Chapter 5
Fast Finite Field Multiplication

Serdar Süer Erdem, Tuğrul Yanık, and Çetin Kaya Koç

5.1 Introduction

Finite fields are the most commonly used arithmetical structures in cryptography [14, 16] and coding [3, 19, 21]. Many algorithms in cryptographic and coding applications are defined in terms of finite field arithmetic operations. The elliptic curve cryptosystems [11, 17] and the Diffie-Hellman key exchange [8] algorithm are important examples of such cryptographic applications. Also, common error control codes such as Reed-Solomon and BCH codes are based on finite field theory [4, 21].

An algebraic field consists of a set and two operations defined over this set. The real numbers, the rational numbers, and the complex numbers under addition and multiplication are examples of algebraic fields. In fact, algebraic fields are the generalization of these usual number systems as described below.

- One of the field operations satisfies the general properties of the usual addition. For this operation, an identity element exists and each element has an inverse. This identity element is called additive identity or zero element.
- The other field operation satisfies the general properties of the usual multiplication. For this operation, an identity element (multiplicative identity) exists and each element, except the zero element, has an inverse. Also, this operation distributes over the first operation like the usual multiplication distributes over the usual addition.

Finite fields are algebraic fields with finite number of elements. These fields take the place of the familiar fields like the real numbers in cryptography and coding. Because they have finite number of elements, the operations on them cannot produce infinitely large results. Also, the finite field operations always produce exact results,

Gebze Institute of Technology, e-mail: serdem@gyte.edu.tr · Fatih University, Istanbul, e-mail: tyanik@fatih.edu.tr · City University of Istanbul & University of California Santa Barbara, e-mail: koc@cryptocode.net

not approximate results. Thus, they do not suffer from truncation errors like the floating point operations.

The fast implementation of the finite field multiplication is essential in cryptographic and coding applications. This is because the finite field addition and multiplication are the most frequently used operations in these applications. The finite field addition is relatively simple, compared to the multiplication. On the other hand, the finite field multiplication is a substantially time-consuming operation in hardware and software implementations.

In this chapter, the efficient finite field multiplication methods are discussed after giving some preliminary facts about finite fields. The discussion is handled separately for the three main classes of finite fields (prime fields, binary extension fields, and general extension fields).

5.2 Finite Fields

A finite field with q elements is denoted by $\mathbb{F}_q$. Such a field exists, if and only if $q = p^m$ for some prime p and a positive integer m. $\mathbb{F}_q$ is unique up to isomorphism. That is, every field with q elements is isomorphic to $\mathbb{F}_q$.

- $\mathbb{F}_p$ has a prime number of elements, and thus it is called prime field.
- $\mathbb{F}_{p^m}$ denotes its extension field with p^m elements.
- $\mathbb{F}_{2^m}$ is a special case of $\mathbb{F}_{p^m}$ and is called binary extension field.

The prime field $\mathbb{F}_p$ can be constructed by using integer modular arithmetic. In this construction, the field elements are represented by the set of integers $\{0,1,2,\ldots,p-1\}$. And, the field operations are defined as integer addition and multiplication modulo p.

The extension field $\mathbb{F}_{p^m}$ can be constructed by using polynomial modular arithmetic. In this construction, the field elements are represented by the polynomials over $\mathbb{F}_p$ of degree less than m. And, the field operations are defined as polynomial addition and multiplication modulo a degree m irreducible polynomial over $\mathbb{F}_p$.

The construction of the extension fields using polynomials over the prime fields is possible due to the fact that the extension field $\mathbb{F}_{p^m}$ is an m-dimensional vector space over the prime field $\mathbb{F}_p$. As an immediate result of this fact, a basis $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ always exists in $\mathbb{F}_{p^m}$ such that each element $a \in \mathbb{F}_{p^m}$ can be given by $a = a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{m-1}\alpha_{m-1}$ for a unique set of $a_i \in \mathbb{F}_p$. According to the theory of finite fields,

- A degree m irreducible polynomial over $\mathbb{F}_p$ always exists. The roots of these irreducible polynomials are in $\mathbb{F}_{p^m}$.
- Let $\alpha \in \mathbb{F}_{p^m}$ be some root of a degree m irreducible polynomial $\omega(x)$. Then, $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ constitutes a basis for $\mathbb{F}_{p^m}$ where 1 denotes the multiplicative identity. Such a basis is called polynomial basis.

In conclusion, when α is a root of an irreducible $\omega(x)$, $\omega(x)|_{x=\alpha} = 0$ and each element of $\mathbb{F}_{p^m}$ can be given by $(a_{m-1}x^{m-1} + \cdots + a_1x + a_0)|_{x=\alpha}$ for a unique set of

$a_i \in \mathbb{F}_p$. This is why the extension field elements can be represented by the polynomials over $\mathbb{F}_p$. However, since $\omega(x) = 0$ for $x = \alpha$, all arithmetic operations are performed modulo $\omega(x)$ in this representation.

As can be understood from the discussion so far, finite field arithmetic is based on modular arithmetic, and thus requires modular reductions. Modular reduction operation is essentially computing the remainder of a division. Thus it is a costly operation unless

- A special modulus is chosen to ease the division, or
- A precomputation based on the chosen modulus is used.

The Barrett and the Montgomery algorithms are two modular reduction algorithms using precomputation. Because of the precomputation overhead, these algorithms are used if a large number of modular reductions need to be performed. Also, the Montgomery algorithm requires a domain transformation, while the Barrett algorithm does not. This domain transformation is thus a slight drawback for the Montgomery algorithm.

5.3 Multiplication in Prime Fields

The prime field $\mathbb{F}_p$ elements are represented by the set of the integers $\{0,1,2,\ldots,p-1\}$. Let a and b be two elements in $\mathbb{F}_p$. Let c be their product in $\mathbb{F}_p$. Then, c is defined as follows.

$$c = a \times b \bmod p\,.$$

As a result, the prime field multiplication needs two arithmetic operations:

- Integer multiplication, and
- Integer modular reduction.

The algorithms used in the modular multiplication of the integers will be studied in this section. However, the multiple precision representation, the addition, and the subtraction of the integers need to be be discussed first.

In practice, a hardware or software implementation supports a fixed w-bit word size. Each w-bit word stores an integer digit and integers are represented in the base $\beta = 2^w$. Let a be an integer in $\mathbb{F}_p$ and a_i be its ith digit. Then, the multiple precision representation for a is

$$a = (a_{n-1},\ldots,a_2,a_1,a_0)_\beta\,.$$

Naturally, the number of digits n in this representation must satisfy $p \le \beta^n$ so that all the integers in the set $\{0,1,2,\ldots,p-1\}$ can be represented.

To perform the integer addition $c = a+b$, the corresponding digits of a and b are added from the least to the most significant as follows.

$$(\varepsilon_{i+1},c_i) = a_i + b_i + \varepsilon_i, \qquad i = 0,1,2,\ldots \tag{5.1}$$

Here, $\varepsilon_0 = 0$ and ε_{i+1} is the carry due to the addition of the ith digits.

Similarly, the integer subtraction $c = a - b$ is performed as follows.

$$(\varepsilon_{i+1}, c_i) = a_i - b_i - \varepsilon_i, \qquad i = 0, 1, 2, \ldots \tag{5.2}$$

Here, $\varepsilon_0 = 0$ and ε_{i+1} is the borrow due to the subtraction of the ith digits.

As seen, these digit-by-digit operations involve carry and borrow propagations which can be handled in hardware easily. Also, the general purpose processors have always the instructions "add with carry" and "subtract with borrow", which are helpful for the carry and borrow propagations.

5.3.1 Integer Multiplication

The standard way of multiplying two integers is to multiply each digit in the first by each digit in the second and combine the resulting partial products. It is easy to see that this computation requires $\mathcal{O}(n^2)$ digit operations for n-digit integers. Algorithm 1 and Algorithm 2 illustrate two different implementations of the standard integer multiplication [6, 15].

Let β be the integer base. Algorithm 1 finds $a \times b$ using the fact that

$$d = a \times b = \textstyle\sum_{i=0}^{n-1} a_i b\, \beta^i.$$

Algorithm 1 computes $a_i b$ for each a_i, then appropriately shifts and combines the results. The inner loop starting at Step 5 scans the second operand digits b_j and computes $A \times b_j = a_i \times b_j$. The result is stored into two-digit integer (H, L) in Step 6, where H and L are the higher and lower digits respectively. The previous values of the higher digit H and the running product digit d_{i+j} are also added to (H, L). Note that (H, L) can hold the result in Step 6 without any overflow because the digits $A, b_j, H, d_{i+j} \leq \beta - 1$, and thus

$$A \times b_j + H + d_{i+j} \leq (\beta - 1)(\beta - 1) + 2(\beta - 1) < \beta^2.$$

Algorithm 1: Integer multiplication (by operand scanning)

```
INPUT: $n$-digit integers $a$ and $b$.
OUTPUT: $2n$-digit integer $d = a \times b$.
1. for $i = 0$ to $n-1$ do $d_i = 0$
2. for $i = 0$ to $n-1$ do
3.     $H = 0$
4.     $A = a_i$
5.     for $j = 0$ to $n-1$ do
6.         $(H, L) = A \times b_j + H + d_{i+j}$
7.         $d_{i+j} = L$
8.     $d_{i+n} = H$
9. return($d$)
```

Algorithm 2 computes the digits of $d = a \times b$ one by one, from the least significant to the most significant. Algorithm 2 uses the fact that

$$d = \sum_{k=0}^{2n-2} \beta^k \left(\sum_{i \in I} a_i b_{k-i} \right), \qquad I = \{ i \mid 0 \leq i, k-i < n \}.$$

For each k, the sum $\sum_{i \in I} a_i b_{k-i}$ is computed and stored into the three-digit integer (U,H,L) in Step 6, where U and L are the most and the least significant digits respectively. Step 7 determines the kth digit of the product d as $d_k = L$. Then, Step 8 removes the digit L by shifting (U,H,L) one digit right. The remaining more significant digits U and H are used to compute the more significant digits of the product d.

