CHAPTER 2

Finite Field Arithmetic

The efficient implementation of finite field arithmetic is an important prerequisite in elliptic curve systems because curve operations are performed using arithmetic operations in the underlying field. §2.1 provides an informal introduction to the theory of finite fields. Three kinds of fields that are especially amenable for the efficient implementation of elliptic curve systems are prime fields, binary fields, and optimal extension fields. Efficient algorithms for software implementation of addition, subtraction, multiplication and inversion in these fields are discussed at length in §2.2, §2.3, and §2.4, respectively. Hardware implementation is considered in §5.2 and chapter notes and references are provided in §2.5.

2.1 Introduction to finite fields

Fields are abstractions of familiar number systems (such as the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$) and their essential properties. They consist of a set $\mathbb{F}$ together with two operations, addition (denoted by $+$) and multiplication (denoted by $\cdot$), that satisfy the usual arithmetic properties:

(i) $(\mathbb{F}, +)$ is an abelian group with (additive) identity denoted by 0.

(ii) $(\mathbb{F} \setminus \{0\}, \cdot)$ is an abelian group with (multiplicative) identity denoted by 1.

(iii) The distributive law holds: $(a+b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in \mathbb{F}$.

If the set $\mathbb{F}$ is finite, then the field is said to be *finite*.

This section presents basic facts about finite fields. Other properties will be presented throughout the book as needed.

Field operations

A field $\mathbb{F}$ is equipped with two operations, addition and multiplication. *Subtraction* of field elements is defined in terms of addition: for $a, b \in \mathbb{F}$, $a - b = a + (-b)$ where $-b$ is the unique element in $\mathbb{F}$ such that $b + (-b) = 0$ ($-b$ is called the *negative* of b). Similarly, *division* of field elements is defined in terms of multiplication: for $a, b \in \mathbb{F}$ with $b \neq 0$, $a/b = a \cdot b^{-1}$ where b^{-1} is the unique element in $\mathbb{F}$ such that $b \cdot b^{-1} = 1$. (b^{-1} is called the *inverse* of b.)

Existence and uniqueness

The *order* of a finite field is the number of elements in the field. There exists a finite field $\mathbb{F}$ of order q if and only if q is a prime power, i.e., $q = p^m$ where p is a prime number called the *characteristic* of $\mathbb{F}$, and m is a positive integer. If $m = 1$, then $\mathbb{F}$ is called a *prime field*. If $m \geq 2$, then $\mathbb{F}$ is called an *extension field*. For any prime power q, there is essentially only one finite field of order q; informally, this means that any two finite fields of order q are structurally the same except that the labeling used to represent the field elements may be different (cf. Example 2.3). We say that any two finite fields of order q are *isomorphic* and denote such a field by $\mathbb{F}_q$.

Prime fields

Let p be a prime number. The integers modulo p, consisting of the integers $\{0, 1, 2, \ldots, p-1\}$ with addition and multiplication performed modulo p, is a finite field of order p. We shall denote this field by $\mathbb{F}_p$ and call p the *modulus* of $\mathbb{F}_p$. For any integer a, $a \bmod p$ shall denote the unique integer remainder r, $0 \leq r \leq p-1$, obtained upon dividing a by p; this operation is called *reduction modulo* p.

Example 2.1 (*prime field* $\mathbb{F}_{29}$) The elements of $\mathbb{F}_{29}$ are $\{0, 1, 2, \ldots, 28\}$. The following are some examples of arithmetic operations in $\mathbb{F}_{29}$.

(i) Addition: $17 + 20 = 8$ since $37 \bmod 29 = 8$.

(ii) Subtraction: $17 - 20 = 26$ since $-3 \bmod 29 = 26$.

(iii) Multiplication: $17 \cdot 20 = 21$ since $340 \bmod 29 = 21$.

(iv) Inversion: $17^{-1} = 12$ since $17 \cdot 12 \bmod 29 = 1$.

Binary fields

Finite fields of order 2^m are called *binary fields* or *characteristic-two finite fields*. One way to construct $\mathbb{F}_{2^m}$ is to use a *polynomial basis representation*. Here, the elements of $\mathbb{F}_{2^m}$ are the binary polynomials (polynomials whose coefficients are in the field $\mathbb{F}_2 = \{0, 1\}$) of degree at most $m-1$:

$$\mathbb{F}_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2 z^2 + a_1 z + a_0 \,:\, a_i \in \{0, 1\}\}.$$

An irreducible binary polynomial $f(z)$ of degree m is chosen (such a polynomial exists for any m and can be efficiently found; see §A.1). Irreducibility of $f(z)$ means that $f(z)$ cannot be factored as a product of binary polynomials each of degree less than m. Addition of field elements is the usual addition of polynomials, with coefficient arithmetic performed modulo 2. Multiplication of field elements is performed modulo the *reduction polynomial* $f(z)$. For any binary polynomial $a(z)$, $a(z) \bmod f(z)$ shall denote the unique remainder polynomial $r(z)$ of degree less than m obtained upon long division of $a(z)$ by $f(z)$; this operation is called *reduction modulo* $f(z)$.

Example 2.2 (*binary field* $\mathbb{F}_{2^4}$) The elements of $\mathbb{F}_{2^4}$ are the 16 binary polynomials of degree at most 3:

$$\begin{array}{llll} 0 & z^2 & z^3 & z^3+z^2 \\ 1 & z^2+1 & z^3+1 & z^3+z^2+1 \\ z & z^2+z & z^3+z & z^3+z^2+z \\ z+1 & z^2+z+1 & z^3+z+1 & z^3+z^2+z+1. \end{array}$$

The following are some examples of arithmetic operations in $\mathbb{F}_{2^4}$ with reduction polynomial $f(z) = z^4+z+1$.

(i) Addition: $(z^3+z^2+1)+(z^2+z+1) = z^3+z$.

(ii) Subtraction: $(z^3+z^2+1)-(z^2+z+1) = z^3+z$. (Note that since $-1=1$ in $\mathbb{F}_2$, we have $-a=a$ for all $a \in \mathbb{F}_{2^m}$.)

(iii) Multiplication: $(z^3+z^2+1)\cdot(z^2+z+1) = z^2+1$ since

$$(z^3+z^2+1)\cdot(z^2+z+1) = z^5+z+1$$

and

$$(z^5+z+1) \bmod (z^4+z+1) = z^2+1.$$

(iv) Inversion: $(z^3+z^2+1)^{-1} = z^2$ since $(z^3+z^2+1)\cdot z^2 \bmod (z^4+z+1) = 1$.

Example 2.3 (*isomorphic fields*) There are three irreducible binary polynomials of degree 4, namely $f_1(z) = z^4+z+1$, $f_2(z) = z^4+z^3+1$ and $f_3(z) = z^4+z^3+z^2+z+1$. Each of these reduction polynomials can be used to construct the field $\mathbb{F}_{2^4}$; let's call the resulting fields K_1, K_2 and K_3. The field elements of K_1, K_2 and K_3 are the same 16 binary polynomials of degree at most 3. Superficially, these fields appear to be different, e.g., $z^3 \cdot z = z+1$ in K_1, $z^3 \cdot z = z^3+1$ in K_2, and $z^3 \cdot z = z^3+z^2+z+1$ in K_3. However, all fields of a given order are isomorphic—that is, the differences are only in the labeling of the elements. An isomorphism between K_1 and K_2 may be constructed by finding $c \in K_2$ such that $f_1(c) \equiv 0 \pmod{f_2}$ and then extending $z \mapsto c$ to an isomorphism $\varphi : K_1 \to K_2$; the choices for c are z^2+z, z^2+z+1, z^3+z^2, and z^3+z^2+1.

Extension fields

The polynomial basis representation for binary fields can be generalized to all extension fields as follows. Let p be a prime and $m \geq 2$. Let $\mathbb{F}_p[z]$ denote the set of all polynomials in the variable z with coefficients from $\mathbb{F}_p$. Let $f(z)$, the *reduction polynomial*, be an irreducible polynomial of degree m in $\mathbb{F}_p[z]$—such a polynomial exists for any p and m and can be efficiently found (see §A.1). Irreducibility of $f(z)$ means that $f(z)$ cannot be factored as a product of polynomials in $\mathbb{F}_p[z]$ each of degree less than m. The elements of $\mathbb{F}_{p^m}$ are the polynomials in $\mathbb{F}_p[z]$ of degree at most $m-1$:

$$\mathbb{F}_{p^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2z^2 + a_1z + a_0 \ : \ a_i \in \mathbb{F}_p\}.$$

Addition of field elements is the usual addition of polynomials, with coefficient arithmetic performed in $\mathbb{F}_p$. Multiplication of field elements is performed modulo the polynomial $f(z)$.

Example 2.4 (*an extension field*) Let $p = 251$ and $m = 5$. The polynomial $f(z) = z^5 + z^4 + 12z^3 + 9z^2 + 7$ is irreducible in $\mathbb{F}_{251}[z]$ and thus can serve as reduction polynomial for the construction of $\mathbb{F}_{251^5}$, the finite field of order 251^5. The elements of $\mathbb{F}_{251^5}$ are the polynomials in $\mathbb{F}_{251}[z]$ of degree at most 4.

The following are some examples of arithmetic operations in $\mathbb{F}_{251^5}$. Let $a = 123z^4 + 76z^2 + 7z + 4$ and $b = 196z^4 + 12z^3 + 225z^2 + 76$.

(i) Addition: $a + b = 68z^4 + 12z^3 + 50z^2 + 7z + 80$.

(ii) Subtraction: $a - b = 178z^4 + 239z^3 + 102z^2 + 7z + 179$.

(iii) Multiplication: $a \cdot b = 117z^4 + 151z^3 + 117z^2 + 182z + 217$.

(iv) Inversion: $a^{-1} = 109z^4 + 111z^3 + 250z^2 + 98z + 85$.

Subfields of a finite field

A subset k of a field K is a *subfield* of K if k is itself a field with respect to the operations of K. In this instance, K is said to be an *extension field* of k. The subfields of a finite field can be easily characterized. A finite field $\mathbb{F}_{p^m}$ has precisely one subfield of order p^l for each positive divisor l of m; the elements of this subfield are the elements $a \in \mathbb{F}_{p^m}$ satisfying $a^{p^l} = a$. Conversely, every subfield of $\mathbb{F}_{p^m}$ has order p^l for some positive divisor l of m.

Bases of a finite field

The finite field $\mathbb{F}_{q^n}$ can be viewed as a vector space over its subfield $\mathbb{F}_q$. Here, vectors are elements of $\mathbb{F}_{q^n}$, scalars are elements of $\mathbb{F}_q$, vector addition is the addition operation in $\mathbb{F}_{q^n}$, and scalar multiplication is the multiplication in $\mathbb{F}_{q^n}$ of $\mathbb{F}_q$-elements with $\mathbb{F}_{q^n}$-elements. The vector space has dimension n and has many bases.

If $B = \{b_1, b_2, \ldots, b_n\}$ is a basis, then $a \in \mathbb{F}_{q^n}$ can be uniquely represented by an n-tuple $(a_1, a_2, \ldots, a_n)$ of $\mathbb{F}_q$-elements where $a = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$. For example, in the polynomial basis representation of the field $\mathbb{F}_{p^m}$ described above, $\mathbb{F}_{p^m}$ is an m-dimensional vector space over $\mathbb{F}_p$ and $\{z^{m-1}, z^{m-2}, \ldots, z^2, z, 1\}$ is a basis for $\mathbb{F}_{p^m}$ over $\mathbb{F}_p$.

Multiplicative group of a finite field

The nonzero elements of a finite field $\mathbb{F}_q$, denoted $\mathbb{F}_q^*$, form a cyclic group under multiplication. Hence there exist elements $b \in \mathbb{F}_q^*$ called *generators* such that

$$\mathbb{F}_q^* = \{b^i : 0 \leq i \leq q-2\}.$$

The *order* of $a \in \mathbb{F}_q^*$ is the smallest positive integer t such that $a^t = 1$. Since $\mathbb{F}_q^*$ is a cyclic group, it follows that t is a divisor of $q-1$.

2.2 Prime field arithmetic

This section presents algorithms for performing arithmetic in the prime field $\mathbb{F}_p$. Algorithms for arbitrary primes p are presented in §2.2.1–§2.2.5. The reduction step can be accelerated considerably when the modulus p has a special form. Efficient reduction algorithms for the NIST primes such as $p = 2^{192} - 2^{64} - 1$ are considered in §2.2.6.

The algorithms presented here are well suited for software implementation. We assume that the implementation platform has a W-bit architecture where W is a multiple of 8. Workstations are commonly 64- or 32-bit architectures. Low-power or inexpensive components may have smaller W, for example, some embedded systems are 16-bit and smartcards may have $W = 8$. The bits of a W-bit word U are numbered from 0 to $W-1$, with the rightmost bit of U designated as bit 0.

The elements of $\mathbb{F}_p$ are the integers from 0 to $p-1$. Let $m = \lceil \log_2 p \rceil$ be the bitlength of p, and $t = \lceil m/W \rceil$ be its wordlength. Figure 2.1 illustrates the case where the binary representation of a field element a is stored in an array $A = (A[t-1], \ldots, A[2], A[1], A[0])$ of t W-bit words, where the rightmost bit of $A[0]$ is the least significant bit.

$A[t-1]$	$\cdots$	$A[2]$	$A[1]$	$A[0]$

Figure 2.1. Representation of $a \in \mathbb{F}_p$ as an array A of W-bit words. As an integer, $a = 2^{(t-1)W} A[t-1] + \cdots + 2^{2W} A[2] + 2^W A[1] + A[0]$.

Hardware characteristics may favour approaches different from those of the algorithms and field element representation presented here. §5.1.1 examines possible bottlenecks in multiplication due to constraints on hardware integer multipliers and

the cost of propagating carries. §5.1.2 briefly discusses the use of floating-point hardware commonly found on workstations, which can give substantial improvement in multiplication times (and uses a different field element representation). Similarly, single-instruction multiple-data (SIMD) registers on some processors can be employed; see §5.1.3. Selected timings for field operations appear in §5.1.5.

2.2.1 Addition and subtraction

Algorithms for field addition and subtraction are given in terms of corresponding algorithms for multi-word integers. The following notation and terminology is used. An assignment of the form "$(\varepsilon, z) \leftarrow w$" for an integer w is understood to mean

$$z \leftarrow w \bmod 2^W, \text{ and}$$
$$\varepsilon \leftarrow 0 \text{ if } w \in [0, 2^W), \text{ otherwise } \varepsilon \leftarrow 1.$$

If $w = x + y + \varepsilon'$ for $x, y \in [0, 2^W)$ and $\varepsilon' \in \{0, 1\}$, then $w = \varepsilon 2^W + z$ and ε is called the *carry bit* from single-word addition (with $\varepsilon = 1$ if and only if $z < x + \varepsilon'$). Algorithm 2.5 performs addition of multi-word integers.

Algorithm 2.5 Multiprecision addition

INPUT: Integers $a, b \in [0, 2^{Wt})$.
OUTPUT: (ε, c) where $c = a + b \bmod 2^{Wt}$ and ε is the carry bit.
1. $(\varepsilon, C[0]) \leftarrow A[0] + B[0]$.
2. For i from 1 to $t - 1$ do
   2.1 $(\varepsilon, C[i]) \leftarrow A[i] + B[i] + \varepsilon$.
3. Return(ε, c).

On processors that handle the carry as part of the instruction set, there need not be any explicit check for carry. Multi-word subtraction (Algorithm 2.6) is similar to addition, with the carry bit often called a "borrow" in this context.

Algorithm 2.6 Multiprecision subtraction

INPUT: Integers $a, b \in [0, 2^{Wt})$.
OUTPUT: (ε, c) where $c = a - b \bmod 2^{Wt}$ and ε is the borrow.
1. $(\varepsilon, C[0]) \leftarrow A[0] - B[0]$.
2. For i from 1 to $t - 1$ do
   2.1 $(\varepsilon, C[i]) \leftarrow A[i] - B[i] - \varepsilon$.
3. Return(ε, c).

Modular addition ($(x + y) \bmod p$) and subtraction ($(x - y) \bmod p$) are adapted directly from the corresponding algorithms above, with an additional step for reduction modulo p.

Algorithm 2.7 Addition in $\mathbb{F}_p$

INPUT: Modulus p, and integers $a, b \in [0, p-1]$.
OUTPUT: $c = (a+b) \bmod p$.

