & \cdots & C_{t+1} & C_t & \cdots & C_1 & C_0 \\ & & & C_s & & & C_s \end{array}$$

Thus, during the ith step of the reduction, we perform 2 `XOR` operations. The multiplication algorithms described in [7, 8] are essentially the same.

The right-to-left reduction algorithm, on the other hand, uses C_0 to reduce the partial product from the right (the least significant). Effectively, it performs the operation

$$c(x) := c(x) + (x^{sw} + x^{tw} + 1)C_0 \ ,$$

in order to reduce the partial product 1-word from the right. This implies that we add C_0 to C_0, C_t, and C_s as follows:

$$\begin{array}{cccccccc} C_s & C_{s-1} & \cdots & C_{t+1} & C_t & \cdots & C_1 \\ C_0 & & & & C_0 & & \end{array}$$

Similarly, we do not perform the addition of the least significant words $C_0 + C_0 = 0$, and obtain the s-word partial product using only 2 `XOR` operations.

If j is not a multiple of w, but, say $j = tw + u$ for a positive integer $1 \leq u < w$, we need to perform the operation

$$c(x) := c(x) + (x^{sw} + x^{tw+u} + 1)C_s$$

to reduce the most significant word of C_s of $c(x)$. This implies that C_s is added to C_s and C_0, which takes care of the part $c(x)+(x^{sw}+1)C_s$. In order to add $x^{tw}(x^u C_s)$ to the partial product, C_s needs to be shifted u bits to left, which produces a 2-word number $(M_1 M_0)$. Let $C_s = (c_{k+w-1} \cdots c_{k+1} c_k)$, then, $(M_1 M_0)$ is obtained as

$$(M_1)\ (M_0) = (0 \cdots 0\ c_{k+w-1} c_{k+w-2} \cdots c_{k+w-u})\ (c_{k+w-u-1} \cdots c_{k+1} c_k\ 0 \cdots 0) \ .$$

We then add M_1 and M_2 to C_{t+1} and C_t, respectively. The operations required to reduce $c(x)$ using the left-to-right algorithms are shown below:

$$\begin{array}{cccccccc} C_{s-1} & \cdots & C_{t+1} & C_t & \cdots & C_1 & C_0 \\ & & M_1 & M_0 & & & C_s \end{array}$$

Similarly, the addition of the most significant words $C_s + C_s = 0$ is ignored. In summary, the reduction operation during the ith step requires a few bit operations to obtain M_1 and M_0, and then 3 word-level `XOR` operations.

On the other hand, the right-to-left reduction algorithm performs the operation

$$c(x) := c(x) + (x^{sw} + x^{tw+u} + 1)C_0$$

in order to reduce the least significant word of C_0 of $c(x)$. This implies that we add C_0 to C_s and C_0, taking care of the part $c(x) + (x^{sw} + 1)C_0$. In order to perform, the operation $c(x) := c(x) + x^{tw}(x^u C_0)$, we shift C_0 to the left u times, obtaining a 2-word number $(M_1' M_0')$ as before. We then add M_1' and M_0' to C_{t+1} and C_t, respectively. The final reduction operation is

$$\begin{array}{cccccccc} C_s & \cdots & C_{t+1} & C_t & \cdots & C_2 & C_1 \\ C_0 & & M_1' & M_0' & & & \end{array}$$

The addition of the least significant words $C_0 + C_0 = 0$ is ignored. The right-to-left reduction algorithm requires a few bit operations to compute M_1' and M_0', followed by 3 word-level `XOR` operations.

9.2 All-One-Polynomials

In this case, the generating polynomial $n(x)$ will be of the form

$$n(x) = x^k + x^{k-1} + \cdots + x + 1 \ .$$

It is known that an all-one-polynomial is irreducible if and only if $k+1$ is prime and 2 is primitive modulo $k+1$ [5]. For $k \leq 100$, the all-one-polynomial is irreducible for the following values of k: 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82, and 100.