Algorithm 2: Integer multiplication (by product scanning)

```
INPUT: $n$-digit integers $a$ and $b$.
OUTPUT: $2n$-digit integer $d = a \times b$.
 1.  $(U,H,L) = (0,0,0)$
 2.  for $k = 0$ to $2n-2$ do
 3.      if $k < n$    $I = \{i \mid 0 \leq i \leq k\}$
 4.      if $k \geq n$    $I = \{i \mid n > i > k-n\}$
 5.      for every $i \in I$
 6.          $(U,H,L) += a_i \times b_{k-i}$
 7.      $d_k = L$
 8.      $(U,H,L) = (0,U,H)$
 9.  $d_{2n-1} = L$
10.  return($d$)
```

To compare the efficiencies of Algorithms 1 and 2, the inner loops of these algorithms must be considered. The inner loops of both the algorithms repeat n^2 times to perform n^2 different digit multiplications. The operations in the inner loop of Algorithm 1 are equivalent to

$$(H',L) = A \times b_j, \qquad (H,L) = (H',L) + (0,H) + (0,d_{i+j}), \qquad d_{i+j} = L.$$

These operations require four w-bit additions, two data reads (b_j, d_{i+j}), and one data write (d_{i+j}). The operations in the inner loop of Algorithm 2 are equivalent to

$$(H',L') = a_i \times b_{k-i}, \qquad (U,H,L) = (U,H,L) + (0,H',L').$$

These operations require three w-bit additions and two data reads (a_i, b_{k-i}). Also, note that the inner loops of the algorithms require multiprecision additions. These additions are performed as shown in (5.1).

Though Algorithm 1 is more straightforward to implement in hardware, Algorithm 2 is more advantageous in software. This is because Algorithm 2 requires fewer digit additions, data reads, and data writes. Here, it is assumed that the temporary variables (A, U, H, L, H', L') are held in the registers of the underlying processor; thus accessing them does not increase the data reads and writes.

5.3.2 Integer Squaring

Algorithm 3 computes the square of an integer. This algorithm is just a simplification of Algorithm 2 for the case that the multiplicands a and b are equal. Since $a = b$, the cross products satisfy $a_i b_j = a_j b_i = a_i a_j$. Thus, the number of the required digit products reduces roughly by half.

Algorithm 3 computes not all but half of the cross products using the fact

$$\sum_{\substack{i\in I\\ i\neq k-i}} a_i a_{k-i} = 2\sum_{\substack{i\in I\\ i>k-i}} a_i a_{k-i} = 2\sum_{\substack{i\in I\\ i<k-i}} a_i a_{k-i}$$

where $I = \{i \mid 0 \le i, k-i < n\}$. Note that this can also be written as follows

$$\sum_{\substack{i\in I\\ i\neq k/2}} a_i a_{k-i} = 2\sum_{\substack{i\in I\\ i>k/2}} a_i a_{k-i} = 2\sum_{\substack{i\in I\\ i<k/2}} a_i a_{k-i} \,.$$

Algorithm 3: Integer squaring

```
INPUT: $n$-digit integer $a$.
OUTPUT: $2n$-digit integer $d = a^2$.
 1. $(U,H,L) = (0,0,0)$
 2. for $k = 0$ to $2n-2$ do
 3.     if $k < n$   $I = \{i \mid 0 \le i < k/2\}$
 4.     if $k \ge n$   $I = \{i \mid n > i > k/2\}$
 5.     for every $i \in I$
 6.         $(U,H,L) += a_i \times a_{k-i}$
 7.     if $k$ is even     $(U,H,L) = 2(U,H,L) + a_{k/2}^2$
 8.     if $k$ is odd      $(U,H,L) = 2(U,H,L)$
 9.     $d_k = L$
10.     $(U,H,L) = (0,U,H)$
11. $d_{2n-1} = L$
12. return($d$)
```

5.3.3 Integer Modular Reduction

This section discusses the following methods for the reduction d mod p :

- The algorithms for moduli of special form
- The Barrett and the Montgomery algorithms using a precomputation based on the modulus p.

The output of the modular reduction is nothing else than the remainder of the division d/p. When the quotient calculation is omitted, the division turns into modular reduction. The multiple precision division for an arbitrary base β is a costly operation. References [10, 15] give a good discussion of the multiple precision division and Ref. [5] presents a multiple precision modular reduction algorithm based on division.

The computation $d \bmod p$ in the base $\beta = 2$ is rather straightforward. In this case, the integer d is reduced bit by bit modulo p. Let $2^m > p \geq 2^{m-1}$ and $p \leq d = (d_{k-1}, \ldots, d_1, d_0)_2$. Then, $d_{k-1}2^{k-1} > d_{k-1}2^{k-1-m}p \geq d_{k-1}2^{k-2}$ and d can be reduced as follows.

$$d = d - d_{k-1}2^{k-1-m}p.$$

To find $d \bmod p$, the bit reductions are performed iteratively until $d < p$. Also, $d \bmod p$ can be computed by using the integer $\hat{p} = 2^m \bmod p$. Then, $d_{k-1}2^{k-1} \equiv d_{k-1}2^{k-1-m}\hat{p} \bmod p$ and d can be reduced as follows.

$$d = (d_{k-2}, \ldots, d_0)_2 + d_{k-1}2^{k-1-m}\hat{p}.$$

Algorithm 4 implements the integer modular reduction using this method.

Algorithm 4: Bit level integer modular reduction

```
INPUT: Integers $d = (d_{k-1}, \ldots, d_0)_2$ and $\hat{p} = 2^m \bmod p$ where $2^m > p \geq 2^{m-1}$.
OUTPUT: $d \bmod p$.
1.  while $k > m$ do
2.      while $d_{k-1} \neq 0$ do
3.          $d = (d_{k-2}, \ldots, d_0)_2 + 2^{k-1-m}\hat{p}$
4.      $k = k - 1$
5.  return($d$)
```

5.3.3.1 Using Special Modulus

The commonly used base to represent the integers in processors is $\beta = 2^{32}$. Thus, it is easier to perform reduction modulo a prime number which can be written as a simple sum of the powers of 2 or 2^{32}, in software and hardware implementations. The following numbers are prime and have this property,

$$\begin{aligned}
&2^{192} - 2^{64} - 1 = \beta^6 - \beta^2 - 1,\\
&2^{224} - 2^{96} + 1 = \beta^7 - \beta^3 + 1,\\
&2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 = \beta^8 - \beta^7 + \beta^6 + \beta^3 - 1,\\
&2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 = \beta^{12} - \beta^4 - \beta^3 + \beta - 1,\\
&2^{521} - 1.
\end{aligned}$$

Here, $\beta = 2^{32}$. Fast modular reduction methods can be developed for these primes [20]. Consider the prime $p = 2^{192} - 2^{64} - 1$ as an example. For $\beta = 2^{32}$,

$$\begin{array}{lll}
\beta^6 \equiv \beta^2 + 1 \mod p & \beta^8 \equiv \beta^4 + \beta^2 \mod p & \beta^{10} \equiv \beta^4 + \beta^2 + 1 \mod p\\
\beta^7 \equiv \beta^3 + \beta \mod p & \beta^9 \equiv \beta^5 + \beta^3 \mod p & \beta^{11} \equiv \beta^5 + \beta^3 + \beta \mod p.
\end{array}$$

Let $d = (d_{11}, d_{10}, d_9, d_8, d_7, d_6, d_5, d_4, d_3, d_2, d_1, d_0)_\beta$. Then, the high digits of d can be reduced efficiently as follows.

$$\begin{aligned} d_7\beta^7 + d_6\beta^6 &= (0\,,0\,,d_7, d_6, d_7, d_6)_\beta\ , \\ d_9\beta^9 + d_8\beta^8 &= (d_9, d_8, d_9, d_8, 0\,, 0\,)_\beta\ , \\ d_{11}\beta^{11} + d_{10}\beta^{10} &= (d_{11}, d_{10}, d_{11}, d_{10}, d_{11}, d_{10})_\beta\ . \end{aligned}$$

Algorithm 5 implements this fast modular reduction method.

Algorithm 5: Integer modular reduction for $p = 2^{192} - 2^{64} - 1$

```
INPUT: Integer $d = (d_{11}, \ldots, d_0)_{2^{32}} < (2^{192} - 2^{64} - 1)^2$.
OUTPUT: $c = d \bmod (2^{192} - 2^{64} - 1)$.
  1.  Define the 6-digit integers in the base $\beta = 2^{32}$ :
      $e = (d_5, d_4, d_3, d_2, d_1, d_0)_\beta, \quad f = (0\,, 0\,, d_7, d_6, d_7, d_6)_\beta,$
      $g = (d_9, d_8, d_9, d_8, 0\,, 0\,)_\beta, \quad h = (d_{11}, d_{10}, d_{11}, d_{10}, d_{11}, d_{10})_\beta.$
  2.  $c = e + f + g + h \bmod (2^{192} - 2^{64} - 1)$
  3.  return($c$)
```

5.3.3.2 Barrett Modular Reduction

The Barrett method computes $c = d \bmod p$ for two integers d and p using a precomputation based on the chosen modulus [2]. The integer $c = d \bmod p$ is the remainder of the division d/p. Thus,

$$c = d - pq$$

for the quotient $q = \lfloor d/p \rfloor$. The Barrett method first finds an estimate of the quotient q using some precomputation. Let $\hat{q}$ denote this estimate. Then, the Barret method computes $c' = d - p\hat{q}$. As shown later in the text, $q - 2 \le \hat{q} \le q$. Thus, the Barrett method actually computes $c' = d - (q - \varepsilon)p = c + \varepsilon p$ where $\varepsilon \in \{0, 1, 2\}$. Thus, the subtraction of the modulus p from the final result one or two times may be needed for correction.