1. Use Algorithm 2.5 to obtain (ε, c) where $c = a + b \bmod 2^{Wt}$ and ε is the carry bit.
2. If $\varepsilon = 1$, then subtract p from $c = (C[t-1], \ldots, C[2], C[1], C[0])$;
   Else if $c \geq p$ then $c \leftarrow c - p$.
3. Return(c).

Algorithm 2.8 Subtraction in $\mathbb{F}_p$

INPUT: Modulus p, and integers $a, b \in [0, p-1]$.
OUTPUT: $c = (a-b) \bmod p$.

1. Use Algorithm 2.6 to obtain (ε, c) where $c = a - b \bmod 2^{Wt}$ and ε is the borrow.
2. If $\varepsilon = 1$, then add p to $c = (C[t-1], \ldots, C[2], C[1], C[0])$.
3. Return(c).

2.2.2 Integer multiplication

Field multiplication of $a, b \in \mathbb{F}_p$ can be accomplished by first multiplying a and b as integers, and then reducing the result modulo p. Algorithms 2.9 and 2.10 are elementary integer multiplication routines which illustrate basic operand scanning and product scanning methods, respectively. In both algorithms, (UV) denotes a $(2W)$-bit quantity obtained by concatenation of W-bit words U and V.

Algorithm 2.9 Integer multiplication (operand scanning form)

INPUT: Integers $a, b \in [0, p-1]$.
OUTPUT: $c = a \cdot b$.

1. Set $C[i] \leftarrow 0$ for $0 \leq i \leq t-1$.
2. For i from 0 to $t-1$ do
   2.1 $U \leftarrow 0$.
   2.2 For j from 0 to $t-1$ do:
   $(UV) \leftarrow C[i+j] + A[i] \cdot B[j] + U$.
   $C[i+j] \leftarrow V$.
   2.3 $C[i+t] \leftarrow U$.
3. Return(c).

The calculation $C[i+j]+A[i]\cdot B[j]+U$ at step 2.2 is called the *inner product operation*. Since the operands are W-bit values, the inner product is bounded by $2(2^W-1)+(2^W-1)^2=2^{2W}-1$ and can be represented by (UV).

Algorithm 2.10 is arranged so that the product $c=ab$ is calculated right-to-left. As in the preceding algorithm, a $(2W)$-bit product of W-bit operands is required. The values R_0, R_1, R_2, U, and V are W-bit words.

Algorithm 2.10 Integer multiplication (product scanning form)

INPUT: Integers $a, b \in [0, p-1]$.
OUTPUT: $c = a \cdot b$.

1. $R_0 \leftarrow 0$, $R_1 \leftarrow 0$, $R_2 \leftarrow 0$.
2. For k from 0 to $2t-2$ do
   2.1 For each element of $\{(i,j) \mid i+j=k,\ 0 \le i, j \le t-1\}$ do
   $(UV) \leftarrow A[i]\cdot B[j]$.
   $(\varepsilon, R_0) \leftarrow R_0 + V$.
   $(\varepsilon, R_1) \leftarrow R_1 + U + \varepsilon$.
   $R_2 \leftarrow R_2 + \varepsilon$.
   2.2 $C[k] \leftarrow R_0$, $R_0 \leftarrow R_1$, $R_1 \leftarrow R_2$, $R_2 \leftarrow 0$.
3. $C[2t-1] \leftarrow R_0$.
4. Return(c).

Note 2.11 (*implementing Algorithms 2.9 and 2.10*) Algorithms 2.9 and 2.10 are written in a form motivated by the case where a W-bit architecture has a multiplication operation giving a $2W$-bit result (e.g., the Intel Pentium or Sun SPARC). A common exception is illustrated by the 64-bit Sun UltraSPARC, where the multiplier produces the lower 64 bits of the product of 64-bit inputs. One variation of these algorithms splits a and b into $(W/2)$-bit half-words, but accumulates in W-bit registers. See also §5.1.3 for an example concerning a 32-bit architecture which has some 64-bit operations.

Karatsuba-Ofman multiplication

Algorithms 2.9 and 2.10 take $O(n^2)$ bit operations for multiplying two n-bit integers. A divide-and-conquer algorithm due to Karatsuba and Ofman reduces the complexity to $O(n^{\log_2 3})$. Suppose that $n=2l$ and $x=x_1 2^l + x_0$ and $y=y_1 2^l + y_0$ are $2l$-bit integers. Then

$$\begin{aligned} xy &= (x_1 2^l + x_0)(y_1 2^l + y_0) \\ &= x_1 \cdot y_1 2^{2l} + [(x_0+x_1)\cdot(y_0+y_1) - x_1 y_1 - x_0 \cdot y_0]2^l + x_0 y_0 \end{aligned}$$

and xy can be computed by performing three multiplications of l-bit integers (as opposed to one multiplication with $2l$-bit integers) along with two additions and two

subtractions.[1] For large values of l, the cost of the additions and subtractions is insignificant relative to the cost of the multiplications. The procedure may be applied recursively to the intermediate values, terminating at some threshold (possibly the word size of the machine) where a classical or other method is employed.

For integers of modest size, the overhead in Karatsuba-Ofman may be significant. Implementations may deviate from the traditional description in order to reduce the shifting required (for multiplications by 2^l and 2^{2l}) and make more efficient use of word-oriented operations. For example, it may be more effective to split on word boundaries, and the split at a given stage may be into more than two fragments.

Example 2.12 (*Karatsuba-Ofman methods*) Consider multiplication of 224-bit values x and y, on a machine with word size $W = 32$. Two possible depth-2 approaches are indicated in Figure 2.2. The split in Figure 2.2(a) is perhaps mathematically more elegant

(a) $n/2$ split (b) split on word boundary

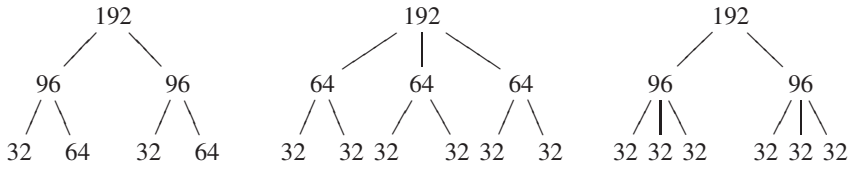
Figure 2.2. Depth-2 splits for 224-bit integers. The product xy using (a) has three 112×112 multiplications, each performed using three 56×56 multiplications. Using (b), xy has a 96×96 (split as a 32×32 and two 64×64) and two 128×128 multiplications (each generating three 64×64 multiplies).

and may have more reusable code compared with that in Figure 2.2(b). However, more shifting will be required (since the splits are not on word boundaries). If multiplication of 56-bit quantities (perhaps by another application of Karatsuba-Ofman) has approximately the same cost as multiplication of 64-bit values, then the split has under-utilized the hardware capabilities since the cost is nine 64-bit multiplications versus one 32-bit and eight 64-bit multiplications in (b). On the other hand, the split on word boundaries in Figure 2.2(b) has more complicated cross term calculations, since there may be carry to an additional word. For example, the cross terms at depth 2 are of the form

$$(x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0$$

where $x_0 + x_1$ and $y_0 + y_1$ are 57-bit in (a) and 65-bit in (b). Split (b) costs somewhat more here, although $(x_0 + x_1)(y_0 + y_1)$ can be managed as a 64×64 mulitply followed by two possible additions corresponding to the high bits.

[1] The cross term can be written $(x_0 - x_1)(y_1 - y_0) + x_0 y_0 + x_1 y_1$ which may be useful on some platforms or if it is known a priori that $x_0 \geq x_1$ and $y_0 \leq y_1$.



(a) binary split (b) 3-way split at depth 1 (c) 3-way split at depth 2

Figure 2.3. Depth-2 splits for 192-bit integers. The product xy using (a) has three 96×96 multiplications. Each is performed with a 32×32 and two 64×64 (each requiring three 32×32) multiplications, for a total of 21 multiplications of size 32×32. Using (b) or (c), only 18 multiplications of size 32×32 are required.

As a second illustration, consider Karatsuba-Ofman applied to 192-bit integers, again with $W = 32$. Three possible depth-2 approaches are given in Figure 2.3. In terms of 32×32 multiplications, the split in Figure 2.3(a) will require 21, while (b) and (c) use 18. The basic idea is that multiplication of $3l$-bit integers $x = x_2 2^{2l} + x_1 2^l + x_0$ and $y = y_2 2^{2l} + y_1 2^l + y_0$ can be done as

$$\begin{aligned} xy &= (x_2 2^{2l} + x_1 2^l + x_0) \cdot (y_2 2^{2l} + y_1 2^l + y_0) \\ &= x_2 y_2 2^{4l} + (x_2 y_1 + x_1 y_2) 2^{3l} + (x_2 y_0 + x_0 y_2 + x_1 y_1) 2^{2l} \\ &\quad + (x_1 y_0 + x_0 y_1) 2^l + x_0 y_0 \\ &= x_2 \cdot y_2 2^{4l} + [(x_2 + x_1) \cdot (y_2 + y_1) - x_2 y_2 - x_1 \cdot y_1] 2^{3l} \\ &\quad + [(x_2 + x_0) \cdot (y_2 + y_0) - x_2 y_2 - x_0 \cdot y_0 + x_1 y_1] 2^{2l} \\ &\quad + [(x_1 + x_0) \cdot (y_1 + y_0) - x_1 y_1 - x_0 y_0] 2^l + x_0 y_0 \end{aligned}$$

for a total of six multiplications of l-bit integers.

The performance of field multiplication is fundamental to mechanisms based on elliptic curves. Constraints on hardware integer multipliers and the cost of carry propagation can result in significant bottlenecks in direct implementations of Algorithms 2.9 and 2.10. As outlined in the introductory paragraphs of §2.2, Chapter 5 discusses alternative strategies applicable in some environments.

2.2.3 Integer squaring

Field squaring of $a \in \mathbb{F}_p$ can be accomplished by first squaring a as an integer, and then reducing the result modulo p. A straightforward modification of Algorithm 2.10 gives the following algorithm for integer squaring, reducing the number of required single-

precision multiplications by roughly half. In step 2.1, a $(2W+1)$-bit result (ε, UV) is obtained from multiplication of the $(2W)$-bit quantity (UV) by 2.

Algorithm 2.13 Integer squaring

INPUT: Integer $a \in [0, p-1]$.
OUTPUT: $c = a^2$.

1. $R_0 \leftarrow 0$, $R_1 \leftarrow 0$, $R_2 \leftarrow 0$.
2. For k from 0 to $2t-2$ do
   2.1 For each element of $\{(i, j) \mid i + j = k,\ 0 \le i \le j \le t-1\}$ do
   $(UV) \leftarrow A[i] \cdot A[j]$.
   If $(i < j)$ then do: $(\varepsilon, UV) \leftarrow (UV) \cdot 2$, $R_2 \leftarrow R_2 + \varepsilon$.
   $(\varepsilon, R_0) \leftarrow R_0 + V$.
   $(\varepsilon, R_1) \leftarrow R_1 + U + \varepsilon$.
   $R_2 \leftarrow R_2 + \varepsilon$.
   2.2 $C[k] \leftarrow R_0$, $R_0 \leftarrow R_1$, $R_1 \leftarrow R_2$, $R_2 \leftarrow 0$.
3. $C[2t-1] \leftarrow R_0$.
4. Return(c).

The multiplication by 2 in step 2.1 may be implemented as two single-precision shift-through-carry (if available) or as two single-precision additions with carry. The step can be rewritten so that each output word $C[k]$ requires at most one multiplication by 2, at the cost of two additional accumulators and an associated accumulation step.

2.2.4 Reduction

For moduli p that are not of special form, the reduction $z \bmod p$ can be an expensive part of modular multiplication. Since the performance of elliptic curve schemes depends heavily on the speed of field multiplication, there is considerable incentive to select moduli, such as the NIST-recommended primes of §2.2.6, that permit fast reduction. In this section, we present only the reduction method of Barrett and an overview of Montgomery multiplication.

The methods of Barrett and Montgomery are similar in that expensive divisions in classical reduction methods are replaced by less-expensive operations. Barrett reduction can be regarded as a direct replacement for classical methods; however, an expensive modulus-dependent calculation is required, and hence the method is applicable when many reductions are performed with a single modulus. Montgomery's method, on the other hand, requires transformations of the data. The technique can be effective when the cost of the input and output conversions is offset by savings in many intermediate multiplications, as occurs in modular exponentiation.

Note that some modular operations are typically required in a larger framework such as the signature schemes of §4.4, and the moduli involved need not be of special form. In these instances, Barrett reduction may be an appropriate method.

Barrett reduction

Barrett reduction (Algorithm 2.14) finds $z \bmod p$ for given positive integers z and p. In contrast to the algorithms presented in §2.2.6, Barrett reduction does not exploit any special form of the modulus p. The quotient $\lfloor z/p \rfloor$ is estimated using less-expensive operations involving powers of a suitably-chosen base b (e.g., $b = 2^L$ for some L which may depend on the modulus but not on z). A modulus-dependent quantity $\lfloor b^{2k}/p \rfloor$ must be calculated, making the algorithm suitable for the case that many reductions are performed with a single modulus.

Algorithm 2.14 Barrett reduction

INPUT: $p, b \geq 3, k = \lfloor \log_b p \rfloor + 1, 0 \leq z < b^{2k}$, and $\mu = \lfloor b^{2k}/p \rfloor$.
OUTPUT: $z \bmod p$.

1. $\widehat{q} \leftarrow \left\lfloor \lfloor z/b^{k-1} \rfloor \cdot \mu \big/ b^{k+1} \right\rfloor$.
2. $r \leftarrow (z \bmod b^{k+1}) - (\widehat{q} \cdot p \bmod b^{k+1})$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r \geq p$ do: $r \leftarrow r - p$.
5. Return(r).

Note 2.15 (*correctness of Algorithm 2.14*) Let $q = \lfloor z/p \rfloor$; then $r = z \bmod p = z - qp$. Step 1 of the algorithm calculates an estimate $\widehat{q}$ to q since

$$\frac{z}{p} = \frac{z}{b^{k-1}} \cdot \frac{b^{2k}}{p} \cdot \frac{1}{b^{k+1}}.$$

Note that

$$0 \leq \widehat{q} = \left\lfloor \frac{\left\lfloor \frac{z}{b^{k-1}} \right\rfloor \cdot \mu}{b^{k+1}} \right\rfloor \leq \left\lfloor \frac{z}{p} \right\rfloor = q.$$

The following argument shows that $q - 2 \leq \widehat{q} \leq q$; that is, $\widehat{q}$ is a good estimate for q. Define

$$\alpha = \frac{z}{b^{k-1}} - \left\lfloor \frac{z}{b^{k-1}} \right\rfloor, \quad \beta = \frac{b^{2k}}{p} - \left\lfloor \frac{b^{2k}}{p} \right\rfloor.$$

Then $0 \leq \alpha, \beta < 1$ and

$$\begin{aligned} q &= \left\lfloor \frac{\left(\left\lfloor \frac{z}{b^{k-1}} \right\rfloor + \alpha\right)\left(\left\lfloor \frac{b^{2k}}{p} \right\rfloor + \beta\right)}{b^{k+1}} \right\rfloor \\ &\leq \left\lfloor \frac{\left\lfloor \frac{z}{b^{k-1}} \right\rfloor \cdot \mu}{b^{k+1}} + \frac{\left\lfloor \frac{z}{b^{k-1}} \right\rfloor + \left\lfloor \frac{b^{2k}}{p} \right\rfloor + 1}{b^{k+1}} \right\rfloor. \end{aligned}$$

Since $z < b^{2k}$ and $p \geq b^{k-1}$, it follows that

$$\left\lfloor \frac{z}{b^{k-1}} \right\rfloor + \left\lfloor \frac{b^{2k}}{p} \right\rfloor + 1 \leq (b^{k+1} - 1) + b^{k+1} + 1 = 2b^{k+1}$$

and

$$q \leq \left\lfloor \frac{\left\lfloor \frac{z}{b^{k-1}} \right\rfloor \cdot \mu}{b^{k+1}} + 2 \right\rfloor = \widehat{q} + 2.$$

The value r calculated in step 2 necessarily satisfies $r \equiv z - \widehat{q}\,p \pmod{b^{k+1}}$ with $|r| < b^{k+1}$. Hence $0 \leq r < b^{k+1}$ and $r = z - \widehat{q}\,p \bmod b^{k+1}$ after step 3. Now, since $0 \leq z - qp < p$, we have

$$0 \leq z - \widehat{q}\,p \leq z - (q-2)p < 3p.$$

Since $b \geq 3$ and $p < b^k$, we have $3p < b^{k+1}$. Thus $0 \leq z - \widehat{q}\,p < b^{k+1}$, and so $r = z - \widehat{q}\,p$ after step 3. Hence, at most two subtractions at step 4 are required to obtain $0 \leq r < p$, and then $r = z \bmod p$.