The reduction is often performed using the polynomial $(x+1)n(x) = x^{k+1} + 1$. Since $k+1$ is not an integer multiple of w, the exact word-level shifting is not possible with this polynomial. In this case, we need to perform the operation on the $(s+1)$-word partial product $c(x)$

$$c(x) := c(x) + (x^{sw+1} + 1)A$$

where A is a 1-word number obtained from C_s, as we will explain shortly. Since the most significant word of the new $c(x)$ needs to be zero, we obtain

$$C_s + (Ax) = 0 \ ,$$

and therefore, $Ax = C_s$. If the least significant bit of C_s, denoted as c_k, is equal to zero, the computation of A is very simple: $A = C_s/x$, i.e., C_s is shifted 1 bit to the right to obtain A. Thus, A is actually an $(w-1)$-bit number, and when Ax is added to C_s, the result is zero. The final reduction is performed using only one `XOR` operation as

$$\begin{array}{cccccc} C_{s-1} & C_{s-2} & \cdots & C_1 & C_0 \\ & & & & A \end{array}$$

However, when c_k is not equal to zero, we have no other choice except to add the entire $n(x)$ to $c(x)$. This implies that we add the w-bit all-one-polynomial $2^w - 1 = (11\cdots1) = 1_w$ to each word of $c(x)$ starting from 0 ending at $s-1$, as

$$\begin{array}{cccccc} C_{s-1} & C_{s-2} & \cdots & C_1 & C_0 \\ 1_w & 1_w & \cdots & 1_w & 1_w \end{array}$$

This operation makes the least significant bit c_k zero. Therefore, if c_k is nonzero, we need to perform an additional s `XOR` operations to reduce the $(s+1)$-word partial product $c(x)$.

On the other hand, the right-to-left reduction algorithm performs

$$c(x) := c(x) + (x^{sw+1} + 1)C_0$$

in order to reduce the least significant word C_0 of $c(x)$. When C_0 is added to $c(x)$, the new least significant word C_0 will be zero. However, we also need to add $x^{sw+1}C_0$ to $c(x)$. If the most significant bit c_{w-1} of C_0 is zero, this operation is easily accomplished. We compute $A_0 = C_0 x$, and since c_{w-1} is zero, A_0 fits into w bits, i.e., 1-word. The following operation accomplishes the reduction:

$$\begin{array}{cccc} C_s & C_{s-1} & \cdots & C_1 \\ A_0 & & & \end{array}$$

If c_{w-1} is not zero, then A_0 is no longer a 1-word number: it is a $(w+1)$-bit number containing a one its wth position: $A_0 = (1c_{w-1}c_{w-2}\cdots c_0 0)$. The reduced partial product in this case becomes

$$c(x) := (1 \ C_s C_{s-1} \cdots C_2 C_1) \ .$$

As in the case for the left-to-right reduction algorithm, we have no other choice except to add the entire all-one-polynomial $n(x)$ in order to reduce the most significant bit:

$$\begin{array}{cccccc} C_s & C_{s-1} & \cdots & C_2 & C_1 \\ 1_w & 1_w & \cdots & 1_w & 1_w \end{array}$$

Therefore, when c_{w-1}, we need to perform an additional s `XOR` operations to reduce the $(s+1)$-word partial product $c(x)$.

10 Integer Case

In this section we consider the extension of the table lookup reduction method to the integers. We are interested in reducing the $(s+1)$-word integer a modulo n, where n is an arbitrary integer of length s words. In the following, we show that the right-to-left reduction algorithm works if and only if n is odd. On the other hand, the left-to-right reduction algorithm works for $n > 2^{k+w}/(2^w+1)$, but it is inefficient.

First we concentrate on the right-to-left reduction algorithm. Let $i \in I_w$ and $m_i = in$. Since n is an s-word number and $0 \leq i \leq 2^w - 1$, the number m_i for all i is an $(s+1)$-word number, represented as

$$m_i = (m_{i,s} m_{i,s-1} \cdots m_{i,1} m_{i,0}) \ .$$

The table T_2 is constructed by the right-to-left reduction algorithm using the least significant words $m_{i,0}$ as

$$T_1(-m_{i,0}) = (m_{i,s} m_{i,s-1} \cdots m_{i,1}) \ .$$

We note the minus sign in the index, which helps us to add the table entry to the partial product, instead of subtracting it. An important requirement is that all $m_{i,s}$ be unique for $i \in I_w$. This way we construct the complete table of the multiples of n to be added to the partial product. The uniqueness of the least significant words depends on whether n is odd, which is easily proven as follows.

Theorem 3 *The least significant words of $m_i = in$ are unique for $i \in I_w$ if and only if n is odd.*

Proof The least significant word of m_i is given as $in \bmod 2^w$. The set of residues $in \bmod 2^w$ for $i \in I_w$ is complete if and only if $\gcd(n, 2^w) = 1$. This is satisfied when n is odd. □

Therefore, the right-to-left reduction algorithm used in the Montgomery multiplication and squaring algorithms works in the integer case for an odd n only. This gives us a table lookup based Montgomery multiplication algorithm, which is more efficient than the regular Montgomery multiplication since the reduction step is significantly simplified. Furthermore, there is no need to compute $n_0' = -n_0^{-1}$.