Quotient Estimation:

The Barrett method exploits the simple fact that

$$\frac{d}{p} = \left(\frac{2^k}{p}\right)\left(\frac{d}{2^{k'}}\right)\left(\frac{1}{2^{k-k'}}\right)$$

for the arbitrary integers k and k'. The divisions $2^k/p$ and $d/2^{k'}$ can be written in terms of their quotients and remainders as follows.

$$\frac{d}{p} = \left(\lfloor 2^k/p \rfloor + \frac{2^k \bmod p}{p}\right)\left(\lfloor d/2^{k'} \rfloor + \frac{d \bmod 2^{k'}}{2^{k'}}\right)\left(\frac{1}{2^{k-k'}}\right)\ .$$

Let $r^{(1)} = 2^k \bmod p$ and $r^{(2)} = d \bmod 2^{k'}$. Then, after some rearrangement,

$$\frac{d}{p} = \frac{\lfloor 2^k/p \rfloor \lfloor d/2^{k'} \rfloor}{2^{k-k'}} + \frac{\lfloor 2^k/p \rfloor r^{(2)}}{2^k} + \frac{\lfloor d/2^{k'} \rfloor r^{(1)}}{p2^{k-k'}} + \frac{r^{(1)} r^{(2)}}{p2^k}. \tag{5.3}$$

The Barrett algorithm estimates the quotient q of the division d/p as

$$q \approx \hat{q} = \left\lfloor \frac{\lfloor 2^k/p \rfloor \lfloor d/2^{k'} \rfloor}{2^{k-k'}} \right\rfloor, \tag{5.4}$$

i.e., the quotient of the first term in (5.3). Note that the divisions by powers of two in this estimation can be handled in software and hardware without any cost. However, the division $2^k/p$ must be precomputed for efficiency.

Estimation Error:

The quotient estimation in (5.4) will be accurate if the integers k and k' are chosen so that the last three terms in (5.3) are rational numbers less than one. In this case, the sum of the last three terms will be less than three. Let ε be the integer part of this sum. Then, $\varepsilon \le 2$ and $q - 2 \le \hat{q} \le q$.

In order that the last three terms in (5.3) are less than one, the denominators must be larger than the numerators. Then,

$$\begin{aligned}
&\lfloor 2^k/p \rfloor (d \bmod 2^{k'}) \le \lfloor 2^k/p \rfloor (2^{k'} - 1) < 2^k, \\
&\lfloor d/2^{k'} \rfloor (2^k \bmod p) \le \lfloor d/2^{k'} \rfloor (p-1) < p2^{k-k'}, \\
&(d \bmod 2^{k'})(2^k \bmod p) \le (2^{k'} - 1)(p-1) < p2^k.
\end{aligned}$$

The inequalities above always hold, if $2^{k'} \le p$, $d \le 2^k$, and $k' \le k$. Then, for $p \le d$, the parameters k and k' can be chosen as

$$k \ge \log_2 d, \qquad k' \le \log_2 p.$$

Barrett Algorithm:

Algorithm 6 implements the Barrett algorithm.

Algorithm 6: Barrett modular reduction

```
INPUT: The integers $d$ and $p$.
OUTPUT: $c = d \bmod p$.
 1.  Precompute $\hat{p} = \lfloor 2^k/p \rfloor$ where $k \ge \log_2 d$.
 2.  $u = \lfloor d/2^{k'} \rfloor$ where $k' \le \log_2 p$.
 3.  $\hat{q} = \lfloor \hat{p}u/2^{k-k'} \rfloor$
 4.  $c = d - \hat{q}p$
 5.  while($c \ge p$)    $c = c - p$
 6.  return $c$
```

Multiprecision Implementation:

Let $d = (d_{l-1}, \ldots, d_0)_\beta$ and $p = (p_{n-1}, \ldots, p_0)_\beta$ where $p \le d$ and β is a power of two. The integers k and k' can be chosen as

$$k = l \log_2 \beta \ge \log_2 d, \qquad k' = (n-1)\log_2 \beta \le \log_2 p.$$

Algorithm 7 implements the Barrett algorithm for $2^k = \beta^l$ and $2^{k'} = \beta^{n-1}$.

Algorithm 7: Multiprecision Barrett modular reduction

```
INPUT: Integers $d = (d_{l-1}, \ldots, d_0)_\beta$ and $p = (p_{n-1}, \ldots, p_0)_\beta > \beta^{n-1}$.
OUTPUT: $c = d \bmod p$.
 1. Precompute $\hat{p} = (\hat{p}_{l-n}, \ldots, \hat{p}_0)_\beta = \lfloor \beta^l / p \rfloor$.
 2. $u = (u_{l-n}, \ldots, u_0)_\beta = (d_{l-1}, \ldots, d_{n-1})_\beta$
 3. $v = \hat{p}u$
 4. $\hat{q} = (\hat{q}_{l-n}, \ldots, \hat{q}_0)_\beta = (v_{2(l-n)+1}, \ldots, v_{l-n+1})_\beta$
 5. $c = (c_n, \ldots, c_0)_\beta = d - \hat{q}p$
 6. while($c \ge p$)    $c = c - p$
 7. return($c$)
```

- Step 2 computes the integer $u = \lfloor d/\beta^{n-1} \rfloor$.
- Step 3 computes the product $v = \hat{p}u$, which can be approximated as

  $$v \approx v' = \textstyle\sum_{i+j \ge l-n-1} \hat{p}_i u_j \beta^{i+j}.$$

  Note that the error due to the ignored terms is

  $$v - v' = \textstyle\sum_{0 \le i+j \le l-n-2} \hat{p}_i u_j \beta^{i+j} \le \sum_{0 \le k \le l-n-2} (k+1)(\beta-1)^2 \beta^k.$$

  It can be shown that $v - v' \le \beta^{l-n-1}((l-n-1)\beta - l + n) + 1$. Moreover,

  $$v - v' < \beta^{l-n+1}$$

  for $\beta \ge (l-n-1)$.
- Step 4 finds the estimate $\hat{q} = \lfloor v/\beta^{l-n+1} \rfloor$. $\hat{q}$ can be approximated as $\lfloor v'/\beta^{l-n+1} \rfloor$. The resulting error will be less than one as shown below.

  $$\lfloor v/\beta^{l-n+1} \rfloor - \lfloor v'/\beta^{l-n+1} \rfloor \le 1.$$

- Step 5 finds ($d \bmod p + \varepsilon p$) where $\varepsilon = q - \hat{q}$. Since $p < \beta^n$ and ε is a small number, the result of Step 5 will not be more than n digits in the worst case. Thus, only the lower n digits of the product $\hat{q}p$ need to be computed in this step. Step 6 removes εp.

5.3.3.3 Montgomery Modular Reduction

The Montgomery modular reduction computes $d\theta^{-1} \bmod p$ for two integers d and p [13, 18]. Here, θ is preferably a power of two such that $\gcd(p,\theta) = 1$ and $p\theta > d$. The Montgomery method requires some precomputation and domain transformation.

The Montgomery method is used to reduce the products of the integers represented in the Montgomery residue domain. Let a' and b' be two integers. Let $c' = a'b' \bmod p$ be their modular product. In the Montgomery residue domain, these integers are represented by

$$a = a'\theta \bmod p\,, \qquad b = b'\theta \bmod p\,, \qquad c = c'\theta \bmod p\,.$$

Let $d = ab$ be the product of the integers in the residue domain. Then, the Montgomery modular reduction $d\theta^{-1} \bmod p$ yields their product in the residue domain c as shown below.

$$\begin{aligned} d\theta^{-1} \bmod p &= (a'\theta \bmod p)(b'\theta \bmod p)\theta^{-1} \bmod p \\ &= a'b'\theta \bmod p \\ &= c'\theta \bmod p \\ &= c. \end{aligned}$$

The Montgomery method computes $c = d\theta^{-1} \bmod p$ as follows.

$$c = \frac{d - (dp^{-1} \bmod \theta)p}{\theta} - \varepsilon p \tag{5.5}$$

where $d < p\theta$ and $\varepsilon \in \{0,1\}$. This computation leads to an efficient modular reduction algorithm when θ is a power of two and $p^{-1} \bmod \theta$ is precomputed.

The correctness of the Montgomery modular reduction method can be shown by using the Bezout's identity. Because θ and p are relatively prime,

$$\theta\hat{\theta} + p\hat{p} = \gcd(\theta, p) = 1$$

where $\hat{\theta} = \theta^{-1} \bmod p$ and $\hat{p} = p^{-1} \bmod \theta$. Then, $d = d\theta\hat{\theta} + dp\hat{p}$. Since $d < p\theta$,

$$\begin{aligned} d &= d\theta\hat{\theta} + dp\hat{p} \bmod p\theta \\ &= (d\theta\hat{\theta} \bmod p\theta) + (dp\hat{p} \bmod p\theta) + \varepsilon p\theta \end{aligned}$$

where $\varepsilon \in \{0,1\}$. Moreover, it can be written that

$$\begin{aligned} d &= (d\hat{\theta} \bmod p)\theta + (d\hat{p} \bmod \theta)p + \varepsilon p\theta \\ &= (d\theta^{-1} \bmod p)\theta + (dp^{-1} \bmod \theta)p + \varepsilon p\theta \\ &= c\theta + (dp^{-1} \bmod \theta)p + \varepsilon p\theta \end{aligned}$$

using the rules of modular arithmetic. See that Equation (5.5) can be obtained by rearranging the above equation.