Note 2.16 (*computational considerations for Algorithm 2.14*)

(i) A natural choice for the base is $b = 2^L$ where L is near the word size of the processor.

(ii) Other than the calculation of μ (which is done once per modulus), the divisions required are simple shifts of the base-b representation.

(iii) Let $z' = \lfloor z/b^{k-1} \rfloor$. Note that z' and μ have at most $k+1$ base-b digits. The calculation of $\widehat{q}$ in step 1 discards the $k+1$ least-significant digits of the product $z'\mu$. Given the base-b representations $z' = \sum z'_i b^i$ and $\mu = \sum \mu_j b^j$, write

$$z'\mu = \sum_{l=0}^{2k} \Bigg(\underbrace{\sum_{i+j=l} z'_i \mu_j}_{w_l} \Bigg) b^l$$

where w_l may exceed $b-1$. If $b \geq k-1$, then $\sum_{l=0}^{k-2} w_l b^l < b^{k+1}$ and hence

$$0 \leq \frac{z'\mu}{b^{k+1}} - \sum_{l=k-1}^{2k} \frac{w_l b^l}{b^{k+1}} = \sum_{l=0}^{k-2} \frac{w_l b^l}{b^{k+1}} < 1.$$

It follows that $\left\lfloor \sum_{l=k-1}^{2k} w_l b^l \big/ b^{k+1} \right\rfloor$ underestimates $\widehat{q}$ by at most 1 if $b \geq k-1$. At most $\binom{k+2}{2} + k = (k^2 + 5k + 2)/2$ single-precision multiplications (i.e., multiplications of values less than b) are required to find this estimate for $\widehat{q}$.

(iv) Only the $k+1$ least significant digits of $\widehat{q} \cdot p$ are required at step 2. Since $p < b^k$, the $k+1$ digits can be obtained with $\binom{k+1}{2} + k$ single-precision multiplications.

Montgomery multiplication

As with Barrett reduction, the strategy in Montgomery's method is to replace division in classical reduction algorithms with less-expensive operations. The method is not efficient for a single modular multiplication, but can be used effectively in computations such as modular exponentiation where many multiplications are performed for given input. For this section, we give only an overview (for more details, see §2.5).

Let $R > p$ with $\gcd(R, p) = 1$. Montgomery reduction produces $zR^{-1} \bmod p$ for an input $z < pR$. We consider the case that p is odd, so that $R = 2^{Wt}$ may be selected and division by R is relatively inexpensive. If $p' = -p^{-1} \bmod R$, then $c = zR^{-1} \bmod p$ may be obtained via

$$c \leftarrow (z + (zp' \bmod R)p)/R,$$
$$\text{if } c \geq p \text{ then } c \leftarrow c - p,$$

with $t(t+1)$ single-precision multiplications (and no divisions).

Given $x \in [0, p)$, let $\widetilde{x} = xR \bmod p$. Note that $(\widetilde{x}\widetilde{y})R^{-1} \bmod p = (xy)R \bmod p$; that is, Montgomery reduction can be used in a multiplication method on representatives $\widetilde{x}$. We define the Montgomery product of $\widetilde{x}$ and $\widetilde{y}$ to be

$$\text{Mont}(\widetilde{x}, \widetilde{y}) = \widetilde{x}\widetilde{y}R^{-1} \bmod p = xyR \bmod p. \tag{2.1}$$

A single modular multiplication cannot afford the expensive transformations $x \mapsto \widetilde{x} = xR \bmod p$ and $\widetilde{x} \mapsto \widetilde{x}R^{-1} \bmod p = x$; however, the transformations are performed only once when used as part of a larger calculation such as modular exponentiation, as illustrated in Algorithm 2.17.

Algorithm 2.17 Montgomery exponentiation (basic)

INPUT: Odd modulus p, $R = 2^{Wt}$, $p' = -p^{-1} \bmod R$, $x \in [0, p)$, $e = (e_l, \ldots, e_0)_2$.
OUTPUT: $x^e \bmod p$.

1. $\widetilde{x} \leftarrow xR \bmod p$, $A \leftarrow R \bmod p$.
2. For i from l downto 0 do
   2.1 $A \leftarrow \text{Mont}(A, A)$.
   2.2 If $e_i = 1$ then $A \leftarrow \text{Mont}(A, \widetilde{x})$.
3. Return($\text{Mont}(A, 1)$).

As a rough comparison, Montgomery reduction requires $t(t+1)$ single-precision multiplications, while Barrett (with $b = 2^W$) uses $t(t+4)+1$, and hence Montgomery methods are expected to be superior in calculations such as general modular exponentiation. Both methods are expected to be much slower than the direct reduction techniques of §2.2.6 for moduli of special form.

Montgomery arithmetic can be used to accelerate modular inversion methods that use repeated multiplication, where a^{-1} is obtained as $a^{p-2} \bmod p$ (since $a^{p-1} \equiv 1 \pmod p$ if $\gcd(a, p) = 1$). Elliptic curve point multiplication (§3.3) can benefit from Montgomery arithmetic, where the Montgomery inverse discussed in §2.2.5 may also be of interest.

2.2.5 Inversion

Recall that the inverse of a nonzero element $a \in \mathbb{F}_p$, denoted $a^{-1} \bmod p$ or simply a^{-1} if the field is understood from context, is the unique element $x \in \mathbb{F}_p$ such that $ax = 1$ in $\mathbb{F}_p$, i.e., $ax \equiv 1 \pmod p$. Inverses can be efficiently computed by the extended Euclidean algorithm for integers.

The extended Euclidean algorithm for integers

Let a and b be integers, not both 0. The *greatest common divisor (gcd)* of a and b, denoted $\gcd(a, b)$, is the largest integer d that divides both a and b. Efficient algorithms for computing $\gcd(a, b)$ exploit the following simple result.

Theorem 2.18 Let a and b be positive integers. Then $\gcd(a, b) = \gcd(b - ca, a)$ for all integers c.

In the classical Euclidean algorithm for computing the gcd of positive integers a and b where $b \geq a$, b is divided by a to obtain a quotient q and a remainder r satisfying $b = qa + r$ and $0 \leq r < a$. By Theorem 2.18, $\gcd(a, b) = \gcd(r, a)$. Thus, the problem of determining $\gcd(a, b)$ is reduced to that of computing $\gcd(r, a)$ where the arguments (r, a) are smaller than the original arguments (a, b). This process is repeated until one of the arguments is 0, and the result is then immediately obtained since $\gcd(0, d) = d$. The algorithm must terminate since the non-negative remainders are strictly decreasing. Moreover, it is efficient because the number of division steps can be shown to be at most $2k$ where k is the bitlength of a.

The Euclidean algorithm can be extended to find integers x and y such that $ax + by = d$ where $d = \gcd(a, b)$. Algorithm 2.19 maintains the invariants

$$ax_1 + by_1 = u, \quad ax_2 + by_2 = v, \quad u \leq v.$$

The algorithm terminates when $u = 0$, in which case $v = \gcd(a, b)$ and $x = x_2$, $y = y_2$ satisfy $ax + by = d$.

Algorithm 2.19 Extended Euclidean algorithm for integers

INPUT: Positive integers a and b with $a \leq b$.
OUTPUT: $d = \gcd(a, b)$ and integers x, y satisfying $ax + by = d$.

1. $u \leftarrow a$, $v \leftarrow b$.
2. $x_1 \leftarrow 1$, $y_1 \leftarrow 0$, $x_2 \leftarrow 0$, $y_2 \leftarrow 1$.
3. While $u \neq 0$ do
   3.1 $q \leftarrow \lfloor v/u \rfloor$, $r \leftarrow v - qu$, $x \leftarrow x_2 - qx_1$, $y \leftarrow y_2 - qy_1$.
   3.2 $v \leftarrow u$, $u \leftarrow r$, $x_2 \leftarrow x_1$, $x_1 \leftarrow x$, $y_2 \leftarrow y_1$, $y_1 \leftarrow y$.
4. $d \leftarrow v$, $x \leftarrow x_2$, $y \leftarrow y_2$.
5. Return(d, x, y).

Suppose now that p is prime and $a \in [1, p-1]$, and hence $\gcd(a, p) = 1$. If Algorithm 2.19 is executed with inputs (a, p), the last nonzero remainder r encountered in step 3.1 is $r = 1$. Subsequent to this occurrence, the integers u, x_1 and y_1 as updated in step 3.2 satisfy $ax_1 + py_1 = u$ with $u = 1$. Hence $ax_1 \equiv 1 \pmod{p}$ and so $a^{-1} = x_1 \bmod p$. Note that y_1 and y_2 are not needed for the determination of x_1. These observations lead to Algorithm 2.20 for inversion in $\mathbb{F}_p$.

Algorithm 2.20 Inversion in $\mathbb{F}_p$ using the extended Euclidean algorithm

INPUT: Prime p and $a \in [1, p-1]$.
OUTPUT: $a^{-1} \bmod p$.

1. $u \leftarrow a$, $v \leftarrow p$.
2. $x_1 \leftarrow 1$, $x_2 \leftarrow 0$.
3. While $u \neq 1$ do
   3.1 $q \leftarrow \lfloor v/u \rfloor$, $r \leftarrow v - qu$, $x \leftarrow x_2 - qx_1$.
   3.2 $v \leftarrow u$, $u \leftarrow r$, $x_2 \leftarrow x_1$, $x_1 \leftarrow x$.
4. Return$(x_1 \bmod p)$.

Binary inversion algorithm

A drawback of Algorithm 2.20 is the requirement for computationally expensive division operations in step 3.1. The binary inversion algorithm replaces the divisions with cheaper shifts (divisions by 2) and subtractions. The algorithm is an extended version of the binary gcd algorithm which is presented next.

Before each iteration of step 3.1 of Algorithm 2.21, at most one of u and v is odd. Thus the divisions by 2 in steps 3.1 and 3.2 do not change the value of $\gcd(u, v)$. In each iteration, after steps 3.1 and 3.2, both u and v are odd and hence exactly one of u and v will be even at the end of step 3.3. Thus, each iteration of step 3 reduces the bitlength of either u or v by at least one. It follows that the total number of iterations of step 3 is at most $2k$ where k is the maximum of the bitlengths of a and b.

Algorithm 2.21 Binary gcd algorithm

INPUT: Positive integers a and b.
OUTPUT: $\gcd(a, b)$.

1. $u \leftarrow a$, $v \leftarrow b$, $e \leftarrow 1$.
2. While both u and v are even do: $u \leftarrow u/2$, $v \leftarrow v/2$, $e \leftarrow 2e$.
3. While $u \neq 0$ do
   3.1 While u is even do: $u \leftarrow u/2$.
   3.2 While v is even do: $v \leftarrow v/2$.
   3.3 If $u \geq v$ then $u \leftarrow u - v$; else $v \leftarrow v - u$.
4. Return$(e \cdot v)$.

Algorithm 2.22 computes $a^{-1} \bmod p$ by finding an integer x such that $ax + py = 1$. The algorithm maintains the invariants

$$ax_1 + py_1 = u, \quad ax_2 + py_2 = v$$

where y_1 and y_2 are not explicitly computed. The algorithm terminates when $u = 1$ or $v = 1$. In the former case, $ax_1 + py_1 = 1$ and hence $a^{-1} = x_1 \bmod p$. In the latter case, $ax_2 + py_2 = 1$ and $a^{-1} = x_2 \bmod p$.

Algorithm 2.22 Binary algorithm for inversion in $\mathbb{F}_p$

INPUT: Prime p and $a \in [1, p-1]$.
OUTPUT: $a^{-1} \bmod p$.

1. $u \leftarrow a$, $v \leftarrow p$.
2. $x_1 \leftarrow 1$, $x_2 \leftarrow 0$.
3. While ($u \neq 1$ and $v \neq 1$) do
   3.1 While u is even do
   $u \leftarrow u/2$.
   If x_1 is even then $x_1 \leftarrow x_1/2$; else $x_1 \leftarrow (x_1 + p)/2$.
   3.2 While v is even do
   $v \leftarrow v/2$.
   If x_2 is even then $x_2 \leftarrow x_2/2$; else $x_2 \leftarrow (x_2 + p)/2$.
   3.3 If $u \geq v$ then: $u \leftarrow u - v$, $x_1 \leftarrow x_1 - x_2$;
   Else: $v \leftarrow v - u$, $x_2 \leftarrow x_2 - x_1$.
4. If $u = 1$ then return$(x_1 \bmod p)$; else return$(x_2 \bmod p)$.

A division algorithm producing $b/a = ba^{-1} \bmod p$ can be obtained directly from the binary algorithm by changing the initialization condition $x_1 \leftarrow 1$ to $x_1 \leftarrow b$. The running times are expected to be the same, since x_1 in the inversion algorithm is expected to be full-length after a few iterations. Division algorithms are discussed in more detail for binary fields (§2.3) where the lower cost of inversion relative to multiplication makes division especially attractive.

Algorithm 2.22 can be converted to a two-stage inversion method that first finds $a^{-1}2^k \bmod p$ for some integer $k \geq 0$ and then solves for a^{-1}. This alternative is similar to the almost inverse method (Algorithm 2.50) for inversion in binary fields, and permits some optimizations not available in a direct implementation of Algorithm 2.22. The basic method is outlined in the context of the Montgomery inverse below, where the strategy is particularly appropriate.

Montgomery inversion

As outlined in §2.2.4, the basic strategy in Montgomery's method is to replace modular reduction $z \bmod p$ by a less-expensive operation $zR^{-1} \bmod p$ for a suitably chosen R. Montgomery arithmetic can be regarded as operating on representatives $\widetilde{x} = xR \bmod p$, and is applicable in calculations such as modular exponentiation where the required initial and final conversions $x \mapsto \widetilde{x}$ and $\widetilde{x} \mapsto \widetilde{x}R^{-1} \bmod p = x$ are an insignificant portion of the overall computation.

Let $p > 2$ be an odd (but possibly composite) integer, and define $n = \lceil \log_2 p \rceil$. The *Montgomery inverse* of an integer a with $\gcd(a, p) = 1$ is $a^{-1}2^n \bmod p$. Algorithm 2.23 is a modification of the binary algorithm (Algorithm 2.22), and computes $a^{-1}2^k \bmod p$ for some integer $k \in [n, 2n]$.

Algorithm 2.23 Partial Montgomery inversion in $\mathbb{F}_p$

INPUT: Odd integer $p > 2$, $a \in [1, p-1]$, and $n = \lceil \log_2 p \rceil$.
OUTPUT: Either "not invertible" or (x, k) where $n \leq k \leq 2n$ and $x = a^{-1}2^k \bmod p$.

1. $u \leftarrow a$, $v \leftarrow p$, $x_1 \leftarrow 1$, $x_2 \leftarrow 0$, $k \leftarrow 0$.
2. While $v > 0$ do
   2.1 If v is even then $v \leftarrow v/2$, $x_1 \leftarrow 2x_1$;
   else if u is even then $u \leftarrow u/2$, $x_2 \leftarrow 2x_2$;
   else if $v \geq u$ then $v \leftarrow (v-u)/2$, $x_2 \leftarrow x_2 + x_1$, $x_1 \leftarrow 2x_1$;
   else $u \leftarrow (u-v)/2$, $x_1 \leftarrow x_2 + x_1$, $x_2 \leftarrow 2x_2$.
   2.2 $k \leftarrow k + 1$.
3. If $u \neq 1$ then return("not invertible").
4. If $x_1 > p$ then $x_1 \leftarrow x_1 - p$.
5. Return(x_1, k).