Unfortunately, the left-to-right reduction algorithm does not work as well. It is impractical for two main reasons: 1) The most significant words are not always unique, 2) Even when they are unique, the left-to-right reduction cannot use addition in place of subtraction since this will cause a carry to higher order bits. As an example, we take $w = 3$ and $n = 35 = (100\ 011)$, and produce the table of values $m_i = in$ below:

i	$m_i = in$
001	000 100 011
010	001 000 110
011	001 101 001
100	010 001 100
101	010 101 111
110	011 010 010
111	011 110 101

An inspection of the above table shows that the most significant words are not unique for $n = 35$, for example, 001, 010, and 011 appear twice, while 100, 101, 110, and 111 do not appear. We prove below that collisions occur if $n < 2^{k+w}/(2^w + 1)$.

Theorem 4 *If $n < 2^{k+w}/(2^w + 1)$, then, there is always a collision, i.e., there are at least two equal most significant words. Otherwise, the most significant words are unique.*

Proof We consider all multiples with s words. Let I be the integer such that $In < 2^{k+w} \leq (I+1)n$, i.e., In is the greatest multiple of n, that has $(s + 1)$ words. Note that the difference of any two consecutive multiples is n, which is less than 2^k, and thus, the difference of the most significant words of the consecutive multiples can be at most one. In particular, In has $2^w - 1 = 1 \cdots 1$ as the most significant word. Thus, the most significant words form a monotone increasing sequence. As the largest one is $1 \cdots 1$, all possible single words exist among them. Therefore, we have $I \geq 2^w$. When $I = 2^w$, we have uniqueness and when $I > 2^w$ we have collision(s).

For $n < 2^{k+w}/(2^w + 1)$, using the definition of I, we have

$$2^{k+w} \leq (I + 1)n < (I + 1)(2^{k+w})/(2^w + 1) \ ,$$

which yields $2^w < I$, i.e., we always have a collision. For $n \geq 2^{k+w}/(2^w+1)$, again by the definition of I, we have

$$2^{k+w} > In \geq 2^{k+w}/(2^w + 1)I \ ,$$

which yields $2^w + 1 > I$, i.e., we have no collision. □

During the reduction, we need to make sure that the number we subtract is less than the partial product. This can be done by comparison, and if the selected multiple is larger, we can use the previous multiple of n instead. If the most significant words of m_is are not unique, i.e., if there are collisions, then we have the problem of choosing the correct multiple. We can solve this problem by assigning the smallest multiple for all of the of the indices at which the collision occurs, which will reduce the probability of a negative result. For example, for the most significant word 001, we can assign the smaller multiple which is 001 000 110. Similarly, we can use 010 001 100 and 011 010 010, for the most significant words 010 and 011, respectively. However, in order to reduce the products with the most significant words not in our list, we will have to either assign the largest multiple 011 110 101 to all of them or continue to produce more multiples until we get all possible most significant words. Note that, even after using this method, we still have to compare the numbers and use the previous multiple if necessary. In short, when there is a collision, the proposed table lookup approach is not practical.

11 Conclusions

In this paper, we proposed a table lookup based reduction method for performing the standard and Montgomery multiplication and squaring operations in $GF(2^k)$ using the polynomial basis. The proposed method yields word-level algorithms, enabling software implementations of the finite field arithmetic operations which find applications most notably in elliptic curve cryptography. We treated the special irreducible polynomials and the integer case in detail and gave the algorithmic details for these methods together with the complexity analysis in terms of the number of basic arithmetic operations. The proposed algorithm is more efficient than the previously published results, and in the case of special irreducible polynomials (particularly, trionomials), the proposed method reduces to already known algorithms found in the literature.

In the integer case, the right-to-left reduction algorithm works well, providing an efficient version of the Montgomery multiplication algorithm. However, the left-to-right reduction algorithm is not efficient mainly due to the fact that the most significant words of the integer multiples of the modulus are not unique.

References

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

[2] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Design, Codes and Cryptography*, 14(1):57–69, April 1998.

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

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

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

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

[7] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. New York, NY: Springer-Verlag, 1995.

[8] 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 K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. New York, NY: Springer-Verlag, 1996.