Montgomery Algorithm and its Multiprecision Implementation:

Let $d = (d_{l-1},\dots,d_0)_\beta$ and $p = (p_{n-1},\dots,p_0)_\beta$ where β is a power of two. The integer θ can be chosen as $\theta = \beta^{l-n}$. Then, Equation (5.5) is given by

$$c = \left\lfloor \frac{d}{\beta^{l-n}} \right\rfloor - \left\lfloor \frac{(d\hat{p} \bmod \beta^{l-n})p}{\beta^{l-n}} \right\rfloor - \varepsilon p$$

where $\gcd(\beta,p) = 1$, $\hat{p} = p^{-1} \bmod \beta^{l-n}$, and $d < p\beta^{l-n}$.

Algorithm 8 implements the Montgomery algorithm.

Algorithm 8: Multiprecision Montgomery modular reduction

```
INPUT: Integers $d = (d_{l-1},\dots,d_0)_\beta$ and $p = (p_{n-1},\dots,p_0)_\beta$ such that $\gcd(\beta,p) = 1$
and $d < p\beta^{l-n}$.
OUTPUT: $c = d\beta^{-(l-n)} \bmod p$.
 1. Precompute $\hat{p} = (\hat{p}_{l-n-1},\dots,\hat{p}_0)_\beta = p^{-1} \bmod \beta^{l-n}$.
 2. $u = (u_{l-n-1},\dots,u_0)_\beta = d\hat{p} \bmod \beta^{l-n}$
 3. $v = up$
 4. $c = (d_{l-1},\dots,d_{l-n})_\beta - (v_{l-1},\dots,v_{l-n})_\beta$
 5. while($c \geq p$)    $c = c - p$
 6. return($c$)
```

- Step 2 computes the $(l-n)$-digit integer $u = d\hat{p} \bmod \beta^{l-n}$ as follows.

$$\begin{aligned} u &= (\textstyle\sum_{i+j<l-n} d_i\hat{p}_j\beta^{i+j}) \bmod \beta^{l-n} \\ &= \textstyle\sum_{i+j<l-n-1} d_i\hat{p}_j\beta^{i+j} + (\sum_{i+j=l-n-1} d_i\hat{p}_j \bmod \beta)\beta^{l-n-1}. \end{aligned}$$

- Step 3 computes the product $v = up$, which can be approximated as

$$v \approx v' = (\textstyle\sum_{l-n-2\leq i+j} u_i p_j \beta^{i+j}).$$

Note that the error due to the ignored terms is

$$v - v' = \textstyle\sum_{0\leq i+j\leq l-n-3} \hat{p}_i u_j \beta^{i+j} \leq \sum_{0\leq k\leq l-n-3} (k+1)(\beta-1)^2\beta^k .$$

It can be shown that $v - v' \leq \beta^{l-n-2}((l-n-2)\beta - l + n + 1) + 1$. Moreover,

$$v - v' < \beta^{l-n}$$

for $\beta \geq (l-n-2)$.

- Step 4 computes

$$c = \lfloor d/\beta^{l-n} \rfloor - \lfloor v/\beta^{l-n} \rfloor \approx \lfloor d/\beta^{l-n} \rfloor - \lfloor v'/\beta^{l-n} \rfloor .$$

Note that, when v is approximated by v', the error in c is less than one since $v - v' < \beta^{l-n}$.

- After Step 4, $c = d \bmod p + \varepsilon p$ for a small number ε. Step 5 removes εp.

5.4 Multiplication in Binary Extension Fields

The binary extension field $\mathbb{F}_{2^m}$ elements are represented by the set of the polynomials of degree less than m with coefficients in $\mathbb{F}_2$. That is

$$\mathbb{F}_{2^m} = \{a(x) \mid a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0,\ a_i \in \mathbb{F}_2\} .$$

Let $a(x)$ and $b(x)$ be two elements in $\mathbb{F}_{2^m}$. Let $c(x)$ be their product in $\mathbb{F}_{2^m}$. Then, $c(x)$ is defined as follows.

$$c(x) = a(x) \times b(x) \bmod \omega(x)$$

where $\omega(x)$ is a degree m irreducible polynomial over $\mathbb{F}_2$. As a result, the binary extension field multiplication needs two arithmetic operations:

- Polynomial multiplication over $\mathbb{F}_2$, and
- Polynomial modular reduction over $\mathbb{F}_2$.

The algorithms used in the modular multiplication of the polynomials over $\mathbb{F}_2$ will be studied in this section. However, the multiple precision representation, the addition, and the subtraction of the polynomials over $\mathbb{F}_2$ need to be discussed first.

Let a fixed w-bit word size be supported in a hardware or software implementation. Each w-bit word can store w polynomial coefficients since the polynomial coefficients are in $\mathbb{F}_2$ and represented by the integers $\{0,1\}$. Let $a(x)$ be a polynomial over $\mathbb{F}_2$ and a_i be its ith coefficient. Let $A_i = \sum_{k=0}^{w-1} a_{iw+k}x^k$. Then, each A_i is a w-coefficient polynomial stored in a single word and the multiple precision representation for $a(x)$ is

$$a(x) = A_{n-1}x^{(n-1)w} + \ldots + A_2 x^{2w} + A_1 x^w + A_0 . \quad (5.6)$$

In this representation, n is the number of the single word polynomials A_i. Naturally, n must satisfy the inequality $x^m \leq x^{wn}$ so that all the polynomials of degree less than m over $\mathbb{F}_2$ can be represented.

To perform the polynomial addition $c(x) = a(x) + b(x)$ and the polynomial subtraction $c(x) = a(x) - b(x)$, the corresponding coefficients of $a(x)$ and $b(x)$ must be added and subtracted in $\mathbb{F}_2$ respectively. The binary-valued elements of $\mathbb{F}_2$ $\{0,1\}$ are added or subtracted modulo 2. As a result, the addition and the subtraction in $\mathbb{F}_2$ are just equivalent to XOR operation. Thus, $c(x) = a(x) \pm b(x)$ are performed by bitwise XORing the corresponding words as follows.

$$C_i = A_i \text{ XOR } B_i, \qquad i = 0,1,2,\ldots \tag{5.7}$$

The bitwise XOR operation is ubiquitously found in hardware and software implementations.

5.4.1 Polynomial Multiplication over $\mathbb{F}_2$

Let $d(x) = a(x)b(x)$. If $a(x)$ and $b(x)$ are represented as shown in (5.6),

$$\begin{aligned} d(x) &= (\textstyle\sum_{i=0}^{n-1} A_i x^{iw}) b(x) \\ &= \textstyle\sum_{i=0}^{n-1} (\sum_{k=0}^{w-1} a_{iw+k} x^k) x^{iw} b(x) \\ &= \textstyle\sum_{k=0}^{w-1} x^k \sum_{i=0}^{n-1} a_{iw+k} x^{iw} (\sum_{j=0}^{n-1} B_j x^{jw}) \\ &= \textstyle\sum_{k=0}^{w-1} x^k \sum_{i=0}^{n-1} a_{iw+k} \sum_{j=0}^{n-1} B_j x^{(i+j)w}. \end{aligned}$$

This discrete summation formula leads to the right-to-left and the left-to-right multiplication methods implemented in Algorithms 9 and 10 for the polynomials over $\mathbb{F}_2$, respectively.

Algorithm 9: Right-to-left comb method

```
INPUT: Polynomials over $\mathbb{F}_2$ $a(x)$ and $b(x)$ of degree less than $m \leq nw$.
OUTPUT: $d(x) = a(x)b(x)$.
1. for $i = 0$ to $2n-1$ do $D_i = 0$
2. for $k = 0$ to $w-1$
3.     for $i = 0$ to $n-1$
4.         if the $k$th bit of $A_i$ is 1
5.             for $j = 0$ to $n$
6.                 $D_{i+j} = D_{i+j} + B_j$
7.     if $k \neq w-1$ then $b(x) = \sum_{l=0}^{n} B_l x^{lw} = xb(x)$
8. return($d(x)$)
```

Algorithm 10: Left-to-right comb method

```
INPUT: Polynomials over $\mathbb{F}_2$ $a(x)$ and $b(x)$ of degree less than $m \le nw$.
OUTPUT: $d(x) = a(x)b(x)$.
1. for $i = 0$ to $2n-1$ do $D_i = 0$
2. for $k = w-1$ downto 0
3.         for $i = 0$ to $n-1$
4.                 if the $k$th bit of $A_i$ is 1
5.                         for $j = 0$ to $n-1$
6.                                 $D_{i+j} = D_{i+j} + B_j$
7.         if $k \neq 0$ then $d(x) = \sum_{l=0}^{2n-1} D_l x^{lw} = xd(x)$
8. return($d(x)$)
```

Algorithm 11 is a faster implementation of the left-to-right comb method given in Algorithm 10. However, this implementation requires more memory. Algorithm 11 computes all the possible products $b(x)u(x)$ where $u(x)$ is a polynomial of degree less than four and stores the resulting polynomials into the local variable space as a lookup table.

Algorithm 11: Left-to-right comb method with 4-bit window

```
INPUT: Polynomials over $\mathbb{F}_2$ $a(x)$ and $b(x)$ of degree less than $m \le nw - 3$.
OUTPUT: $d(x) = a(x)b(x)$.
1. Compute $f(u(x)) = b(x)u(x)$ for all $u(x)$ with $\deg(u(x)) < 4$.
2. for $i = 0$ to $2n-1$ do $D_i = 0$
3. for $k = 4\lfloor (w-1)/4 \rfloor$ downto 0 by 4
4.         for $i = 0$ to $n-1$
5.                 $u(x) = \lfloor A_i / x^k \rfloor \bmod x^4$
6.                 $b'(x) = \sum_{l=0}^{n-1} B'_l x^{lw} = f(u(x))$
7.                 for $j = 0$ to $n-1$
8.                         $D_{i+j} = D_{i+j} + B'_j$
9.         if $k \neq 0$ then $d(x) = \sum_{l=0}^{2n-1} D_l x^{lw} = x^4 d(x)$
10. return($d(x)$)
```

Note that the polynomial $u(x)$ has $2^4 = 16$ different possible values and the product $b(x)u(x)$ has $m+3$ coefficients. Thus, the required memory space for the lookup table is $16(m+3)$ bits. Algorithm 11 multiplies each four consecutive polynomial terms of $a(x)$ by $b(x)$ using the lookup table. The window size four can be increased, but then a larger lookup table will be needed and the overhead of the lookup table computation will increase.