For invertible a, the Montgomery inverse $a^{-1}2^n \bmod p$ may be obtained from the output (x, k) by $k - n$ repeated divisions of the form:

$$\text{if } x \text{ is even then } x \leftarrow x/2; \text{ else } x \leftarrow (x + p)/2. \tag{2.2}$$

Compared with the binary method (Algorithm 2.22) for producing the ordinary inverse, Algorithm 2.23 has simpler updating of the variables x_1 and x_2, although $k - n$ of the more expensive updates occur in (2.2).

Note 2.24 (*correctness of and implementation considerations for Algorithm 2.23*)

(i) In addition to $\gcd(u, v) = \gcd(a, p)$, the invariants

$$ax_1 \equiv u2^k \pmod{p} \quad \text{and} \quad ax_2 \equiv -v2^k \pmod{p}$$

are maintained. If $\gcd(a, p) = 1$, then $u = 1$ and $x_1 \equiv a^{-1}2^k \pmod{p}$ at the last iteration of step 2.

(ii) Until the last iteration, the conditions

$$p = vx_1 + ux_2, \quad x_1 \geq 1, \quad v \geq 1, \quad 0 \leq u \leq a,$$

hold, and hence $x_1, v \in [1, p]$. At the last iteration, $x_1 \leftarrow 2x_1 \leq 2p$; if $\gcd(a, p) = 1$, then necessarily $x_1 < 2p$ and step 4 ensures $x_1 < p$. Unlike Algorithm 2.22, the variables x_1 and x_2 grow slowly, possibly allowing some implementation optimizations.

(iii) Each iteration of step 2 reduces the product uv by at least half and the sum $u + v$ by at most half. Initially $u + v = a + p$ and $uv = ap$, and $u = v = 1$ before the final iteration. Hence $(a + p)/2 \leq 2^{k-1} \leq ap$, and it follows that $2^{n-2} < 2^{k-1} < 2^{2n}$ and $n \leq k \leq 2n$.

Montgomery arithmetic commonly selects $R = 2^{Wt} \geq 2^n$ for efficiency and uses representatives $\widetilde{x} = xR \bmod p$. The Montgomery product $\text{Mont}(\widetilde{x}, \widetilde{y})$ of $\widetilde{x}$ and $\widetilde{y}$ is as defined in (2.1). The second stage (2.2) can be modified to use Montgomery multiplication to produce $a^{-1} \bmod p$ or $a^{-1}R \bmod p$ (rather than $a^{-1}2^n \bmod p$) from a, or to calculate $a^{-1}R \bmod p$ when Algorithm 2.23 is presented with $\widetilde{a}$ rather than a. Algorithm 2.25 is applicable in elliptic curve point multiplication (§3.3) if Montgomery arithmetic is used with affine coordinates.

Algorithm 2.25 Montgomery inversion in $\mathbb{F}_p$

INPUT: Odd integer $p > 2$, $n = \lceil \log_2 p \rceil$, $R^2 \bmod p$, and $\widetilde{a} = aR \bmod p$ with $\gcd(a, p) = 1$.
OUTPUT: $a^{-1}R \bmod p$.

1. Use Algorithm 2.23 to find (x, k) where $x = \widetilde{a}^{-1}2^k \bmod p$ and $n \leq k \leq 2n$.
2. If $k < Wt$ then
   2.1 $x \leftarrow \text{Mont}(x, R^2) = a^{-1}2^k \bmod p$.
   2.2 $k \leftarrow k + Wt$. {Now, $k > Wt$.}
3. $x \leftarrow \text{Mont}(x, R^2) = a^{-1}2^k \bmod p$.
4. $x \leftarrow \text{Mont}(x, 2^{2Wt-k}) = a^{-1}R \bmod p$.
5. Return(x).

The value $a^{-1}R \equiv R^2/(aR) \pmod{p}$ may also be obtained by a division algorithm variant of Algorithm 2.22 with inputs $R^2 \bmod p$ and $\widetilde{a}$. However, Algorithm 2.25 may have implementation advantages, and the Montgomery multiplications required are expected to be relatively inexpensive compared to the cost of inversion.

Simultaneous inversion

Field inversion tends to be expensive relative to multiplication. If inverses are required for several elements, then the method of simultaneous inversion finds the inverses with a single inversion and approximately three multiplications per element. The method is based on the observation that $1/x = y(1/xy)$ and $1/y = x(1/xy)$, which is generalized in Algorithm 2.26 to k elements.

Algorithm 2.26 Simultaneous inversion

INPUT: Prime p and nonzero elements $a_1, \ldots, a_k$ in $\mathbb{F}_p$
OUTPUT: Field elements $a_1^{-1}, \ldots, a_k^{-1}$, where $a_i a_i^{-1} \equiv 1 \pmod{p}$.

1. $c_1 \leftarrow a_1$.
2. For i from 2 to k do: $c_i \leftarrow c_{i-1} a_i \bmod p$.
3. $u \leftarrow c_k^{-1} \bmod p$.
4. For i from k downto 2 do
   4.1 $a_i^{-1} \leftarrow u c_{i-1} \bmod p$.
   4.2 $u \leftarrow u a_i \bmod p$.
5. $a_1^{-1} \leftarrow u$.
6. Return$(a_1^{-1}, \ldots, a_k^{-1})$.

For k elements, the algorithm requires one inversion and $3(k-1)$ multiplications, along with k elements of temporary storage. Although the algorithm is presented in the context of prime fields, the technique can be adapted to other fields and is superior to k separate inversions whenever the cost of an inversion is higher than that of three multiplications.

2.2.6 NIST primes

The FIPS 186-2 standard recommends elliptic curves over the five prime fields with moduli:

$$\begin{aligned}
p_{192} &= 2^{192} - 2^{64} - 1 \\
p_{224} &= 2^{224} - 2^{96} + 1 \\
p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\
p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \\
p_{521} &= 2^{521} - 1.
\end{aligned}$$

These primes have the property that they can be written as the sum or difference of a small number of powers of 2. Furthermore, except for p_{521}, the powers appearing in these expressions are all multiples of 32. These properties yield reduction algorithms that are especially fast on machines with wordsize 32.

For example, consider $p = p_{192} = 2^{192} - 2^{64} - 1$, and let c be an integer with $0 \leq c < p^2$. Let

$$c = c_5 2^{320} + c_4 2^{256} + c_3 2^{192} + c_2 2^{128} + c_1 2^{64} + c_0 \tag{2.3}$$

be the base-2^{64} representation of c, where each $c_i \in [0, 2^{64} - 1]$. We can then reduce the higher powers of 2 in (2.3) using the congruences

$$\begin{aligned} 2^{192} &\equiv 2^{64} + 1 \pmod{p} \\ 2^{256} &\equiv 2^{128} + 2^{64} \pmod{p} \\ 2^{320} &\equiv 2^{128} + 2^{64} + 1 \pmod{p}. \end{aligned}$$

We thus obtain

$$\begin{aligned} c \equiv \quad & c_5 2^{128} + c_5 2^{64} + c_5 \\ & + c_4 2^{128} + c_4 2^{64} \\ & + \qquad\quad c_3 2^{64} + c_3 \\ & + c_2 2^{128} + c_1 2^{64} + c_0 \pmod{p}. \end{aligned}$$

Hence, c modulo p can be obtained by adding the four 192-bit integers $c_5 2^{128} + c_5 2^{64} + c_5$, $c_4 2^{128} + c_4 2^{64}$, $c_3 2^{64} + c_3$ and $c_2 2^{128} + c_1 2^{64} + c_0$, and repeatedly subtracting p until the result is less than p.

Algorithm 2.27 Fast reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$

INPUT: An integer $c = (c_5, c_4, c_3, c_2, c_1, c_0)$ in base 2^{64} with $0 \leq c < p_{192}^2$.
OUTPUT: $c \bmod p_{192}$.

1. Define 192-bit integers:
   $s_1 = (c_2, c_1, c_0)$, $s_2 = (0, c_3, c_3)$,
   $s_3 = (c_4, c_4, 0)$, $s_4 = (c_5, c_5, c_5)$.
2. Return$(s_1 + s_2 + s_3 + s_4 \bmod p_{192})$.

Algorithm 2.28 Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$

INPUT: An integer $c = (c_{13}, \ldots, c_2, c_1, c_0)$ in base 2^{32} with $0 \leq c < p_{224}^2$.
OUTPUT: $c \bmod p_{224}$.

1. Define 224-bit integers:
   $s_1 = (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$, $s_2 = (c_{10}, c_9, c_8, c_7, 0, 0, 0)$,
   $s_3 = (0, c_{13}, c_{12}, c_{11}, 0, 0, 0)$, $s_4 = (c_{13}, c_{12}, c_{11}, c_{10}, c_9, c_8, c_7)$,
   $s_5 = (0, 0, 0, 0, c_{13}, c_{12}, c_{11})$.
2. Return$(s_1 + s_2 + s_3 - s_4 - s_5 \bmod p_{224})$.

Algorithm 2.29 Fast reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

INPUT: An integer $c = (c_{15}, \ldots, c_2, c_1, c_0)$ in base 2^{32} with $0 \leq c < p_{256}^2$.
OUTPUT: $c \bmod p_{256}$.

1. Define 256-bit integers:
   $s_1 = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
   $s_2 = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0)$,
   $s_3 = (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0)$,
   $s_4 = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$,
   $s_5 = (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$,
   $s_6 = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$,
   $s_7 = (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$,
   $s_8 = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13})$,
   $s_9 = (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$.
2. Return$(s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \bmod p_{256})$.

Algorithm 2.30 Fast reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

INPUT: An integer $c = (c_{23}, \ldots, c_2, c_1, c_0)$ in base 2^{32} with $0 \leq c < p_{384}^2$.
OUTPUT: $c \bmod p_{384}$.

1. Define 384-bit integers:
   $s_1 = (c_{11}, c_{10}, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
   $s_2 = (0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, 0, 0)$,
   $s_3 = (c_{23}, c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12})$,
   $s_4 = (c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23}, c_{22}, c_{21})$,
   $s_5 = (c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{20}, 0, c_{23}, 0)$,
   $s_6 = (0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0, 0, 0, 0)$,
   $s_7 = (0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, c_{20})$,
   $s_8 = (c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23})$,
   $s_9 = (0, 0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0)$,
   $s_{10} = (0, 0, 0, 0, 0, 0, 0, c_{23}, c_{23}, 0, 0, 0)$.
2. Return$(s_1 + 2s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10} \bmod p_{384})$.

Algorithm 2.31 Fast reduction modulo $p_{521} = 2^{521} - 1$

INPUT: An integer $c = (c_{1041}, \ldots, c_2, c_1, c_0)$ in base 2 with $0 \leq c < p_{521}^2$.
OUTPUT: $c \bmod p_{521}$.

1. Define 521-bit integers:
   $s_1 = (c_{1041}, \ldots, c_{523}, c_{522}, c_{521})$,
   $s_2 = (c_{520}, \ldots, c_2, c_1, c_0)$.
2. Return$(s_1 + s_2 \bmod p_{521})$.

2.3 Binary field arithmetic

This section presents algorithms that are suitable for performing binary field arithmetic in software. Chapter 5 includes additional material on use of single-instruction multiple-data (SIMD) registers found on some processors (§5.1.3), and on design considerations for hardware implementation (§5.2.2). Selected timings for field operations appear in §5.1.5.

We assume that the implementation platform has a W-bit architecture where W is a multiple of 8. The bits of a W-bit word U are numbered from 0 to $W-1$, with the rightmost bit of U designated as bit 0. The following standard notation is used to denote operations on words U and V:

$U \oplus V$	bitwise exclusive-or
$U \,\&\, V$	bitwise AND
$U \gg i$	right shift of U by i positions with the i high-order bits set to 0
$U \ll i$	left shift of U by i positions with the i low-order bits set to 0.

Let $f(z)$ be an irreducible binary polynomial of degree m, and write $f(z) = z^m + r(z)$. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$. Addition of field elements is the usual addition of binary polynomials. Multiplication is performed modulo $f(z)$. A field element $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ of length m. Let $t = \lceil m/W \rceil$, and let $s = Wt - m$. In software, a may be stored in an array of t W-bit words: $A = (A[t-1], \ldots, A[2], A[1], A[0])$, where the rightmost bit of $A[0]$ is a_0, and the leftmost s bits of $A[t-1]$ are unused (always set to 0).

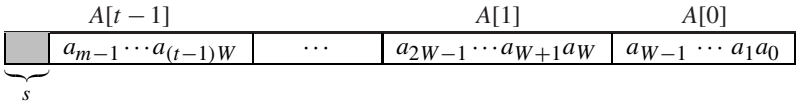


Figure 2.4. Representation of $a \in \mathbb{F}_{2^m}$ as an array A of W-bit words. The $s = tW - m$ highest order bits of $A[t-1]$ remain unused.

2.3.1 Addition

Addition of field elements is performed bitwise, thus requiring only t word operations.

Algorithm 2.32 Addition in $\mathbb{F}_{2^m}$

INPUT: Binary polynomials $a(z)$ and $b(z)$ of degrees at most $m-1$.
OUTPUT: $c(z) = a(z) + b(z)$.

1. For i from 0 to $t-1$ do
   1.1 $C[i] \leftarrow A[i] \oplus B[i]$.
2. Return(c).

2.3.2 Multiplication

The shift-and-add method (Algorithm 2.33) for field multiplication is based on the observation that

$$a(z) \cdot b(z) = a_{m-1}z^{m-1}b(z) + \cdots + a_2 z^2 b(z) + a_1 z b(z) + a_0 b(z).$$

Iteration i in the algorithm computes $z^i b(z) \bmod f(z)$ and adds the result to the accumulator c if $a_i = 1$. If $b(z) = b_{m-1}z^{m-1} + \cdots + b_2 z^2 + b_1 z + b_0$, then

$$\begin{aligned} b(z) \cdot z &= b_{m-1}z^m + b_{m-2}z^{m-1} + \cdots + b_2 z^3 + b_1 z^2 + b_0 z \\ &\equiv b_{m-1}r(z) + (b_{m-2}z^{m-1} + \cdots + b_2 z^3 + b_1 z^2 + b_0 z) \pmod{f(z)}. \end{aligned}$$

Thus $b(z) \cdot z \bmod f(z)$ can be computed by a left-shift of the vector representation of $b(z)$, followed by addition of $r(z)$ to $b(z)$ if the high order bit b_{m-1} is 1.

Algorithm 2.33 Right-to-left shift-and-add field multiplication in $\mathbb{F}_{2^m}$

INPUT: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$.
OUTPUT: $c(z) = a(z) \cdot b(z) \bmod f(z)$.

1. If $a_0 = 1$ then $c \leftarrow b$; else $c \leftarrow 0$.
2. For i from 1 to $m-1$ do
   2.1 $b \leftarrow b \cdot z \bmod f(z)$.
   2.2 If $a_i = 1$ then $c \leftarrow c + b$.
3. Return(c).

While Algorithm 2.33 is well-suited for hardware where a vector shift can be performed in one clock cycle, the large number of word shifts make it less desirable for software implementation. We next consider faster methods for field multiplication which first multiply the field elements as polynomials (§2.3.3 and §2.3.4), and then reduce the result modulo $f(z)$ (§2.3.5).

2.3.3 Polynomial multiplication

The *right-to-left comb method* (Algorithm 2.34) for polynomial multiplication is based on the observation that if $b(z) \cdot z^k$ has been computed for some $k \in [0, W-1]$, then $b(z) \cdot z^{Wj+k}$ can be easily obtained by appending j zero words to the right of the vector representation of $b(z) \cdot z^k$. Algorithm 2.34 processes the bits of the words of A from right to left, as shown in Figure 2.5 when the parameters are $m = 163$, $W = 32$. The following notation is used: if $C = (C[n], \ldots, C[2], C[1], C[0])$ is an array, then $C\{j\}$ denotes the truncated array $(C[n], \ldots, C[j+1], C[j])$.

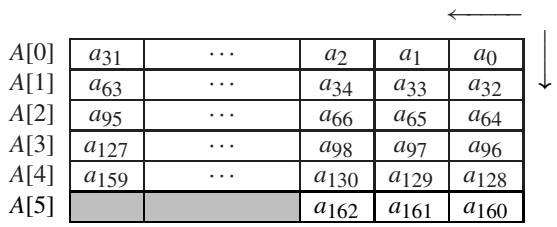


Figure 2.5. The right-to-left comb method (Algorithm 2.34) processes the columns of the exponent array for a right-to-left. The bits in a column are processed from top to bottom. Example parameters are $W = 32$ and $m = 163$.

Algorithm 2.34 Right-to-left comb method for polynomial multiplication

INPUT: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$.
OUTPUT: $c(z) = a(z) \cdot b(z)$.

1. $C \leftarrow 0$.
2. For k from 0 to $W-1$ do
   2.1 For j from 0 to $t-1$ do
   If the kth bit of $A[j]$ is 1 then add B to $C\{j\}$.
   2.2 If $k \neq (W-1)$ then $B \leftarrow B \cdot z$.
3. Return(C).

The left-to-right comb method for polynomial multiplication processes the bits of a from left to right as follows:

$$a(z) \cdot b(z) = \Big(\cdots \big((a_{m-1}b(z)z + a_{m-2}b(z))z + a_{m-3}b(z) \big) z + \cdots + a_1 b(z) \Big) z + a_0 b(z).$$

Algorithm 2.35 is a modification of this method where the bits of the words of A are processed from left to right. This is illustrated in Figure 2.6 when $m = 163$, $W = 32$ are the parameters.

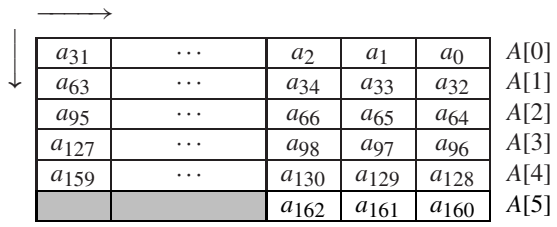


Figure 2.6. The left-to-right comb method (Algorithm 2.35) processes the columns of the exponent array for a left-to-right. The bits in a column are processed from top to bottom. Example parameters are $W = 32$ and $m = 163$.

Algorithm 2.35 Left-to-right comb method for polynomial multiplication

INPUT: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$.
OUTPUT: $c(z) = a(z) \cdot b(z)$.

1. $C \leftarrow 0$.
2. For k from $W-1$ downto 0 do
   2.1 For j from 0 to $t-1$ do
   If the kth bit of $A[j]$ is 1 then add B to $C\{j\}$.
   2.2 If $k \neq 0$ then $C \leftarrow C \cdot z$.
3. Return(C).

Algorithms 2.34 and 2.35 are both faster than Algorithm 2.33 since there are fewer vector shifts (multiplications by z). Algorithm 2.34 is faster than Algorithm 2.35 since the vector shifts in the former involve the t-word array B (which can grow to size $t+1$), while the vector shifts in the latter involve the $2t$-word array C.

Algorithm 2.35 can be accelerated considerably at the expense of some storage overhead by first computing $u(z) \cdot b(z)$ for all polynomials $u(z)$ of degree less than w, and then processing the bits of $A[j]$ w at a time. The modified method is presented as Algorithm 2.36. The order in which the bits of a are processed is shown in Figure 2.7 when the parameters are $M = 163$, $W = 32$, $w = 4$.

Algorithm 2.36 Left-to-right comb method with windows of width w

INPUT: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$.
OUTPUT: $c(z) = a(z) \cdot b(z)$.

1. Compute $B_u = u(z) \cdot b(z)$ for all polynomials $u(z)$ of degree at most $w-1$.
2. $C \leftarrow 0$.
3. For k from $(W/w)-1$ downto 0 do
   3.1 For j from 0 to $t-1$ do
   Let $u = (u_{w-1}, \ldots, u_1, u_0)$, where u_i is bit $(wk+i)$ of $A[j]$.
   Add B_u to $C\{j\}$.
   3.2 If $k \neq 0$ then $C \leftarrow C \cdot z^w$.
4. Return(C).

As written, Algorithm 2.36 performs polynomial multiplication—modular reduction for field multiplication is performed separately. In some situations, it may be advantageous to include the reduction polynomial f as an input to the algorithm. Step 1 may then be modified to calculate $ub \bmod f$, which may allow optimizations in step 3.

Note 2.37 (*enhancements to Algorithm 2.36*) Depending on processor characteristics, one potentially useful variation of Algorithm 2.36 exchanges shifts for additions and table lookups. Precomputation is split into l tables; for simplicity, we assume $l \mid w$. Table i, $0 \leq i < l$, consists of values $B_{v,i} = v(z)z^{iw/l}b(z)$ for all polynomials v of degree

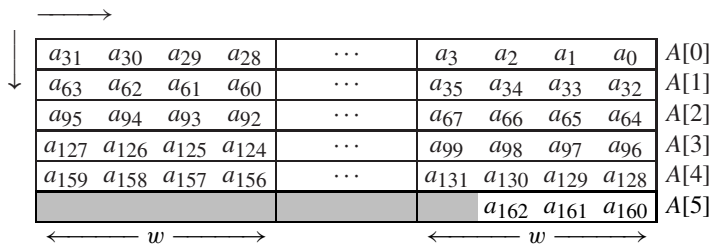


Figure 2.7. Algorithm 2.36 processes columns of the exponent array for a left-to-right. The entries within a width w column are processed from top to bottom. Example parameters are $W = 32$, $m = 163$, *and* $w = 4$.

less than w/l. Step 3.1 of Algorithm 2.36 is modified to calculate $B_u = \sum_{i=0}^{l-1} B_{u^i,i}$ where $u = (u_{w-1}, \ldots, u_0) = (u^{l-1}, \ldots, u^0)$ and u^i has w/l bits. As an example, Algorithm 2.36 with $w = 4$ has 16 elements of precomputation. The modified algorithm with parameters $w = 8$ and $l = 4$ has the same amount of precomputation (four tables of four points each). Compared with the original algorithm, there are fewer iterations at step 3 (and hence fewer shifts at step 3.2); however, step 3.1 is more expensive.

The comb methods are due to López and Dahab, and are based on the observation that the exponentiation methods of Lim and Lee can be adapted for use in binary fields. §3.3.2 discusses Lim-Lee methods in more detail in the context of elliptic curve point multiplication; see Note 3.47.

Karatsuba-Ofman multiplication

The divide-and-conquer method of Karatsuba-Ofman outlined in §2.2.2 can be directly adapted for the polynomial case. For example,

$$\begin{aligned} a(z)b(z) &= (A_1 z^l + A_0)(B_1 z^l + B_0) \\ &= A_1 B_1 z^{2l} + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^l + A_0 B_0 \end{aligned}$$

where $l = \lceil m/2 \rceil$ and the coefficients A_0, A_1, B_0, B_1 are binary polynomials in z of degree less than l. The process may be repeated, using table-lookup or other methods at some threshold. The overhead, however, is often sufficient to render such strategies inferior to Algorithm 2.36 for m of practical interest.

Note 2.38 (*implementing polynomial multiplication*) Algorithm 2.36 appears to be among the fastest in practice for binary fields of interest in elliptic curve methods, provided that the hardware characteristics are targeted reasonably accurately. The code produced by various C compilers can differ dramatically in performance, and compilers can be sensitive to the precise form in which the algorithm is written.

The contribution by Sun Microsystems Laboratories (SML) to the OpenSSL project in 2002 provides a case study of the compromises chosen in practice. OpenSSL is widely used to provide cryptographic services for the Apache web server and the OpenSSH secure shell communication tool. SML's contribution must be understood in context: OpenSSL is a public and collaborative effort—it is likely that Sun's proprietary code has significant enhancements.

To keep the code size relatively small, SML implemented a fairly generic polynomial multiplication method. Karatsuba-Ofman is used, but only on multiplication of 2-word quantities rather than recursive application. At the lowest level of multiplication of 1-word quantities, a simplified Algorithm 2.36 is applied (with $w = 2$, $w = 3$, and $w = 4$ on 16-bit, 32-bit, and 64-bit platforms, respectively). As expected, the result tends to be much slower than the fastest versions of Algorithm 2.36. In our tests on Sun SPARC and Intel P6-family hardware, the Karatsuba-Ofman method implemented is less efficient than use of Algorithm 2.36 at the 2-word stage. However, the contribution from SML may be a better compromise in OpenSSL if the same code is used across platforms and compilers.

2.3.4 Polynomial squaring

Since squaring a binary polynomial is a linear operation, it is much faster than multiplying two arbitrary polynomials; i.e., if $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$, then

$$a(z)^2 = a_{m-1}z^{2m-2} + \cdots + a_2z^4 + a_1z^2 + a_0.$$

The binary representation of $a(z)^2$ is obtained by inserting a 0 bit between consecutive bits of the binary representation of $a(z)$ as shown in Figure 2.8. To facilitate this process, a table T of size 512 bytes can be precomputed for converting 8-bit polynomials into their expanded 16-bit counterparts. Algorithm 2.39 describes this procedure for the parameter $W = 32$.

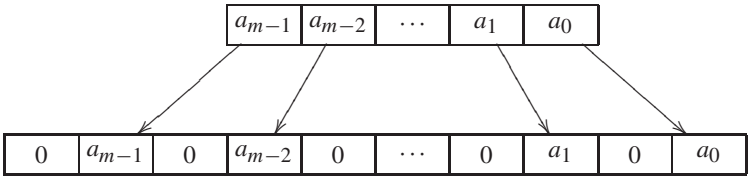


Figure 2.8. Squaring a binary polynomial $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$.

Algorithm 2.39 Polynomial squaring (with wordlength $W = 32$)

INPUT: A binary polynomial $a(z)$ of degree at most $m-1$.
OUTPUT: $c(z) = a(z)^2$.

1. *Precomputation*. For each byte $d = (d_7, \ldots, d_1, d_0)$, compute the 16-bit quantity $T(d) = (0, d_7, \ldots, 0, d_1, 0, d_0)$.
2. For i from 0 to $t-1$ do
   2.1 Let $A[i] = (u_3, u_2, u_1, u_0)$ where each u_j is a byte.
   2.2 $C[2i] \leftarrow (T(u_1), T(u_0))$, $C[2i+1] \leftarrow (T(u_3), T(u_2))$.
3. Return(c).

2.3.5 Reduction

We now discuss techniques for reducing a binary polynomial $c(z)$ obtained by multiplying two binary polynomials of degree $\leq m-1$, or by squaring a binary polynomial of degree $\leq m-1$. Such polynomials $c(z)$ have degree at most $2m-2$.

Arbitrary reduction polynomials

Recall that $f(z) = z^m + r(z)$, where $r(z)$ is a binary polynomial of degree at most $m-1$. Algorithm 2.40 reduces $c(z)$ modulo $f(z)$ one bit at a time, starting with the leftmost bit. It is based on the observation that

$$\begin{aligned} c(z) &= c_{2m-2}z^{2m-2} + \cdots + c_m z^m + c_{m-1}z^{m-1} + \cdots + c_1 z + c_0 \\ &\equiv (c_{2m-2}z^{m-2} + \cdots + c_m)r(z) + c_{m-1}z^{m-1} + \cdots + c_1 z + c_0 \pmod{f(z)}. \end{aligned}$$

The reduction is accelerated by precomputing the polynomials $z^k r(z)$, $0 \leq k \leq W-1$. If $r(z)$ is a low-degree polynomial, or if $f(z)$ is a trinomial, then the space requirements are smaller, and furthermore the additions involving $z^k r(z)$ in step 2.1 are faster. The following notation is used: if $C = (C[n], \ldots, C[2], C[1], C[0])$ is an array, then $C\{j\}$ denotes the truncated array $(C[n], \ldots, C[j+1], C[j])$.

Algorithm 2.40 Modular reduction (one bit at a time)

INPUT: A binary polynomial $c(z)$ of degree at most $2m-2$.
OUTPUT: $c(z) \bmod f(z)$.

1. *Precomputation*. Compute $u_k(z) = z^k r(z)$, $0 \leq k \leq W-1$.
2. For i from $2m-2$ downto m do
   2.1 If $c_i = 1$ then
   Let $j = \lfloor (i-m)/W \rfloor$ and $k = (i-m) - Wj$.
   Add $u_k(z)$ to $C\{j\}$.
3. Return($C[t-1], \ldots, C[1], C[0]$).

If $f(z)$ is a trinomial, or a pentanomial with middle terms close to each other, then reduction of $c(z)$ modulo $f(z)$ can be efficiently performed one word at a time. For example, suppose $m = 163$ and $W = 32$ (so $t = 6$), and consider reducing the word $C[9]$ of $c(z)$ modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$. The word $C[9]$ represents the polynomial $c_{319}z^{319} + \cdots + c_{289}z^{289} + c_{288}z^{288}$. We have

$$\begin{aligned} z^{288} &\equiv z^{132} + z^{131} + z^{128} + z^{125} \pmod{f(z)}, \\ z^{289} &\equiv z^{133} + z^{132} + z^{129} + z^{126} \pmod{f(z)}, \\ &\vdots \\ z^{319} &\equiv z^{163} + z^{162} + z^{159} + z^{156} \pmod{f(z)}. \end{aligned}$$

By considering the four columns on the right side of the above congruences, we see that reduction of $C[9]$ can be performed by adding $C[9]$ four times to C, with the rightmost bit of $C[9]$ added to bits 132, 131, 128 and 125 of C; this is illustrated in Figure 2.9.

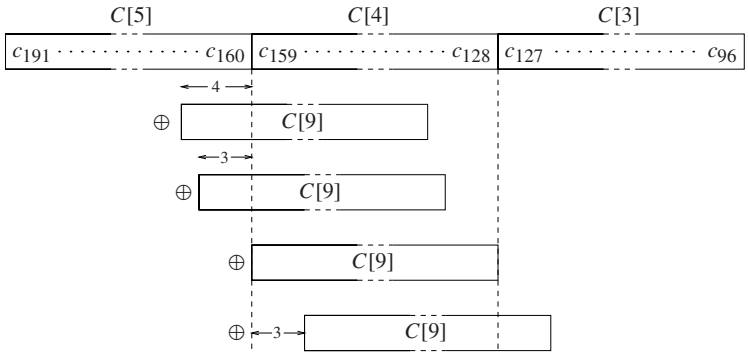


Figure 2.9. Reducing the 32-bit word $C[9]$ modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$.

NIST reduction polynomials

We next present algorithms for fast reduction modulo the following reduction polynomials recommended by NIST in the FIPS 186-2 standard:

$$\begin{aligned} f(z) &= z^{163} + z^7 + z^6 + z^3 + 1 \\ f(z) &= z^{233} + z^{74} + 1 \\ f(z) &= z^{283} + z^{12} + z^7 + z^5 + 1 \\ f(z) &= z^{409} + z^{87} + 1 \\ f(z) &= z^{571} + z^{10} + z^5 + z^2 + 1. \end{aligned}$$

These algorithms, which assume a wordlength $W = 32$, are based on ideas similar to those leading to Figure 2.9. They are faster than Algorithm 2.40 and furthermore have no storage overhead.

Algorithm 2.41 Fast reduction modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$ (with $W = 32$)

INPUT: A binary polynomial $c(z)$ of degree at most 324.
OUTPUT: $c(z) \bmod f(z)$.

1. For i from 10 downto 6 do {Reduce $C[i]z^{32i}$ modulo $f(z)$}
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-6] \leftarrow C[i-6] \oplus (T \ll 29)$.
   1.3 $C[i-5] \leftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
   1.4 $C[i-4] \leftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$.
2. $T \leftarrow C[5] \gg 3$. {Extract bits 3–31 of $C[5]$}
3. $C[0] \leftarrow C[0] \oplus (T \ll 7) \oplus (T \ll 6) \oplus (T \ll 3) \oplus T$.
4. $C[1] \leftarrow C[1] \oplus (T \gg 25) \oplus (T \gg 26)$.
5. $C[5] \leftarrow C[5] \,\&\, 0\text{x}7$. {Clear the reduced bits of $C[5]$}
6. Return $(C[5], C[4], C[3], C[2], C[1], C[0])$.

Algorithm 2.42 Fast reduction modulo $f(z) = z^{233} + z^{74} + 1$ (with $W = 32$)

INPUT: A binary polynomial $c(z)$ of degree at most 464.
OUTPUT: $c(z) \bmod f(z)$.