5.4.2 Polynomial Squaring over $\mathbb{F}_2$

Let $a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0$ be a polynomial over $\mathbb{F}_2$. The square of a polynomial is given by

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} + \sum_{0\le j<i<m} \underbrace{2a_{i+j}x^{i+j}}_{0} .$$

As shown above, multiplication by 2 yields zero result in a characteristic 2 field. Thus, the cross products are zero and

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} = a_{m-1}x^{2(m-1)} + \cdots + a_1x^2 + a_0 .$$

Algorithm 12 computes the square of the polynomials over $\mathbb{F}_2$ where the word size w is divisible by 8. This algorithm precomputes and stores the square of all possible polynomials of degree less than 8 in a lookup table. Then, it computes the square of each consecutive eight terms of the input polynomial using the lookup table. The lookup table contains 2^8 polynomials of degree less than 16, and thus is of size 512 bytes.

Algorithm 12: Squaring of Polynomials over $\mathbb{F}_2$ where $8 \mid w$

```
INPUT: A polynomial over $\mathbb{F}_2$ $a(x)$ of degree less than $m \le nw$.
OUTPUT: $d(x) = a(x)^2$.
 1.  Precompute $f(u(x)) = u(x)^2$ for all $u(x)$ with $\deg(u(x)) < 8$.
 2.  for $i = 0$ to $n-1$
 3.      $C_{2i} = 0$
 4.      for $k = 8\lfloor (w/2-1)/8 \rfloor$ downto 0 by 8
 5.          $u(x) = \lfloor A_i/x^k \rfloor \bmod x^8$
 6.          $C_{2i} = C_{2i} + f(u(x))x^{2k}$
 7.      $C_{2i+1} = 0$
 8.      for $k = 8\lfloor (w-1)/8 \rfloor$ downto $8\lfloor w/16 \rfloor$ by 8
 9.          $u(x) = \lfloor A_i/x^k \rfloor \bmod x^8$
10.          $C_{2i+1} = C_{2i+1} + f(u(x))x^{2k}$
11.  return($d(x)$)
```

5.4.3 Polynomial Modular Reduction over $\mathbb{F}_2$

This section discusses the following methods for the modular reduction of the polynomials $d(x) \bmod \omega(x)$:

- The algorithms for moduli of special form
- The Barrett and the Montgomery algorithms using a precomputation based on the modulus $\omega(x)$.

In general, $d(x) \bmod \omega(x)$ over $\mathbb{F}_2$ can be performed as follows.

$$d(x) = d(x) + d_k x^{k-m} \omega(x)$$

where $k = \deg(d(x))$ and $m = \deg(\omega(x))$. To find $d(x) \bmod \omega(x)$, the coefficient reductions are performed iteratively until $d(x) < \omega(x)$. Algorithm 13 implements the integer modular reduction using this method.

Algorithm 13: Polynomial modular reduction over $\mathbb{F}_2$

```
INPUT: Polynomials $d(x)$ and $\omega(x)$ where $\deg(d(x)) = k$ and $\deg(\omega(x)) = m$.
OUTPUT: $d(x) \bmod \omega(x)$.
1.  while $k > m$ do
2.      if $d_k \neq 0$ do
3.          $d(x) = d(x) + x^{k-m}\omega(x)$
4.      $k = k - 1$
5.  return($d(x)$)
```

5.4.3.1 Using Special Modulus

It is easy to see that Algorithm 13 can be optimized when the modulus $\omega(x)$ is a sparse polynomial. In practice, $\omega(x)$ is used to construct the field $\mathbb{F}_{2^m}$ and must be irreducible. Irreducible polynomials with the minimum number of terms are trinomials and pentanomials. A trinomial is a polynomial with only three terms, while a pentanomial is a polynomial with only five terms. A trinomial or a pentanomial always exists for any field size $m < 1000$ [9].

The following irreducible trinomials and pentanomials are recommended in the FIPS 186-2 standard by NIST:

$$\begin{aligned}
&x^{163} + x^7 + x^6 + x^3 + 1\ ,\\
&x^{233} + x^{74} + 1\ ,\\
&x^{283} + x^{12} + x^7 + x^5 + 1\ ,\\
&x^{409} + x^{87} + 1\ ,\\
&x^{571} + x^{10} + x^5 + x^2 + 1\ .
\end{aligned}$$

The general form $\omega(x) = x^m + x^{m_1} + x^{m_2} + x^{m_3} + 1$ can be assumed for trinomials and pentanomials. Algorithm 14 performs fast modular reduction for a modulus in this special form.

Algorithm 14: Polynomial modular reduction for pentanomials

```
INPUT: Polynomials $d(x)$ and $\omega(x) = x^m + x^{m_1} + x^{m_2} + x^{m_3} + 1$.
OUTPUT: $d(x) \bmod \omega(x)$.
 1.  Set $\kappa = \lfloor m/w \rfloor$ and $\kappa_i = \lfloor (m - m_i)/w \rfloor$.
 2.  Set $\lambda = m \bmod w$ and $\lambda_i = (m - m_i) \bmod w$.
 3.  for $i = \lfloor \deg(d(x))/w \rfloor$ downto $n$
 4.      $D_{i-\kappa_1}$ += $D_i \gg \lambda_1$,    $D_{i-\kappa_1-1}$ += $D_i \ll (w - \lambda_1)$
 5.      $D_{i-\kappa_2}$ += $D_i \gg \lambda_2$,    $D_{i-\kappa_2-1}$ += $D_i \ll (w - \lambda_2)$
 6.      $D_{i-\kappa_3}$ += $D_i \gg \lambda_3$,    $D_{i-\kappa_3-1}$ += $D_i \ll (w - \lambda_3)$
 7.      $D_{i-\kappa}$ += $D_i \gg \lambda$ ,    $D_{i-\kappa-1}$ += $D_i \ll (w - \lambda)$
 8.  $i = n - 1$
 9.  $D_{i-\kappa_1}$ += $D_i \gg \lambda_1$,  if($i > \kappa_1$) then $D_{i-\kappa_1-1}$ += $D_i \ll (w - \lambda_1)$
10.  $D_{i-\kappa_2}$ += $D_i \gg \lambda_2$,  if($i > \kappa_2$) then $D_{i-\kappa_2-1}$ += $D_i \ll (w - \lambda_2)$
11.  $D_{i-\kappa_3}$ += $D_i \gg \lambda_3$,  if($i > \kappa_3$) then $D_{i-\kappa_3-1}$ += $D_i \ll (w - \lambda_3)$
12.  $D_{i-\kappa}$ += $D_i \gg \lambda$ ,  if($i > \kappa$ ) then $D_{i-\kappa-1}$ += $D_i \ll (w - \lambda)$
13.  return($d(x)$)
```

This algorithm uses the following equivalance relations for fast modular reduction.

$$\begin{aligned} x^m &\equiv x^{m_1} + x^{m_2} + x^{m_3} + 1 \bmod \omega(x) , \\ 1 &\equiv x^{-(m-m_1)} + x^{-(m-m_2)} + x^{-(m-m_3)} + x^{-m} \bmod \omega(x) , \\ 1 &\equiv x^{-\kappa_1 w - \lambda_1} + x^{-\kappa_2 w - \lambda_2} + x^{-\kappa_3 w - \lambda_3} + x^{-\kappa w - \lambda} \bmod \omega(x) . \end{aligned}$$

Here, the parameters $\kappa = \lfloor m/w \rfloor$, $\kappa_i = \lfloor (m - m_i)/w \rfloor$, $\lambda = m \bmod w$, and $\lambda_i = (m - m_i) \bmod w$.

5.4.3.2 Barrett Modular Reduction

The Barrett method for integers can be adapted to the polynomials over $\mathbb{F}_2$ to compute $c(x) = d(x) \bmod \omega(x)$ efficiently [7].

Quotient Estimation:

For the arbitrary integers k and k', the following equality always holds

$$\frac{d(x)}{\omega(x)} = \left(\frac{x^k}{\omega(x)} \right) \left(\frac{d(x)}{x^{k'}} \right) \left(\frac{1}{x^{k-k'}} \right) .$$

This equality leads to a result similar to (5.3)

$$\begin{aligned}\frac{d(x)}{\omega(x)} = \frac{\lfloor x^k/\omega(x)\rfloor \lfloor d(x)/x^{k'}\rfloor}{x^{k-k'}} + \frac{\lfloor x^k/\omega(x)\rfloor r^{(2)}(x)}{x^k} \\ + \frac{\lfloor d(x)/x^{k'}\rfloor r^{(1)}(x)}{\omega(x)x^{k-k'}} + \frac{r^{(1)}(x)r^{(2)}(x)}{\omega(x)x^k}\end{aligned} \tag{5.8}$$

where $r^{(1)}(x) = x^k \bmod \omega(x)$ and $r^{(2)}(x) = d(x) \bmod 2^{k'}$.

Then, the quotient $q(x)$ of the division $d(x)/\omega(x)$ is estimated as

$$q(x) \approx \hat{q}(x) = \left\lfloor \frac{\lfloor x^k/\omega(x)\rfloor \lfloor d(x)/x^{k'}\rfloor}{x^{k-k'}} \right\rfloor . \tag{5.9}$$

Note that this estimation for polynomials is the same as the one for integers in (5.4), except the powers of two are replaced with the powers of x.

Estimation Error:

The quotient estimation in (5.9) will be exact, if the integers k and k' are chosen so that the last three terms in (5.3) are rational functions whose denominator degrees are greater than their numerator degrees. For this case, the quotients of the last three terms in (5.3) are zero and the quotient of the first term $\hat{q}(x) = \lfloor d(x)/\omega(x)\rfloor = q(x)$.