1. For i from 15 downto 8 do {Reduce $C[i]z^{32i}$ modulo $f(z)$}
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-8] \leftarrow C[i-8] \oplus (T \ll 23)$.
   1.3 $C[i-7] \leftarrow C[i-7] \oplus (T \gg 9)$.
   1.4 $C[i-5] \leftarrow C[i-5] \oplus (T \ll 1)$.
   1.5 $C[i-4] \leftarrow C[i-4] \oplus (T \gg 31)$.
2. $T \leftarrow C[7] \gg 9$. {Extract bits 9–31 of $C[7]$}
3. $C[0] \leftarrow C[0] \oplus T$.
4. $C[2] \leftarrow C[2] \oplus (T \ll 10)$.
5. $C[3] \leftarrow C[3] \oplus (T \gg 22)$.
6. $C[7] \leftarrow C[7] \,\&\, 0\text{x}1\text{FF}$. {Clear the reduced bits of $C[7]$}
7. Return $(C[7], C[6], C[5], C[4], C[3], C[2], C[1], C[0])$.

Algorithm 2.43 Fast reduction modulo $f(z) = z^{283} + z^{12} + z^7 + z^5 + 1$ (with $W = 32$)

INPUT: A binary polynomial $c(z)$ of degree at most 564.
OUTPUT: $c(z) \bmod f(z)$.

1. For i from 17 downto 9 do {Reduce $C[i]z^{32i}$ modulo $f(z)$}
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-9] \leftarrow C[i-9] \oplus (T \ll 5) \oplus (T \ll 10) \oplus (T \ll 12) \oplus (T \ll 17)$.
   1.3 $C[i-8] \leftarrow C[i-8] \oplus (T \gg 27) \oplus (T \gg 22) \oplus (T \gg 20) \oplus (T \gg 15)$.
2. $T \leftarrow C[8] \gg 27$. {Extract bits 27–31 of $C[8]$}
3. $C[0] \leftarrow C[0] \oplus T \oplus (T \ll 5) \oplus (T \ll 7) \oplus (T \ll 12)$.
4. $C[8] \leftarrow C[8]$ & 0x7FFFFFF. {Clear the reduced bits of $C[8]$}
5. Return $(C[8], C[7], C[6], C[5], C[4], C[3], C[2], C[1], C[0])$.

Algorithm 2.44 Fast reduction modulo $f(z) = z^{409} + z^{87} + 1$ (with $W = 32$)

INPUT: A binary polynomial $c(z)$ of degree at most 816.
OUTPUT: $c(z) \bmod f(z)$.

1. For i from 25 downto 13 do {Reduce $C[i]z^{32i}$ modulo $f(z)$}
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-13] \leftarrow C[i-13] \oplus (T \ll 7)$.
   1.3 $C[i-12] \leftarrow C[i-12] \oplus (T \gg 25)$.
   1.4 $C[i-11] \leftarrow C[i-11] \oplus (T \ll 30)$.
   1.5 $C[i-10] \leftarrow C[i-10] \oplus (T \gg 2)$.
2. $T \leftarrow C[12] \gg 25$. {Extract bits 25–31 of $C[12]$}
3. $C[0] \leftarrow C[0] \oplus T$.
4. $C[2] \leftarrow C[2] \oplus$ (T¡¡23).
5. $C[12] \leftarrow C[12]$ & 0x1FFFFFF. {Clear the reduced bits of $C[12]$}
6. Return $(C[12], C[11], \ldots, C[1], C[0])$.

Algorithm 2.45 Fast reduction modulo $f(z) = z^{571} + z^{10} + z^5 + z^2 + 1$ (with $W = 32$)

INPUT: A binary polynomial $c(z)$ of degree at most 1140.
OUTPUT: $c(z) \bmod f(z)$.

1. For i from 35 downto 18 do {Reduce $C[i]z^{32i}$ modulo $f(z)$}
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-18] \leftarrow C[i-18] \oplus (T \ll 5) \oplus (T \ll 7) \oplus (T \ll 10) \oplus (T \ll 15)$.
   1.3 $C[i-17] \leftarrow C[i-17] \oplus (T \gg 27) \oplus (T \gg 25) \oplus (T \gg 22) \oplus (T \gg 17)$.
2. $T \leftarrow C[17] \gg 27$. {Extract bits 27–31 of $C[17]$}
3. $C[0] \leftarrow C[0] \oplus T \oplus (T \ll 2) \oplus (T \ll 5) \oplus (T \ll 10)$.
4. $C[17] \leftarrow C[17]$ & 0x7FFFFFFF. {Clear the reduced bits of $C[17]$}
5. Return $(C[17], C[16], \ldots, C[1], C[0])$.

2.3.6 Inversion and division

In this subsection, we simplify the notation and denote binary polynomials $a(z)$ by a. Recall that the inverse of a nonzero element $a \in \mathbb{F}_{2^m}$ is the unique element $g \in \mathbb{F}_{2^m}$ such that $ag = 1$ in $\mathbb{F}_{2^m}$, that is, $ag \equiv 1 \pmod{f}$. This inverse element is denoted $a^{-1} \bmod f$ or simply a^{-1} if the reduction polynomial f is understood from context. Inverses can be efficiently computed by the extended Euclidean algorithm for polynomials.

The extended Euclidean algorithm for polynomials

Let a and b be binary polynomials, not both 0. The *greatest common divisor (gcd)* of a and b, denoted $\gcd(a, b)$, is the binary polynomial d of highest degree that divides both a and b. Efficient algorithms for computing $\gcd(a, b)$ exploit the following polynomial analogue of Theorem 2.18.

Theorem 2.46 Let a and b be binary polynomials. Then $\gcd(a, b) = \gcd(b - ca, a)$ for all binary polynomials c.

In the classical Euclidean algorithm for computing the gcd of binary polynomials a and b, where $\deg(b) \geq \deg(a)$, b is divided by a to obtain a quotient q and a remainder r satisfying $b = qa + r$ and $\deg(r) < \deg(a)$. By Theorem 2.46, $\gcd(a, b) = \gcd(r, a)$. Thus, the problem of determining $\gcd(a, b)$ is reduced to that of computing $\gcd(r, a)$ where the arguments (r, a) have lower degrees than the degrees of the original arguments (a, b). This process is repeated until one of the arguments is zero—the result is then immediately obtained since $\gcd(0, d) = d$. The algorithm must terminate since the degrees of the remainders are strictly decreasing. Moreover, it is efficient because the number of (long) divisions is at most k where $k = \deg(a)$.

In a variant of the classical Euclidean algorithm, only one step of each long division is performed. That is, if $\deg(b) \geq \deg(a)$ and $j = \deg(b) - \deg(a)$, then one computes $r = b + z^j a$. By Theorem 2.46, $\gcd(a, b) = \gcd(r, a)$. This process is repeated until a zero remainder is encountered. Since $\deg(r) < \deg(b)$, the number of (partial) division steps is at most $2k$ where $k = \max\{\deg(a), \deg(b)\}$.

The Euclidean algorithm can be extended to find binary polynomials g and h satisfying $ag + bh = d$ where $d = \gcd(a, b)$. Algorithm 2.47 maintains the invariants

$$ag_1 + bh_1 = u$$
$$ag_2 + bh_2 = v.$$

The algorithm terminates when $u = 0$, in which case $v = \gcd(a, b)$ and $ag_2 + bh_2 = d$.

Algorithm 2.47 Extended Euclidean algorithm for binary polynomials

INPUT: Nonzero binary polynomials a and b with $\deg(a) \leq \deg(b)$.
OUTPUT: $d = \gcd(a, b)$ and binary polynomials g, h satisfying $ag + bh = d$.

1. $u \leftarrow a$, $v \leftarrow b$.
2. $g_1 \leftarrow 1$, $g_2 \leftarrow 0$, $h_1 \leftarrow 0$, $h_2 \leftarrow 1$.
3. While $u \neq 0$ do
   3.1 $j \leftarrow \deg(u) - \deg(v)$.
   3.2 If $j < 0$ then: $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$, $h_1 \leftrightarrow h_2$, $j \leftarrow -j$.
   3.3 $u \leftarrow u + z^j v$.
   3.4 $g_1 \leftarrow g_1 + z^j g_2$, $h_1 \leftarrow h_1 + z^j h_2$.
4. $d \leftarrow v$, $g \leftarrow g_2$, $h \leftarrow h_2$.
5. Return(d, g, h).

Suppose now that f is an irreducible binary polynomial of degree m and the nonzero polynomial a has degree at most $m - 1$ (hence $\gcd(a, f) = 1$). If Algorithm 2.47 is executed with inputs a and f, the last nonzero u encountered in step 3.3 is $u = 1$. After this occurrence, the polynomials g_1 and h_1, as updated in step 3.4, satisfy $ag_1 + fh_1 = 1$. Hence $ag_1 \equiv 1 \pmod{f}$ and so $a^{-1} = g_1$. Note that h_1 and h_2 are not needed for the determination of g_1. These observations lead to Algorithm 2.48 for inversion in $\mathbb{F}_{2^m}$.

Algorithm 2.48 Inversion in $\mathbb{F}_{2^m}$ using the extended Euclidean algorithm

INPUT: A nonzero binary polynomial a of degree at most $m - 1$.
OUTPUT: $a^{-1} \bmod f$.

1. $u \leftarrow a$, $v \leftarrow f$.
2. $g_1 \leftarrow 1$, $g_2 \leftarrow 0$.
3. While $u \neq 1$ do
   3.1 $j \leftarrow \deg(u) - \deg(v)$.
   3.2 If $j < 0$ then: $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$, $j \leftarrow -j$.
   3.3 $u \leftarrow u + z^j v$.
   3.4 $g_1 \leftarrow g_1 + z^j g_2$.
4. Return(g_1).

Binary inversion algorithm

Algorithm 2.49 is the polynomial analogue of the binary algorithm for inversion in $\mathbb{F}_p$ (Algorithm 2.22). In contrast to Algorithm 2.48 where the bits of u and v are cleared from left to right (high degree terms to low degree terms), the bits of u and v in Algorithm 2.49 are cleared from right to left.

Algorithm 2.49 Binary algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: A nonzero binary polynomial a of degree at most $m-1$.
OUTPUT: $a^{-1} \bmod f$.

1. $u \leftarrow a$, $v \leftarrow f$.
2. $g_1 \leftarrow 1$, $g_2 \leftarrow 0$.
3. While ($u \neq 1$ and $v \neq 1$) do
   3.1 While z divides u do
   $u \leftarrow u/z$.
   If z divides g_1 then $g_1 \leftarrow g_1/z$; else $g_1 \leftarrow (g_1 + f)/z$.
   3.2 While z divides v do
   $v \leftarrow v/z$.
   If z divides g_2 then $g_2 \leftarrow g_2/z$; else $g_2 \leftarrow (g_2 + f)/z$.
   3.3 If $\deg(u) > \deg(v)$ then: $u \leftarrow u + v$, $g_1 \leftarrow g_1 + g_2$;
   Else: $v \leftarrow v + u$, $g_2 \leftarrow g_2 + g_1$.
4. If $u = 1$ then return(g_1); else return(g_2).

The expression involving degree calculations in step 3.3 may be replaced by a simpler comparison on the binary representations of the polynomials. This differs from Algorithm 2.48, where explicit degree calculations are required in step 3.1.

Almost inverse algorithm

The almost inverse algorithm (Algorithm 2.50) is a modification of the binary inversion algorithm (Algorithm 2.49) in which a polynomial g and a positive integer k are first computed satisfying

$$ag \equiv z^k \pmod{f}.$$

A reduction is then applied to obtain

$$a^{-1} = z^{-k} g \bmod f.$$

The invariants maintained are

$$ag_1 + fh_1 = z^k u$$
$$ag_2 + fh_2 = z^k v$$

for some h_1, h_2 that are not explicitly calculated.

Algorithm 2.50 Almost Inverse Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: A nonzero binary polynomial a of degree at most $m-1$.
OUTPUT: $a^{-1} \bmod f$.

1. $u \leftarrow a$, $v \leftarrow f$.
2. $g_1 \leftarrow 1$, $g_2 \leftarrow 0$, $k \leftarrow 0$.
3. While ($u \neq 1$ and $v \neq 1$) do
   3.1 While z divides u do
   $u \leftarrow u/z$, $g_2 \leftarrow z \cdot g_2$, $k \leftarrow k+1$.
   3.2 While z divides v do
   $v \leftarrow v/z$, $g_1 \leftarrow z \cdot g_1$, $k \leftarrow k+1$.
   3.3 If $\deg(u) > \deg(v)$ then: $u \leftarrow u+v$, $g_1 \leftarrow g_1+g_2$.
   Else: $v \leftarrow v+u$, $g_2 \leftarrow g_2+g_1$.
4. If $u = 1$ then $g \leftarrow g_1$; else $g \leftarrow g_2$.
5. Return($z^{-k} g \bmod f$).

The reduction in step 5 can be performed as follows. Let $l = \min\{i \geq 1 \mid f_i = 1\}$, where $f(z) = f_m z^m + \cdots + f_1 z + f_0$. Let S be the polynomial formed by the l rightmost bits of g. Then $Sf + g$ is divisible by z^l and $T = (Sf+g)/z^l$ has degree less than m; thus $T = gz^{-l} \bmod f$. This process can be repeated to finally obtain $gz^{-k} \bmod f$. The reduction polynomial f is said to be *suitable* if l is above some threshold (which may depend on the implementation; e.g., $l \geq W$ is desirable with W-bit words), since then less effort is required in the reduction step.

Steps 3.1–3.2 are simpler than those in Algorithm 2.49. In addition, the g_1 and g_2 appearing in these algorithms grow more slowly in almost inverse. Thus one can expect Algorithm 2.50 to outperform Algorithm 2.49 if the reduction polynomial is suitable, and conversely. As with the binary algorithm, the conditional involving degree calculations may be replaced with a simpler comparison.

Division

The binary inversion algorithm (Algorithm 2.49) can be easily modified to perform division $b/a = ba^{-1}$. In cases where the ratio I/M of inversion to multiplication costs is small, this could be especially significant in elliptic curve schemes, since an elliptic curve point operation in affine coordinates (see §3.1.2) could use division rather than an inversion and multiplication.

Division based on the binary algorithm To obtain b/a, Algorithm 2.49 is modified at step 2, replacing $g_1 \leftarrow 1$ with $g_1 \leftarrow b$. The associated invariants are

$$ag_1 + fh_1 = ub$$
$$ag_2 + fh_2 = vb.$$

On termination with $u = 1$, it follows that $g_1 = ba^{-1}$. The division algorithm is expected to have the same running time as the binary algorithm, since g_1 in Algorithm 2.49 goes to full-length in a few iterations at step 3.1 (i.e., the difference in initialization of g_1 does not contribute significantly to the time for division versus inversion).

If the binary algorithm is the inversion method of choice, then affine point operations would benefit from use of division, since the cost of a point double or addition changes from $I + 2M$ to $I + M$. (I and M denote the time to perform an inversion and a multiplication, respectively.) If I/M is small, then this represents a significant improvement. For example, if I/M is 3, then use of a division algorithm variant of Algorithm 2.49 provides a 20% reduction in the time to perform an affine point double or addition. However, if $I/M > 7$, then the savings is less than 12%. Unless I/M is very small, it is likely that schemes are used which reduce the number of inversions required (e.g., halving and projective coordinates), so that point multiplication involves relatively few field inversions, diluting any savings from use of a division algorithm.

Division based on the extended Euclidean algorithm Algorithm 2.48 can be transformed to a division algorithm in a similar fashion. However, the change in the initialization step may have significant impact on implementation of a division algorithm variant. There are two performance issues: tracking of the lengths of variables, and implementing the addition to g_1 at step 3.4.

In Algorithm 2.48, it is relatively easy to track the lengths of u and v efficiently (the lengths shrink), and, moreover, it is also possible to track the lengths of g_1 and g_2. However, the change in initialization for division means that g_1 goes to full-length immediately, and optimizations based on shorter lengths disappear.

The second performance issue concerns the addition to g_1 at step 3.4. An implementation may assume that ordinary polynomial addition with no reduction may be performed; that is, the degrees of g_1 and g_2 never exceed $m - 1$. In adapting for division, step 3.4 may be less-efficiently implemented, since g_1 is full-length on initialization.

Division based on the almost inverse algorithm Although Algorithm 2.50 is similar to the binary algorithm, the ability to efficiently track the lengths of g_1 and g_2 (in addition to the lengths of u and v) may be an implementation advantage of Algorithm 2.50 over Algorithm 2.49 (provided that the reduction polynomial f is suitable). As with Algorithm 2.48, this advantage is lost in a division algorithm variant.