The denominators of the last three terms in (5.3) are greater than their numerators, if

$$\begin{aligned}\deg(\lfloor x^k/\omega(x)\rfloor) + \deg(r^{(2)}(x)) &< \deg(x^k)\,,\\ \deg(\lfloor d(x)/x^{k'}\rfloor) + \deg(r^{(1)}(x)) &< \deg(\omega(x)) + \deg(x^{k-k'})\,,\\ \deg(r^{(1)}(x)) + \deg(r^{(2)}(x)) &< \deg(\omega(x)) + \deg(x^k)\,.\end{aligned}$$

Let $\deg(d(x)) \geq \deg(\omega(x))$. The inequalities above always hold, if

$$k \geq \deg(d(x))\,, \qquad k' \leq \deg(\omega(x))\,.$$

Barrett Algorithm:

Algorithm 15 implements the Barrett algorithm. This algorithm is very similar to Algorithm 6. However, the powers of two are replaced with the powers of x. Also, the final correction step after Step 4 is omitted since the quotient estimation is exact.

Algorithm 15: Barrett modular reduction in $\mathbb{F}_2[x]$

```
INPUT: Polynomials over $\mathbb{F}_2$ $d(x)$ and $\omega(x)$.
OUTPUT: $c(x) = d(x) \bmod \omega(x)$.
  1. Precompute $\hat{\omega}(x) = \lfloor x^k/\omega(x)\rfloor$ where $k \geq \deg(d(x))$.
  2. $u(x) = \lfloor d(x)/x^{k'}\rfloor$ where $k' \leq \deg(\omega(x))$.
```

```
3. $\hat{q}(x) = \lfloor \hat{\omega}(x)u(x)/x^{k-k'} \rfloor$
4. $c(x) = d(x) + \hat{q}(x)\omega(x)$
5. **return** $c(x)$
```

Multiprecision Implementation:

The multiprecision Barrett implementation for integers in Algorithm 6 can be adapted for polynomials over $\mathbb{F}_2$ simply by replacing the powers of two with the powers of x.

Let $d(x)$ and $\omega(x)$ be polynomials such that $\deg(d(x)) < lw$ and $(n-1)w < \deg(\omega(x)) \leq nw$. Then, the integers k and k' in the Barrett method can be chosen as

$$k = lw \geq \deg(d(x)), \qquad k' = (n-1)w \leq \deg(\omega(x)).$$

to compute $d(x) \bmod \omega(x)$. Algorithm 16 gives the resulting Barrett algorithm using the notation in (5.6).

Algorithm 16: Multiprecision Barrett modular reduction in $\mathbb{F}_2[x]$

INPUT: $d(x)$ and $\omega(x)$ in $\mathbb{F}_2[x]$ where $d(x) < x^{lw}$ and $x^{(n-1)w} < \omega(x) \leq x^{nw}$.
OUTPUT: $c(x) = d(x) \bmod \omega(x)$.

```
1. Precompute $\hat{\omega}(x) = \sum_{i=0}^{l-n} \hat{\Omega}_i x^{iw} = \lfloor x^{lw}/\omega(x) \rfloor$.
2. $u(x) = \sum_{i=0}^{l-n} U_i x^{iw} = \lfloor d(x)/x^{(n-1)w} \rfloor$
3. $v(x) = \sum_{l-n \leq i+j \leq 2(l-n)} \hat{\Omega}_i U_j x^{i+j}$
4. $\hat{q}(x) = \sum_{i=0}^{l-n} \hat{Q}_i x^{iw} = \lfloor v/x^{l-n+1} \rfloor$
5. $c(x) = \sum_{i=0}^{n-1} D_i x^i + \sum_{i+j<n} \hat{Q}_i \Omega_j x^{i+j} \bmod x^{nw}$
6. **return**$(c(x))$
```

Note that only the required terms of $v(x)$ in Step 3 are computed. But this does not cause any approximation error since there is no carry propagation in the polynomial arithmetic. Step 5 is performed modulo x^{nw} since the quotient estimation is exact, and thus $c(x) = d(x) \bmod \omega(x) < x^{nw}$ in this step.

Algorithm 17 illustrates a w-bit Barrett modular reduction scheme presented in the work in [7]. In this scheme,

$$k = \deg(\omega(x)) + w - 1, \qquad k' = \deg(\omega(x)),$$

and $\lfloor d(x)/\omega(x) \rfloor < x^w$.

Algorithm 17: w-bit Barrett modular reduction in $\mathbb{F}_2[x]$

INPUT: $d(x)$ and $\omega(x)$ in $\mathbb{F}_2[x]$ such that $nw \geq \deg(\omega(x)) > (n-1)w$ and $\lfloor d(x)/\omega(x) \rfloor < x^w$.
OUTPUT: $c(x) = d(x) \bmod \omega(x)$.

```
1. Precompute $Q^{(1)} = \lfloor x^k/\omega(x) \rfloor$ where $k = \deg(\omega(x)) + w - 1$.
2. Find $Q^{(2)} = \lfloor (D_n x^w + D_{n-1})/x^{k' \bmod w} \rfloor$ where $k' = \deg(\omega(x))$.
3. $\hat{Q} = \lfloor Q^{(1)} Q^{(2)}/x^{w-1} \rfloor$
4. $c(x) = \sum_{i=0}^{n-1} D_i + \sum_{i=0}^{n-1} \hat{Q}\Omega_i \bmod x^{nw}$
5. return $c(x)$
```

5.4.3.3 Montgomery Modular Reduction

The analog of the Montgomery modular reduction for polynomials in $\mathbb{F}_2[x]$ is proposed in [12]. The Montgomery modular reduction for polynomials is given by $d(x)\theta^{-1}(x) \bmod \omega(x)$ where $\gcd(\omega(x), \theta(x)) = 1$ and $\omega(x)\theta(x) > d(x)$. For an efficient computation, $\theta(x)$ is chosen as a power of x preferably.

The Montgomery reduction $c(x) = d(x)\theta^{-1}(x) \bmod \omega(x)$ is given by

$$c(x) = \frac{d(x) + (d(x)\omega(x)^{-1} \bmod \theta(x))\omega(x)}{\theta(x)} \tag{5.10}$$

where $d(x) < \omega(x)\theta(x)$. This computation leads to an efficient modular reduction algorithm when $\theta(x)$ is a power of x and $\omega(x)^{-1} \bmod \theta(x)$ is precomputed.

Equation (5.10) is similar to the Montgomery computation in (5.5) given for integers, except, there is no need for an extra subtraction with modulus. This is because no carry propagation occurs in the polynomial arithmetic. Thus, $\varepsilon = 0$ in the following equation is obtained by using the Bezout's identity.

$$\begin{aligned} d(x) &= d(x)\theta(x)\hat{\theta}(x) + d(x)\omega(x)\hat{\omega}(x) \bmod \omega(x)\theta(x) \\ &= (d(x)\theta(x)\hat{\theta}(x) \bmod \omega(x)\theta(x)) + \\ &\quad (d(x)\omega(x)\hat{\omega}(x) \bmod \omega(x)\theta(x)) + \varepsilon\omega(x)\theta(x) . \end{aligned}$$

As a result, a derivation similar to the integer case yields Equation (5.10).

Montgomery Algorithm:

Let $d(x)$ and $\omega(x)$ be polynomials in $\mathbb{F}_2[x]$ such that $d(x) < x^{w(l-n)}\omega(x)$ and $\gcd(\omega(x), x) = 1$.

The polynomial θ can be chosen as $\theta = x^{w(l-n)}$. Then, Equation (5.10) is given by

$$c(x) = \left\lfloor \frac{d(x)}{x^{w(l-n)}} \right\rfloor + \left\lfloor \frac{(d(x)\hat{\omega}(x) \bmod x^{w(l-n)})\omega(x)}{x^{w(l-n)}} \right\rfloor$$

where $\hat{\omega}(x) = \omega(x)^{-1} \bmod x^{w(l-n)}$.

5.5 Multiplication in General Extension Fields

The extension field $\mathbb{F}_{p^m}$ elements are represented by the set of the polynomials of degree less than m with coefficients in $\mathbb{F}_p$. That is

$$\mathbb{F}_{p^m} = \{a(x) \mid a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0,\ a_i \in \mathbb{F}_p\}\,.$$

Let $a(x)$ and $b(x)$ be two elements in $\mathbb{F}_{p^m}$. Let $c(x)$ be their product in $\mathbb{F}_{p^m}$. Then, $c(x)$ is defined as follows.

$$c(x) = a(x) \times b(x) \bmod \omega(x)$$

where $\omega(x)$ is a degree m irreducible polynomial over $\mathbb{F}_p$. As a result, the extension field multiplication needs two arithmetic operations:

- Polynomial multiplication over $\mathbb{F}_p$, and
- Polynomial modular reduction over $\mathbb{F}_p$.

The algorithms used in the modular multiplication of the polynomials over $\mathbb{F}_p$ will be studied in this section. However, the multiple precision representation, the addition, and the subtraction of the polynomials over $\mathbb{F}_p$ need to be discussed first.

Let a fixed w-bit word size be supported in a hardware or software implementation. Each w-bit word can store a single polynomial coefficient in $\mathbb{F}_p$, if $p < 2^w$. Let $a(x)$ be a polynomial over $\mathbb{F}_p$ and a_i be its ith coefficient. Then, each a_i is an integer stored in a single word and $a(x)$ is represented by an m-word array.

To perform the polynomial addition $c(x) = a(x) + b(x)$ or the polynomial subtraction $c(x) = a(x) - b(x)$, the corresponding coefficients of $a(x)$ and $b(x)$ are added or subtracted in $\mathbb{F}_p$ respectively. The coefficient additions and subtractions can be handled by single-word addition and subtraction operations ubiquitously found in the hardware and software implementations.

The previous section focuses on arithmetic in binary extension fields $\mathbb{F}_{2^m}$, which is a special case of the general extension field $\mathbb{F}_{p^m}$. The binary extension fields are preferred in hardware implementations due to the fact that subfield elements are easily representable using the signals logic zero and logic one. Also, the binary circuit technology makes the implementation of arithmetic operations rather straightforward. The addition and subtraction in the binary extension field can be performed simply by XOR operation and the multiplication involves shift and XOR operations.

Because the bit operations are slower in the general purpose processors, binary extension fields are not so great from the software point of view. The general purpose processors perform word level operations faster. Thus, some special classes of $\mathbb{F}_{p^m}$ called OEF are proposed to exploit this fast word level operation capability [1].

Let w denote the word size supported by the underlying system. An optimal extension field (OEF) is a finite field $\mathbb{F}_{p^m}$ where

- $p = 2^{w-1} \pm \alpha$ is a pseudo-Mersenne prime such that $\log_2 \alpha \leq \lfloor \frac{1}{2} w \rfloor$.
- An irreducible bionomial $\omega(x) = x^m - \lambda$ exists over $\mathbb{F}_p$.

In an OEF, elements are represented as degree $m-1$ polynomials as follows:

$$a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$$

where $a_i \in \mathbb{F}_p$. Addition of the two elements $a(x)$ and $b(x)$ is given by

$$a(x) + b(x) = \sum_{i=0}^{m-1} c_i x^i,$$

where $c_i = (a_i + b_i) \bmod p$. To add two OEF elements we need at the most m coefficient subtractions where p is the subtrahend. Subtraction is done similarly. Choosing the prime p close to but smaller that the word size of the underlying hardware architecture makes it possible to use the efficient integer arithmetic instructions supported by the hardware. With such a choice, the result of a coefficient multiplication will fit into a double word, where it can be accessed and reduced efficiently.

5.5.1 Field Multiplication in OEF

The two steps of the field multiplication in OEF are as follows:

- The OEF elements $a(x)$ and $b(x)$ are multiplied.

  $$d(x) = a(x)b(x) = d_{2m-2}x^{2m-2} + \cdots + d_1 x + d_0$$

  where $d_i \in \mathbb{F}_p$. The polynomial $d(x)$ is calculated by m^2 coefficient multiplications and $(m-1)^2$ coefficient additions.
- The reduction $c(x) = d(x) \bmod \omega(x)$ is performed where $\omega(x) = x^m - \lambda$ is an irreducible binomial over $\mathbb{F}_p$. Since the binomial $\omega(x)$ has only two terms, reduction with $\omega(x)$ can be done efficiently. The terms of $d(x)$ with degree greater than $m-1$ can be given by $d_{m+i}x^{m+i}$ for $i \geq 0$. These terms can be reduced by

  $$d_{m+i}x^{m+i} = \lambda d_{m+i}x^i \bmod \omega(x)$$

  for $i = 0, 1, \cdots, m-2$.

Since the degree of $d(x)$ is at most $2m-2$, we need at most $m-1$ multiplications by λ and $m-1$ coefficient additions to obtain the reduced polynomial $c(x)$ where

$$c(x) = d_{m-1}x^{m-1} + [\lambda d_{2m-2} + d_{m-1}]\, x^{m-2} + \cdots + \\ [\lambda d_{m+1} + d_1]\, x + [\lambda d_m + d_0] \bmod\ \omega(x).$$

The following algorithm integrates the reduction into the multiplication steps without focusing on the coefficient arithmetic operations.

Algorithm 18: OEF Modular Multiplication Algorithm

```
INPUT: OEF elements $a(x), b(x)$ with degree at most $m-1$. $\omega(x) = x^m - \lambda$.
OUTPUT: $c(x) = a(x)b(x) \bmod \omega(x)$.
1. for $i = 0$ to $m-1$ do $c_i = 0$
2. for $i = 0$ to $m-1$
3.     for $j = 0$ to $m-1$
4.         if $i + j \le m-1$ then $c_{i+j} = c_{i+j} + b_i a_j$
5.         else $c_{i+j-m} = c_{i+j-m} + b_i a_j\ w$
6. return $c(x)$
```

In Step 4 and Step 5 of Algorithm 18, we are performing coefficient multiplications and additions. If we skip the coefficient addition operation for $i + j = 0$ in these steps, we end up with $(m-1)^2$ coefficient additions. The total number of coefficient multiplications is $m^2 + m + 1$ where $m - 1$ of them come from the multiplication by λ. When $\omega(x)$ is selected as $\omega(x) = x^w - 2$, the coefficient multiplications by λ become simple right shift operations which can be implemented very fast. OEF's with this optimization are called Type II OEF's.

5.5.2 Coefficient Multiplication and Reductions

The coefficient multiplications and reductions can be calculated efficiently when $p = 2^{w-1} \pm \alpha$ is a pseudo-Mersenne prime not exceeding the word boundary and α is a small number. The result of the coefficient multiplication can be stored in a double word before reduction is performed. The reduction operation will reduce the result allowing it to fit into a single word. Algorithms that perform this reduction are reported in the literature. Algorithm 19 performs such a reduction operation where the α term is fixed to a negative integer.

Algorithm 19: Coefficient Reduction Algorithm

```
INPUT: $p = 2^{w-1} - \alpha$. Coefficient $c < p^2$.
OUTPUT: $c \bmod p$.
1. $q_0 = \lfloor c/2^{w-1} \rfloor$, $r_0 = c - q_0 2^{w-1}$
2. $r = r_0$, $i = 0$
3. while $q_i > 0$
4.     $q_{i+1} = \lfloor q_i \alpha / 2^{w-1} \rfloor$
5.     $r_{i+1} = q_i \alpha - q_{i+1} 2^{w-1}$
6.     $i = i + 1$, $r = r + r_i$
7. while $r \ge p$   do $r = r - p$
8. return $r$
```

In Step 1 of Algorithm 19, q_0 is initialized with the upper word and r_0 is initialized with the lower word of the input c. We want to reduce the upper word in one big step by taking out $q_0 2^{w-1}$. But by doing so we have taken out an extra $q_0 c$ value. We need to add this value back to the remainder. In Steps 5 and 6, we can see this effort. But, before adding this value back we further reduce it with in Steps 4 and 5. Because α is small, Step 4 is executed at the most twice. If α is selected as 1 the multiplications in Steps 3 and 4 become trivial. An OEF that supports this optimization is named as Type I.

5.6 Karatsuba–Ofman Algorithm

In this section, the fast multiplication method Karatsuba–Ofman is discussed for polynomials. This algorithms can also be used in the multiplication of large integers. In this case, x can be thought as the radix value in the multidigit representation of the integers.

Let $a_0 + a_1 x$ and $b_0 + b_1 x$ be two polynomials over a ring $\mathbb{R}$. As seen below, their multiplication using the schoolbook method

$$(a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2$$

needs the computation of four ring products. The Karatsuba method performs this multiplication by computing only three ring products as follows

$$\begin{aligned}(a_0 + a_1 x)(b_0 + b_1 x) &= a_0 b_0 + a_1 b_1 x^2 + [a_0 b_0 + a_1 b_1 + (a_0 - a_1)(b_1 - b_0)]x \\ &= a_0 b_0 (1 + x) + a_1 b_1 (x + x^2) + (a_0 - a_1)(b_1 - b_0)x.\end{aligned}$$

This method can be generalized for arbitrary degree polynomials. Let $y = x^n$. Let $a_i(x)$ and $b_i(x)$ be polynomials with degree at the most $n-1$. Then,

$$(a_0(x) + a_1(x)y)\,(b_0(x) + b_1(x)y) \tag{5.11}$$

is a product of the polynomials with degree at most $2n-1$ and can be computed with the Karatsuba method using the following three half-sized products

$$a_0(x)b_0(x), \qquad a_1(x)b_1(x), \qquad (a_0(x) - a_1(x))(b_1(x) - b_0(x)). \tag{5.12}$$

Here, $a_i(x)$ and $b_i(x)$ are the coefficients of the linear polynomials in y in the ring $\mathbb{R}[x]$.

As seen, the Karatsuba method computes a product from three half-sized products. In the same fashion, it computes each of these half-sized products from three quarter-sized products. This process goes recursively. When the products get very small, the recursion stops and these small products are computed by the schoolbook method. This recursive computation constitutes a multiplication method asymptotically faster than the $\mathcal{O}(n^2)$ schoolbook method.

5.6.1 Complexity

It can be shown that the Karatsuba multiplication is $\mathscr{O}(n^{1.58})$ [10]. Let $T(n)$ denote the complexity of the multiplying polynomials with degree $n-1$. Then, the complexity of the multiplying polynomials with degree $2n-1$ is $T(2n)$. And, if the Karatsuba method is used in the computation,

$$T(2n) \le 3T(n) + \alpha n$$

for some constant α, since the Karatsuba method uses three half-sized products plus some additions and subtractions. The recursion above implies by induction that

$$T(2^k) \le \alpha(3^k - 2^k), \qquad k \ge 1.$$

Then, $T(n) \le \alpha(3^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil}) < \alpha 3^{1+\log_2 n} = 3\alpha 3^{\log_2 n} = 3\alpha n^{\log_2 3} \approx 3\alpha n^{1.58}$.