It should be noted that efficient tracking of the lengths of g_1 and g_2 (in addition to the lengths of u and v) in Algorithm 2.50 may involve significant code expansion (perhaps t^2 fragments rather than the t fragments in the binary algorithm). If the expansion cannot be tolerated (because of application constraints or platform characteristics), then almost inverse may not be preferable to the other inversion algorithms (even if the reduction polynomial is suitable).

2.4 Optimal extension field arithmetic

Preceding sections discussed arithmetic for fields $\mathbb{F}_{p^m}$ in the case that $p = 2$ (binary fields) and $m = 1$ (prime fields). As noted on page 28, the polynomial basis representation in the binary field case can be generalized to all extension fields $\mathbb{F}_{p^m}$, with coefficient arithmetic performed in $\mathbb{F}_p$.

For hardware implementations, binary fields are attractive since the operations involve only shifts and bitwise addition modulo 2. The simplicity is also attractive for software implementations on general-purpose processors; however the field multiplication is essentially a few bits at a time and can be much slower than prime field arithmetic if a hardware integer multiplier is available. On the other hand, the arithmetic in prime fields can be more difficult to implement efficiently, due in part to the propagation of carry bits.

The general idea in optimal extension fields is to select p, m, and the reduction polynomial to more closely match the underlying hardware characteristics. In particular, the value of p may be selected to fit in a single word, simplifying the handling of carry (since coefficients are single-word).

Definition 2.51 An *optimal extension field* (OEF) is a finite field $\mathbb{F}_{p^m}$ such that:

1. $p = 2^n - c$ for some integers n and c with $\log_2 |c| \leq n/2$; and
2. an irreducible polynomial $f(z) = z^m - \omega$ in $\mathbb{F}_p[z]$ exists.

If $c \in \{\pm 1\}$, then the OEF is said to be of *Type I* (p is a Mersenne prime if $c = 1$); if $\omega = 2$, the OEF is said to be of *Type II*.

Type I OEFs have especially simple arithmetic in the subfield $\mathbb{F}_p$, while Type II OEFs allow simplifications in the $\mathbb{F}_{p^m}$ extension field arithmetic. Examples of OEFs are given in Table 2.1.

p	f	parameters	Type
$2^7 + 3$	$z^{13} - 5$	$n = 7,\ c = -3,\ m = 13,\ \omega = 5$	—
$2^{13} - 1$	$z^{13} - 2$	$n = 13,\ c = 1,\ m = 13,\ \omega = 2$	I, II
$2^{31} - 19$	$z^6 - 2$	$n = 31,\ c = 19,\ m = 6,\ \omega = 2$	II
$2^{31} - 1$	$z^6 - 7$	$n = 31,\ c = 1,\ m = 6,\ \omega = 7$	I
$2^{32} - 5$	$z^5 - 2$	$n = 32,\ c = 5,\ m = 5,\ \omega = 2$	II
$2^{57} - 13$	$z^3 - 2$	$n = 57,\ c = 13,\ m = 3,\ \omega = 2$	II
$2^{61} - 1$	$z^3 - 37$	$n = 61,\ c = 1,\ m = 3,\ \omega = 37$	I
$2^{89} - 1$	$z^2 - 3$	$n = 89,\ c = 1,\ m = 2,\ \omega = 3$	I

Table 2.1. OEF example parameters. Here, $p = 2^n - c$ is prime, and $f(z) = z^m - \omega \in \mathbb{F}_p[z]$ is irreducible over $\mathbb{F}_p$. The field is $\mathbb{F}_{p^m} = \mathbb{F}_p[z]/(f)$ of order approximately 2^{mn}.

The following results can be used to determine if a given polynomial $f(z) = z^m - \omega$ is irreducible in $\mathbb{F}_p[z]$.

Theorem 2.52 Let $m \geq 2$ be an integer and $\omega \in \mathbb{F}_p^*$. Then the binomial $f(z) = z^m - \omega$ is irreducible in $\mathbb{F}_p[z]$ if and only if the following two conditions are satisfied:

(i) each prime factor of m divides the order e of ω in $\mathbb{F}_p^*$, but not $(p-1)/e$;

(ii) $p \equiv 1 \pmod 4$ if $m \equiv 0 \pmod 4$.

If the order of ω as an element of $\mathbb{F}_p^*$ is $p-1$, then ω is said to be *primitive*. It is easily verified that conditions (i) and (ii) of Theorem 2.52 are satisfied if ω is primitive and $m|(p-1)$.

Corollary 2.53 If ω is a primitive element of $\mathbb{F}_p^*$ and $m|(p-1)$, then $z^m - \omega$ is irreducible in $\mathbb{F}_p[z]$.

Elements of $\mathbb{F}_{p^m}$ are polynomials

$$a(z) = a_{m-1}z^{m-1} + \cdots + a_2 z^2 + a_1 z + a_0$$

where the coefficients a_i are elements of $\mathbb{F}_p$. We next present algorithms for performing arithmetic operations in OEFs. Selected timings for field operations appear in §5.1.5.

2.4.1 Addition and subtraction

If $a(z) = \sum_{i=0}^{m-1} a_i z^i$ and $b(z) = \sum_{i=0}^{m-1} b_i z^i$ are elements of $\mathbb{F}_{p^m}$, then

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

where $c_i = (a_i + b_i) \bmod p$; that is, p is subtracted whenever $a_i + b_i \geq p$. Subtraction of elements of $\mathbb{F}_{p^m}$ is done similarly.

2.4.2 Multiplication and reduction

Multiplication of elements $a, b \in \mathbb{F}_{p^m}$ can be done by ordinary polynomial multiplication in $\mathbb{Z}[z]$ (i.e., multiplication of polynomials having integer coefficients), along with coefficient reductions in $\mathbb{F}_p$ and a reduction by the polynomial f. This multiplication takes the form

$$\begin{aligned} c(z) = a(z)b(z) &= \left(\sum_{i=0}^{m-1} a_i z^i\right)\left(\sum_{j=0}^{m-1} b_j z^j\right) \\ &\equiv \sum_{k=0}^{2m-2} c_k z^k \equiv c_{m-1}z^{m-1} + \sum_{k=0}^{m-2}(c_k + \omega c_{k+m})z^k \pmod{f(z)} \end{aligned}$$

where

$$c_k = \sum_{i+j=k} a_i b_j \bmod p.$$

Karatsuba-Ofman techniques may be applied to reduce the number of $\mathbb{F}_p$ multiplications. For example,

$$\begin{aligned} a(z)b(z) &= (A_1 z^l + A_0)(B_1 z^l + B_0) \\ &= A_1 B_1 z^{2l} + [(A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0] z^l + A_0 B_0 \end{aligned}$$

where $l = \lceil m/2 \rceil$ and the coefficients A_0, A_1, B_0, B_1 are polynomials in $\mathbb{F}_p[z]$ of degree less than l. The process may be repeated, although for small values of m it may be advantageous to consider splits other than binary. The analogous case for prime fields was discussed in §2.2.2.

Reduction in $\mathbb{F}_p$

The most straightforward implementation performs reductions in $\mathbb{F}_p$ for every addition and multiplication encountered during the calculation of each c_k. The restriction $\log_2 |c| \leq n/2$ means that reduction in the subfield $\mathbb{F}_p$ requires only a few simple operations. Algorithm 2.54 performs reduction of base-B numbers, using only shifts, additions, and single-precision multiplications.

Algorithm 2.54 Reduction modulo $M = B^n - c$

INPUT: A base B, positive integer x, and modulus $M = B^n - c$ where c is an l-digit base-B positive integer for some $l < n$.
OUTPUT: $x \bmod M$.

1. $q_0 \leftarrow \lfloor x/B^n \rfloor$, $r_0 \leftarrow x - q_0 B^n$. $\{x = q_0 B^n + r_0 \text{ with } r_0 < B^n\}$
2. $r \leftarrow r_0$, $i \leftarrow 0$.
3. While $q_i > 0$ do
   3.1 $q_{i+1} \leftarrow \lfloor q_i c / B^n \rfloor$. $\{q_i c = q_{i+1} B^n + r_{i+1} \text{ with } r_{i+1} < B^n\}$
   3.2 $r_{i+1} \leftarrow q_i c - q_{i+1} B^n$.
   3.3 $i \leftarrow i + 1$, $r \leftarrow r + r_i$.
4. While $r \geq M$ do: $r \leftarrow r - M$.
5. Return(r).

Note 2.55 (*implementation details for Algorithm 2.54*)

(i) If $l \leq n/2$ and x has at most $2n$ base-B digits, Algorithm 2.54 executes step 3.1 at most twice (i.e., there are at most two multiplications by c).

(ii) As an alternative, the quotient and remainder may be folded into x at each stage. Steps 1–4 are replaced with the following.

1. While $x \geq B^n$
   1.1 Write $x = vB^n + u$ with $u < B^n$.
   1.2 $x \leftarrow cv + u$.
2. If $x \geq M$ then $x \leftarrow x - M$.

(iii) Algorithm 2.54 can be modified to handle the case $M = B^n + c$ for some positive integer $c < B^{n-1}$: in step 3.3, replace $r \leftarrow r + r_i$ with $r \leftarrow r + (-1)^i r_i$, and modify step 4 to also process the case $r < 0$.

For OEFs, Algorithm 2.54 with $B = 2$ may be applied, requiring at most two multiplications by c in the case that $x < 2^{2n}$. When $c = 1$ (a type I OEF) and $x \leq (p-1)^2$, the reduction is given by:

$$\text{write } x = 2^n v + u; \quad x \leftarrow v + u; \quad \text{if } x \geq p \text{ then } x \leftarrow x - p.$$

Type I OEFs are attractive in the sense that $\mathbb{F}_p$ multiplication (with reduction) can be done with a single multiplication and a few other operations. However, the reductions modulo p are likely to contribute a significant amount to the cost of multiplication in $\mathbb{F}_{p^m}$, and it may be more efficient to employ a direct multiply-and-accumulate strategy to decrease the number of reductions.

Accumulation and reduction

The number of $\mathbb{F}_p$ reductions performed in finding the product $c(z) = a(z)b(z)$ in $\mathbb{F}_{p^m}$ can be decreased by accumulation strategies on the coefficients of $c(z)$. Since $f(z) = z^m - \omega$, the product can be written

$$\begin{aligned} c(z) = a(z)b(z) &\equiv \sum_{k=0}^{2m-2} c_k z^k \equiv \sum_{k=0}^{m-1} c_k z^k + \omega \sum_{k=m}^{2m-2} c_k z^{k-m} \\ &\equiv \sum_{k=0}^{m-1} \Bigg(\underbrace{\sum_{i=0}^{k} a_i b_{k-i} + \omega \sum_{i=k+1}^{m-1} a_i b_{k+m-i}}_{c'_k} \Bigg) z^k \pmod{f(z)}. \end{aligned}$$

If the coefficient c'_k is calculated as an expression in $\mathbb{Z}$ (i.e., as an integer without reduction modulo p), then $c'_k \bmod p$ may be performed with a single reduction (rather than m reductions). The penalty incurred is the multiple-word operations (additions and multiplication by ω) required in accumulating the terms of c'_k.

In comparison with the straightforward reduce-on-every-operation strategy, it should be noted that complete reduction on each $\mathbb{F}_p$ operation may not be necessary; for example, it may suffice to reduce the result to a value which fits in a single word. However, frequent reduction (to a single word or value less than 2^n) is likely to be expensive, especially if a "carry" or comparison must be processed.

Depending on the value of p, the multiply-and-accumulate strategy employs two or three registers for the accumulation (under the assumption that p fits in a register). The arithmetic resembles that commonly used in prime-field implementations, and multiplication cost in $\mathbb{F}_{p^m}$ is expected to be comparable to that in a prime field $\mathbb{F}_q$ where $q \approx p^m$ and which admits fast reduction (e.g., the NIST-recommended primes in §2.2.6).

For the reduction $c'_k \bmod p$, note that

$$c'_k \le (p-1)^2 + \omega(m-1)(p-1)^2 = (p-1)^2(1+\omega(m-1)).$$

If $p = 2^n - c$ is such that

$$\log_2(1+\omega(m-1)) + 2\log_2|c| \le n, \tag{2.4}$$

then reduction can be done with at most two multiplications by c. As an example, if $p = 2^{28} - 165$ and $f(z) = z^6 - 2$, then

$$\log_2(1+\omega(m-1)) + 2\log_2|c| = \log_2 11 + 2\log_2 165 < n = 28$$

and condition (2.4) is satisfied.

If accumulation is in a series of registers each of size W bits, then selecting $p = 2^n - c$ with $n < W$ allows several terms to be accumulated in two registers (rather than spilling into a third register or requiring a partial reduction). The example with $p = 2^{28} - 165$ is attractive in this sense if $W = 32$. However, this strategy competes with optimal use of the integer multiply, and hence may not be effective if it requires use of a larger m to obtain a field of sufficient size.

Example 2.56 (*accumulation strategies*) Consider the OEF defined by $p = 2^{31} - 1$ and $f(z) = z^6 - 7$, on a machine with wordsize $W = 32$. Since this is a Type I OEF, subfield reduction is especially simple, and a combination of partial reduction with accumulation may be effective in finding $c'_k \bmod p$. Although reduction into a single register after each operation may be prohibitively expensive, an accumulation into two registers (with some partial reductions) or into three registers can be employed.

Suppose the accumulator consists of two registers. A partial reduction may be performed on each term of the form $a_i b_j$ by writing $a_i b_j = 2^{32} v + u$ and then $2v + u$ is added to the accumulator. Similarly, the accumulator itself could be partially reduced after the addition of a product $a_i b_j$.

If the accumulator is three words, then the partial reductions are unnecessary, and a portion of the accumulation involves only two registers. On the other hand, the multiplication by $\omega = 7$ and the final reduction are slightly more complicated than in the two-register approach.

The multiply-and-accumulate strategies also apply to field squaring in $\mathbb{F}_{p^m}$. Squaring requires a total of $m + \binom{m}{2} = m(m+1)/2$ integer multiplications (and possibly $m-1$ multiplications by ω). The cost of the $\mathbb{F}_p$ reductions depends on the method; in particular, if only a single reduction is used in finding c'_k, then the number of reductions is the same as for general multiplication.

2.4.3 Inversion

Inversion of $a \in \mathbb{F}_{p^m}$, $a \neq 0$, finds a polynomial $a^{-1} \in \mathbb{F}_{p^m}$ such that $aa^{-1} \equiv 1 \pmod{f}$. Variants of the Euclidean Algorithm have been proposed for use with OEFs. However, the action of the Frobenius map along with the special form of f can be used to obtain an inversion method that is among the fastest. The method is also relatively simple to implement efficiently once field multiplication is written, since only a few multiplications are needed to reduce inversion in $\mathbb{F}_{p^m}$ to inversion in the subfield $\mathbb{F}_p$.

Algorithm 2.59 computes

$$a^{-1} = (a^r)^{-1} a^{r-1} \bmod f \tag{2.5}$$

where

$$r = \frac{p^m - 1}{p - 1} = p^{m-1} + \cdots + p^2 + p + 1.$$

Since $(a^r)^{p-1} \equiv 1 \pmod{p^m}$, it follows that $a^r \in \mathbb{F}_p$. Hence a suitable algorithm may be applied for inversion in $\mathbb{F}_p$ in order to compute the term $(a^r)^{-1}$ in (2.5).

Efficient calculation of $a^{r-1} = a^{p^{m-1}+\cdots+p}$ in (2.5) is performed by using properties of the *Frobenius map* $\varphi : \mathbb{F}_{p^m} \to \mathbb{F}_{p^m}$ defined by $\varphi(a) = a^p$. Elements of $\mathbb{F}_p$ are fixed by this map. Hence, if $a = a_{m-1}z^{m-1} + \cdots + a_2 z^2 + a_1 z + a_0$, then

$$\varphi^i : a \mapsto a_{m-1} z^{(m-1)p^i} + \cdots + a_1 z^{p^i} + a_0 \bmod f.$$

To reduce the powers of z modulo f, write a given nonnegative integer e as $e = qm + r$, where $q = \lfloor e/m \rfloor$ and $r = e \bmod m$. Since $f(z) = z^m - \omega$, it follows that

$$z^e = z^{qm+r} \equiv \omega^q z^r \pmod{f(z)}.$$

Notice that $\varphi^i(a)$ is somewhat simpler to evaluate if $p \equiv 1 \pmod{m}$. By Theorem 2.52, every prime factor of m divides $p - 1$. Necessarily, if m is square free, the condition $p \equiv 1 \pmod{m}$ holds. The results are collected in the following theorem.