5.6.2 Number of Scalar Multiplications

Let #mul(n) denote the number of the scalar products required for the multiplication of two degree $n-1$ polynomials. As can be understood from (5.11) and (5.12), the Karatsuba method computes a product of degree $2n-1$ polynomials from the three products of degree $n-1$ polynomials. Thus,

$$\#mul(2n) = 3\ \#mul(n)$$

for the Karatsuba method. As a result, if n is a power of two,

$$\#mul(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Let n be a power of 2, the number of scalar products

$$\#mul(2n) = 2\ \#mul(n) + \#mul(n-1) - 1.$$

5.6.2.1 Integer Multiplication

To multiply two n-digit integers a and b with the Karatsuba–Ofman method, these integers are first split into the half-sized integers

$$\begin{aligned} a_H &= (a_{n-1}, \ldots, a_{\lceil n/2 \rceil}), & a_L &= (a_{\lceil n/2 \rceil - 1}, \ldots, a_0), \\ b_H &= (b_{n-1}, \ldots, b_{\lceil n/2 \rceil}), & b_L &= (b_{\lceil n/2 \rceil - 1}, \ldots, b_0). \end{aligned} \tag{5.13}$$

The integers above are made up from the higher and the lower digits of a and b. Thus, $a = a_L + a_H \beta^{\lceil n/2 \rceil}$ and $b = b_L + b_H \beta^{\lceil n/2 \rceil}$ where β is the integer base. Next, the three subproducts $f = a_L b_L$, $g = a_H b_H$, and $e = (a_L - a_H)(b_L - b_H)$. Finally, the results are combined to produce

$$d = f + g\beta^{2\lceil n/2\rceil} + (f+g-e)\beta^{\lceil n/2\rceil}. \tag{5.14}$$

Notice $f+g-e = a_Lb_H + a_Hb_L$ gives the sum of the cross products. Thus, the Karatsuba–Ofman method actually computes

$$d = a_Lb_L + a_Hb_H\beta^{2\lceil n/2\rceil} + [a_Lb_H + a_Hb_L]\beta^{\lceil n/2\rceil} = a \times b.$$

Algorithm 20 multiplies two integers using Karatsuba–Ofman method. In Step 1, the standard multiplication is used without any recursion, if the inputs are smaller than a threshold. Otherwise, the remaining steps are executed. First, $f+g-e = a_Lb_L + a_Hb_H - (a_L - a_H)(b_L - b_H)$ needs to be computed from the half-sized operands. To work with only positive operands, this term can also be computed as $f+g-e = a_Lb_L + a_Hb_H - s_as_b|a_L - a_H||b_L - b_H|$ where $s_a = \text{sign}(a_L - a_H)$ and $s_b = \text{sign}(b_L - b_H)$.

Algorithm 20: Karatsuba–Ofman multiplication for integers

```
INPUT: $n$-digit integers $a$ and $b$.
OUTPUT: $2n$-digit integer $d = a \times b$.
 1.  **if** $n \le n_{\text{threshold}}$ **then** $d = a \times b$, **return**($d$)
 2.  Split $a$ into $a_H = (a_{n-1}, \ldots, a_{\lceil n/2\rceil})$ and $a_L = (a_{\lceil n/2\rceil-1}, \ldots, a_0)$.
 3.  Split $b$ into $b_H = (b_{n-1}, \ldots, b_{\lceil n/2\rceil})$ and $b_L = (b_{\lceil n/2\rceil-1}, \ldots, b_0)$.
 4.  $s_a = \text{sign}(a_L - a_H)$ (Use Algorithm 21.)
 5.  $s_b = \text{sign}(b_L - b_H)$ (Use Algorithm 21.)
 6.  **if** $s_a = +1$ **then** $a_M = a_L - a_H$ **else** $a_M = a_H - a_L$
 7.  **if** $s_b = +1$ **then** $b_M = b_L - b_H$ **else** $b_M = b_H - b_L$
 8.  $e = s_a s_b$ recursive-call$(a_M, b_M)$
 9.  $f =$ recursive-call$(a_L, b_L)$
10.  $g =$ recursive-call$(a_H, b_H)$
11.  $h = f + g - e$
12.  $d = f + g\beta^{2\lceil n/2\rceil} + h\beta^{\lceil n/2\rceil}$
13.  **return**($d$)
```

The signs s_a and s_b are obtained by Algorithm 21.

Algorithm 21: Integer comparison

```
INPUT: $k$-digit integer $u$ and $l$-digit integer $v$ where $k \ge l$.
OUTPUT: $s = \text{sign}(u - v)$.
 1.  $s = +1, i = k$
 2.  **while** $i > l$ and $u_i = 0$ **do** $i = i - 1$
 3.  **if** $i = l$ **then**
 4.      **while** $i \ge 0$ and $u_i = v_i$ **do** $i = i - 1$
 5.      **if** $i \ge 0$ and $u_i < v_i$ **then** $s = -1$
 6.  **return**($s$)
```

Algorithm 20 requires some multiprecision additions and subtractions. These operations are performed as shown in (5.1) and (5.2). The subtractions in Steps 6 and 7 have at the most $\lceil n/2 \rceil$-digit operands and produce a positive $\lceil n/2 \rceil$-digit result. The addition and the subtraction in Step 11 have at the most $2\lceil n/2 \rceil$-digit operands. These operations produce $h = a_L b_H + a_H b_L$. Element h is an $(n+1)$-digit positive integer since the sizes of $a_L b_H$ and $a_H b_L$ are $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ digits. Also, the multiplications by the powers of the base β in Step 12 are nothing else than multi-digit left shifts.

5.7 Exercises

1. As shown in Section 5.2, the elements in $\mathbb{F}_p$ can be represented by the integers $\{0,1,2,\ldots,p-1\}$ and the field multiplication in $\mathbb{F}_p$ can be defined as the multiplication modulo p in $\mathbb{Z}$. Show that every non-zero field element has a multiplicative inverse according to this definition. Hint: Use the Bezout's identity for integers $u\hat{u} + v\hat{v} = \gcd(u,v)$ and investigate the case $u = p$ and $0 \le v < p$.
2. As shown in Section 5.2, the elements in $\mathbb{F}_{p^m}$ can be represented by the polynomials over $\mathbb{F}_p$ of degree less than m. Also, the field multiplication in $\mathbb{F}_{p^m}$ can be defined as the polynomial multiplication modulo $\omega(x)$ where $\omega(x)$ is degree m irreducible polynomial over $\mathbb{F}_p$. Show that every non-zero field element has a multiplicative inverse according to this definition. Hint: Use the Bezout's identity for polynomials
$$u(x)\hat{u}(x) + v(x)\hat{v}(x) = \gcd(u(x), v(x))$$
and investigate the case $u(x) = \omega(x)$ and $0 \le \deg(v(x)) < m$.
3. Use the equality $\sum_{k=0}^{n-1} k\beta^{k-1}(\beta-1)^2 = n(\beta-1)\beta^n - (\beta^n - 1)$ and show that the three digit number (U,H,L) in Algorithm 2 does not overflow, if $n(\beta-1) \le \beta^2$ where β is the integer base and n is operand size in the number of digits.
4. Use Algorithm 5 as an example and construct an efficient algorithm to reduce the integers modulo $2^{224} - 2^{96} + 1$.
5. As shown in the chapter, the Barret Algorithm for integers estimates the quotient $\lfloor d/p \rfloor$ with at most two errors, if the parameters k and k' satisfy that $k \ge \log_2 d \ge \log_2 p \ge k'$. Let these parameters be chosen such that $k \ge \log_2 d - u$ and $k' \le \log_2 p + v$ where $k \ge k'$ still holds. Show that the quotient estimation error will be at most $2^u + 2^v$.
6. As shown in the chapter, the Barret Algorithm for polynomials over $\mathbb{F}_2$ estimates the quotient $\lfloor d(x)/\omega(x) \rfloor$ without any error, if the parameters k and k' satisfy that $k \ge \deg(d(x)) \ge \deg(\omega(x)) \ge k'$. Let these parameters be chosen such that $k \ge \deg(d(x)) - u$ and $k' \le \deg(\omega(x)) + v$ where $k \ge k'$ still holds. What will be the error in the quotient estimation?
7. Algorithm 19 fixes the pseudo-Mersenne prime to the form of $p = 2^{w-1} - \alpha$. What changes do you need to make to this algorithm so that it will support pseudo-Mersenne primes in the form $p = 2^{w-1} \pm \alpha$.

5.8 Projects

1. Implement the recursive Karatsuba-Ofman algorithm in C for integer multiplication and polynomial multiplication in $\mathbb{F}_2$.
2. Implement the algorithms given in this chapter in an algebraic computational system (such as, Maple, Mathematica, or Matlab).

References

1. D. V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 2000.
2. P. Barrett. Implementing the Rivest Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology—CRYPTO 86, Proceedings*, Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer, Berlin, Germany, 1986.
3. E. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, NY, 1968.
4. R. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, MA, 1983.
5. A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In *Crypto '93*, Lecture Notes in Computer Science, vol. 773, pp. 175–186, 1994.
6. M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. *Topics in Cryptology – CT-RSA 2001*, Lecture Notes in Computer Science, vol. 2020, pp. 250–265, Springer, Berlin, Germany, 2001
7. J. F. Dhem. Efficient modular reduction algorithm in $\mathbb{F}_q[x]$ and its application to "left to right" modular multiplication in $\mathbb{F}_2[x]$. In C. D. Walter, editor, *Cryptographic Hardware and Embedded Systems – CHES 2003*, Lecture Notes in Computer Science, vol. 2779, pp. 203–213. Springer, Berlin, Germany, 2003.
8. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
9. IEEE P1363. Standard specifications for public-key cryptography.
10. D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, Third edition, 1998.
11. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
12. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.
13. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
14. R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, Boston, MA, Second edition, 1989.

15. A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.
16. A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. Kluwer Academic Publishers, Boston, MA, 1993.
17. V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology—CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pp. 417–426. Springer, Berlin, Germany, 1985.
18. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
19. W. W. Peterson and E. J. Weldon Jr. *Error-Correcting Codes*. MIT Press, Cambridge, MA, 1972.
20. J. Solinas. *Generalized Mersenne numbers*. Technical Report CORR 99-39, Dept. of C&O, University of Waterloo, 1999.
21. S. B. Wicker and V. K. Bhargava, editors. *Reed-Solomon Codes and Their Applications*. IEEE Press, New York, NY, 1994.