Theorem 2.57 (*action of Frobenius map iterates*) Given an OEF with $p = 2^n - c$ and $f(z) = z^m - \omega$, let the Frobenius map on $\mathbb{F}_{p^m}$ be given by $\varphi : a \mapsto a^p \bmod f$.

(i) The ith iterate of φ is the map

$$\varphi^i : a \mapsto \sum_{j=0}^{m-1} a_j \omega^{\lfloor jp^i/m \rfloor} z^{jp^i \bmod m}.$$

(ii) If m is square-free, then $p \equiv 1 \pmod{m}$ and hence $jp^i \bmod m = j$ for all $0 \leq j \leq m - 1$.

The values $z^e \equiv \omega^{\lfloor e/m \rfloor} z^{e \bmod m} \pmod{f(z)}$ may be precomputed for $e = jp^i$ of interest, in which case $\varphi^i(a)$ may be evaluated with only $m-1$ multiplications in $\mathbb{F}_p$. Use of an addition chain then efficiently finds a^{r-1} in equation (2.5) using a few field multiplications and applications of φ^i.

Example 2.58 (*calculating* a^{r-1}) The OEF defined by $p = 2^{31}-1$ and $f(z) = z^6 - 7$ has $r - 1 = p^5 + p^4 + \cdots + p$. We may calculate a^{r-1} using the sequence indicated in Table 2.2 (an addition-chain-like method) for $m = 6$. Evaluation of φ and φ^2 uses the precomputed values in Table 2.3 obtained from Theorem 2.57.

$m = 3$	$m = 5$	$m = 6$
$T \leftarrow a^p$	$T_1 \leftarrow a^p$	$T_1 \leftarrow a^p$
$T \leftarrow Ta = a^{p+1}$	$T_1 \leftarrow T_1 a = a^{p+1}$	$T_2 \leftarrow T_1 a = a^{p+1}$
$a^{r-1} \leftarrow T^p = a^{p^2+p}$	$T_2 \leftarrow T_1^{p^2} = a^{p^3+p^2}$	$T_3 \leftarrow T_2^{p^2} = a^{p^3+p^2}$
	$T_1 \leftarrow T_1 T_2 = a^{p^3+p^2+p+1}$	$T_2 \leftarrow T_3 T_2 = a^{p^3+p^2+p+1}$
	$a^{r-1} \leftarrow T_1^p$	$T_2 \leftarrow T_2^{p^2} = a^{p^5+p^4+p^3+p^2}$
		$a^{r-1} \leftarrow T_2 T_1$
Cost: $1M + 2\varphi$	Cost: $2M + 3\varphi$	Cost: $3M + 3\varphi$

Table 2.2. Computation of a^{r-1} *for* $r = \frac{p^m-1}{p-1}$, $m \in \{3, 5, 6\}$. *The final row indicates the cost in* $\mathbb{F}_{p^m}$ *multiplications (*M*) and applications of an iterate of the Frobenius map (*φ*).*

$z^{jp} \equiv \omega^{\lfloor jp/m \rfloor} z^j \pmod{f}$	$z^{jp^2} \equiv \omega^{\lfloor jp^2/m \rfloor} z^j \pmod{f}$
$z^p \equiv 1513477736z$	$z^{p^2} \equiv 1513477735z$
$z^{2p} \equiv 1513477735z^2$	$z^{2p^2} \equiv 634005911z^2$
$z^{3p} \equiv 2147483646z^3 \equiv -1z^3$	$z^{3p^2} \equiv 1z^3$
$z^{4p} \equiv 634005911z^4$	$z^{4p^2} \equiv 1513477735z^4$
$z^{5p} \equiv 634005912z^5$	$z^{5p^2} \equiv 634005911z^5$

Table 2.3. Precomputation for evaluating φ^i, $i \in \{1, 2\}$, *in the case* $p = 2^{31}-1$ *and* $f(z) = z^6 - 7$ *(cf. Example 2.58). If* $a = a_5 z^5 + \cdots + a_1 z + a_0 \in \mathbb{F}_{p^6}$, *then* $\varphi^i(a) = a^{p^i} \equiv \sum_{j=0}^{5} a_j \omega^{\lfloor jp^i/m \rfloor} z^j \pmod{f}$.

In general, if $w(x)$ is the Hamming weight of the integer x, then a^{r-1} can be calculated with

$$t_1(m) = \lfloor \log_2(m-1) \rfloor + w(m-1) - 1$$

multiplications in $\mathbb{F}_{p^m}$, and

$$t_2(m) = \begin{cases} t_1(m)+1, & m \text{ odd}, \\ j = \lfloor \log_2(m-1) \rfloor + 1, & m = 2^j \text{ for some } j, \\ \lfloor \log_2(m-1) \rfloor + w(m) - 1, & \text{otherwise}, \end{cases}$$

applications of Frobenius map iterates. Since $t_2(m) \leq t_1(m)+1$, the time for calculating a^{r-1} with $m > 2$ is dominated by the multiplications in $\mathbb{F}_{p^m}$ (each of which is much more expensive than the $m-1$ multiplications in $\mathbb{F}_p$ needed for evaluation of φ^i).

Algorithm 2.59 OEF inversion

INPUT: $a \in \mathbb{F}_{p^m}$, $a \neq 0$.
OUTPUT: The element $a^{-1} \in \mathbb{F}_{p^m}$ such that $aa^{-1} \equiv 1 \pmod{f}$.

1. Use an addition-chain approach to find a^{r-1}, where $r = (p^m - 1)/(p-1)$.
2. $c \leftarrow a^r = a^{r-1}a \in \mathbb{F}_p$.
3. Obtain c^{-1} such that $cc^{-1} \equiv 1 \pmod{p}$ via an inversion algorithm in $\mathbb{F}_p$.
4. Return$(c^{-1}a^{r-1})$.

Note 2.60 (*implementation details for Algorithm 2.59*)

(i) The element c in step 2 of Algorithm 2.59 belongs to $\mathbb{F}_p$. Hence, only arithmetic contributing to the constant term of $a^{r-1}a$ need be performed (requiring m multiplications of elements in $\mathbb{F}_p$ and a multiplication by ω).

(ii) Since $c^{-1} \in \mathbb{F}_p$, the multiplication in step 4 requires only m $\mathbb{F}_p$-multiplications.

(iii) The running time is dominated by the $t_1(m)$ multiplications in $\mathbb{F}_{p^m}$ in finding a^{r-1}, and the cost of the subfield inversion in step 3.

The ratio I/M of field inversion cost to multiplication cost is of fundamental interest. When $m = 6$, Algorithm 2.59 will require significantly more time than the $t_1(6) = 3$ multiplications involved in finding a^{r-1}, since the time for subfield inversion (step 3) will be substantial. However, on general-purpose processors, the ratio is expected to be much smaller than the corresponding ratio in a prime field $\mathbb{F}_q$ where $q \approx p^m$.

2.5 Notes and further references

§2.1
For an introduction to the theory of finite fields, see the books of Koblitz [254] and McEliece [311]. A more comprehensive treatment is given by Lidl and Niederreiter [292].

§2.2
Menezes, van Oorshot, and Vanstone [319] concisely cover algorithms for ordinary and modular integer arithmetic of practical interest in cryptography. Knuth [249] is a standard reference. Koç [258] describes several (modular) multiplication methods, including classical and Karatsuba-Ofman, a method which interleaves multiplication with reduction, and Montgomery multiplication.

The decision to base multiplication on operand scanning (Algorithm 2.9) or product scanning (Algorithm 2.10) is platform dependent. Generally speaking, Algorithm 2.9 has more memory accesses, while Algorithm 2.10 has more complex control code unless loops are unrolled. Comba [101] compares the methods in detail for 16-bit Intel 80286 processors, and the unrolled product-scanning versions were apparently the inspiration for the "comba" routines in OpenSSL.

Scott [416] discusses multiplication methods on three 32-bit Intel IA-32 processors (the 80486, Pentium, and Pentium Pro), and provides experimental results for modular exponentiation with multiplication based on operand scanning, product scanning (Comba's method), Karatsuba-Ofman with product scanning, and floating-point hardware. Multiplication with features introduced on newer IA-32 processors is discussed in §5.1.3. On the Motorola digital signal processor 56000, Dussé and Kaliski [127] note that extraction of U in the inner loop of Algorithm 2.9 is relatively expensive. The processor has a 56-bit accumulator but only signed multiplication of 24-bit quantities, and the product scanning approach in Montgomery multiplication is reportedly significantly faster.

The multiplication method of Karatsuba-Ofman is due to Karatsuba and Ofman [239]. For integers of relatively small size, the savings in multiplications is often insufficient in Karatsuba-Ofman variants to make the methods competitive with optimized versions of classical algorithms. Knuth [249] and Koç [258] cover Karatsuba-Ofman in more detail.

Barrett reduction (Algorithm 2.14) is due to Barrett [29]. Bosselaers, Govaerts, and Vandewalle [66] provide descriptions and comparative results for classical reduction and the reduction methods of Barrett and Montgomery. If the transformations and precomputation are excluded, their results indicate that the methods are fairly similar in cost, with Montgomery reduction fastest and classical reduction likely to be slightly slower than Barrett reduction. These operation count comparisons are supported by implementation results on an Intel 80386 in portable C. De Win, Mister, Preneel and Wiener [111] report that the difference between Montgomery and Barrett reduction was negligible in their implementation on an Intel Pentium Pro of field arithmetic in $\mathbb{F}_p$ for a 192-bit prime p.

Montgomery reduction is due to Montgomery [330]. Koç, Acar, and Kaliski [260] analyze five Montgomery multiplication algorithms. The methods were identified as having a separate reduction phase or reduction integrated with multiplication, and

according to the general form of the multiplication as operand-scanning or product-scanning. Among the algorithms tested, they conclude that a "coarsely integrated operand scanning" method (where a reduction step follows a multiplication step at each index of an outer loop through one of the operands) is simplest and probably best for general-purpose processors. Koç and Acar [259] extend Montgomery multiplication to binary fields.

The binary gcd algorithm (Algorithm 2.21) is due to Stein [451], and is analyzed by Knuth [249]. Bach and Shallit [23] provide a comprehensive analysis of several gcd algorithms. The binary algorithm for inversion (Algorithm 2.22) is adapted from the corresponding extended gcd algorithm.

Lehmer [278] proposed a variant of the classical Euclidean algorithm which replaces most of the expensive multiple-precision divisions by single-precision operations. The algorithm is examined in detail by Knuth [249], and a slight modification is analyzed by Sorenson [450]. Durand [126] provides concise coverage of inversion algorithms adapted from the extended versions of the Euclidean, binary gcd, and Lehmer algorithms, along with timings for RSA and elliptic curve point multiplication on 32-bit RISC processors (for smartcards) from SGS-Thomson. On these processors, Lehmer's method showed significant advantages, and in fact produced point multiplication times faster than was obtained with projective coordinates.

Algorithm 2.23 for the partial Montgomery inverse is due to Kaliski [234]. De Win, Mister, Preneel and Wiener [111] report that an inversion method based on this algorithm was superior to variations of the extended Euclidean algorithm (Algorithm 2.19) in their tests on an Intel Pentium Pro, although details are not provided. The generalization in Algorithm 2.25 is due to Savas and Koç [403]; a similar algorithm is provided for finding the usual inverse.

Simultaneous inversion (Algorithm 2.26) is attributed to Montgomery [331], where the technique was suggested for accelerating the elliptic curve method (ECM) of factoring. Cohen [99, Algorithm 10.3.4] gives an extended version of Algorithm 2.26, presented in the context of ECM.

The NIST primes (§2.2.6) are given in the Federal Information Processing Standards (FIPS) publication 186-2 [140] on the Digital Signature Standard, as part of the recommended elliptic curves for US Government use. Solinas [445] discusses generalizations of Mersenne numbers $2^k - 1$ that permit fast reduction (without division); the NIST primes are special cases.

§2.3

Algorithms 2.35 and 2.36 for polynomial multiplication are due to López and Dahab [301]. Their work expands on "comb" exponentiation methods of Lim and Lee [295]. Operation count comparisons and implementation results (on Intel family and Sun UltraSPARC processors) suggest that Algorithm 2.36 will be significantly faster than

Algorithm 2.34 at relatively modest storage requirements. The multiple-table variants in Note 2.37 are essentially described by López and Dahab [301, Remark 2].

The OpenSSL contribution by Sun Microsystems Laboratories mentioned in Note 2.38 is authored by Sheueling Chang Shantz and Douglas Stebila. Our notes are based in part on OpenSSL-0.9.8 snapshots. A significant enchancement is discussed by Weimerskirch, Stebila, and Chang Shantz [478]. Appendix C has a few notes on the OpenSSL library.

The NIST reduction polynomials (§2.3.5) are given in the Federal Information Processing Standards (FIPS) publication 186-2 [140] on the Digital Signature Standard, as part of the recommended elliptic curves for US Government use.

The binary algorithm for inversion (Algorithm 2.49) is the polynomial analogue of Algorithm 2.22. The almost inverse algorithm (Algorithm 2.50) is due to Schroeppel, Orman, O'Malley, and Spatscheck [415]; a similar algorithm (Algorithm 2.23) in the context of Montgomery inversion was described by Kaliski [234].

Algorithms for field division were described by Goodman and Chandrakasan [177], Chang Shantz [90], Durand [126], and Schroeppel [412]. Inversion and division algorithm implementations are especially sensitive to compiler differences and processor characteristics, and rough operation count analysis can be misleading. Fong, Hankerson, López and Menezes [144] discuss inversion and division algorithm considerations and provide comparative timings for selected compilers on the Intel Pentium III and Sun UltraSPARC.

In a *normal basis* representation, elements of $\mathbb{F}_{2^m}$ are expressed in terms of a basis of the form $\{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$. One advantage of normal bases is that squaring of a field element is a simple rotation of its vector representation. Mullin, Onyszchuk, Vanstone and Wilson [337] introduced the concept of an *optimal normal basis* in order to reduce the hardware complexity of multiplying field elements in $\mathbb{F}_{2^m}$ whose elements are represented using a normal basis. Hardware implementations of the arithmetic in $\mathbb{F}_{2^m}$ using optimal normal bases are described by Agnew, Mullin, Onyszchuk and Vanstone [6] and Sunar and Koç [456].

Normal bases of low complexity, also known as *Gaussian normal bases*, were further studied by Ash, Blake and Vanstone [19]. Gaussian normal bases are explicitly described in the ANSI X9.62 standard [14] for the ECDSA. Experience has shown that optimal normal bases do not have any significant advantages over polynomial bases for hardware implementation. Moreover, field multiplication in software for normal basis representations is very slow in comparison to multiplication with a polynomial basis; see Reyhani-Masoleh and Hasan [390] and Ning and Yin [348].

§2.4
Optimal extension fields were introduced by Bailey and Paar [25, 26]. Theorem 2.52 is from Lidl and Niederreiter [292, Theorem 3.75]. Theorem 2.57 corrects [26, Corollary

2]. The OEF construction algorithm of [26] has a minor flaw in the test for irreducibility, leading to a few incorrect entries in their table of Type II OEFs (e.g, $z^{25} - 2$ is not irreducible when $p = 2^8 - 5$). The inversion method of §2.4.3 given by Bailey and Paar is based on Itoh and Tsujii [217]; see also [183].

Lim and Hwang [293] give thorough coverage to various optimization strategies and provide useful benchmark timings on Intel and DEC processors. Their operation count analysis favours a Euclidean algorithm variant over Algorithm 2.59 for inversion. However, rough operation counts at this level often fail to capture processor or compiler characteristics adequately, and in subsequent work [294] they note that Algorithm 2.59 appears to be significantly faster in implementation on Intel Pentium II and DEC Alpha processors. Chung, Sim, and Lee [97] note that the count for the number of required Frobenius-map applications in inversion given in [26] is not necessarily minimal. A revised formula is given, along with inversion algorithm comparisons and implementation results for a low-power Samsung CalmRISC 8-bit processor with a math coprocessor.

http://www.springer.com/978-0-387-95273-4

Guide to Elliptic Curve Cryptography
Hankerson, D.; Menezes, A.J.; Vanstone, S.
2004, XX, 312 p., Hardcover
ISBN: 978-0-387-95273-4