IEEE P1363 / D9 (Draft Version 9). Standard Specifications for Public Key Cryptography

Annex A (informative). Number-Theoretic Background.

Comments and suggestions are welcome. Please contact the editor, Yiqun Lisa Yin, at yiqun@rsa.com.

ANNEX A (informative)
Number-Theoretic Background

A.1 Integer and Modular Arithmetic: Overview

A.1.1. Modular arithmetic

Modular reduction.

Modular arithmetic is based on a fixed integer $m > 1$ called the *modulus*. The fundamental operation is *reduction modulo m*. To reduce an integer a modulo m, one divides a by m and takes the remainder r. This operation is written

$$r := a \bmod m.$$

The remainder must satisfy $0 \le r < m$.

Examples:

$$11 \bmod 8 = 3$$
$$7 \bmod 9 = 7$$
$$-2 \bmod 11 = 9$$
$$12 \bmod 12 = 0$$

Congruences.

Two integers a and b are said to be *congruent modulo m* if they have the same result upon reduction modulo m. This relationship is written

$$a \equiv b \pmod m.$$

Two integers are congruent modulo m if and only if their difference is divisible by m.

Example:

$$11 \equiv 19 \pmod 8.$$

If $r = a \bmod m$, then $r \equiv a \pmod m$.

If $a_0 \equiv b_0 \pmod m$ and $a_1 \equiv b_1 \pmod m$, then

$$a_0 + a_1 \equiv b_0 + b_1 \pmod m$$
$$a_0 - a_1 \equiv b_0 - b_1 \pmod m$$
$$a_0 a_1 \equiv b_0 b_1 \pmod m.$$

Integers modulo m.

The *integers modulo m* are the possible results of reduction modulo m. Thus the set of integers modulo m is

$$Z_m = \{0, 1, \ldots, m - 1\}.$$

One performs addition, subtraction, and multiplication on the set Z_m by performing the corresponding integer operation and reducing the result modulo m. For example, in Z_7

$$3 = 6 + 4$$
$$5 = 1 - 3$$
$$6 = 4 \times 5.$$

Modular exponentiation.

If v is a positive integer and g is an integer modulo m, then *modular exponentiation* is the operation of computing $g^v \bmod m$ (also written $\exp(g, v) \bmod m$). Section A.2.1 contains an efficient method for modular exponentiation.

G.C.D.'s and L.C.M.'s.

If m and h are integers, the *greatest common divisor* (or *G.C.D.*) is the largest positive integer d dividing both m and h. If $d = 1$, then m and h are said to be *relatively prime* (or *coprime*). Section A.2.2 contains an efficient method for computing the G.C.D.

The *least common multiple* (or *L.C.M.*) is the smallest positive integer l divisible by both m and h. The G.C.D. and L.C.M. are related by

$$\text{GCD}(h, m) \times \text{LCM}(h, m) = hm$$

(for h and m positive), so that the L.C.M. is easily computed if the G.C.D. is known.

Modular division.

The *multiplicative inverse* of h modulo m is the integer k modulo m such that $hk \equiv 1 \pmod{m}$. The multiplicative inverse of h is commonly written $h^{-1} \pmod{m}$. It exists if h is relatively prime to m and not otherwise.

If g and h are integers modulo m, and h is relatively prime to m, then the *modular quotient* g/h modulo m is the integer $gh^{-1} \bmod m$. If c is the modular quotient, then c satisfies $g \equiv hc \pmod{m}$.

The process of finding the modular quotient is called *modular division*. Section A.2.2 contains an efficient method for modular division.

A.1.2. Prime finite fields

The field GF (p).

In the case in which m equals a prime p, the set Z_p forms a *prime finite field* and is denoted $GF(p)$.

In the finite field $GF(p)$, modular division is possible for any denominator other than 0. The set of nonzero elements of $GF(p)$ is denoted $GF(p)^*$.

Orders.

The *order* of an element c of $GF(p)^*$ is the smallest positive integer v such that $c^v \equiv 1 \pmod{p}$. The order always exists and divides $p - 1$. If k and l are integers, then $c^k \equiv c^l \pmod{p}$ if and only if $k \equiv l \pmod{v}$.

Generators.

If v divides $p - 1$, then there exists an element of $GF(p)^*$ having order v. In particular, there always exists an element g of order $p - 1$ in $GF(p)^*$. Such an element is called a *generator* for $GF(p)^*$ because every element of $GF(p)^*$ is some power of g. In number-theoretic language, g is also called a *primitive root* for p.

Exponentiation and discrete logarithms.

Suppose that the element g of $GF(p)^*$ has order v. Then an element h of $GF(p)^*$ satisfies

$$h \equiv g^l \pmod{p}$$

for some l if and only if $h^v \equiv 1 \pmod{p}$. The exponent l is called the *discrete logarithm* of h (with respect to the base g). The discrete logarithm is an integer modulo v.

DL-based cryptography

Suppose that g is an order-r element of $GF(p)^*$ where r is prime. Then a key pair can be defined as follows.

— The private key s is an integer modulo r.
— The corresponding public key w is an element of $GF(p)^*$ defined by

$$w := g^s \pmod{p}.$$

It is necessary to compute a discrete logarithm in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the discrete logarithm problem. Thus it is an example of *DL-based cryptography*. The difficulty of the discrete logarithm problem is discussed in Annex D.4.1.

A.1.3 Composite Moduli

RSA moduli.

An *RSA modulus* is the product n of two odd (distinct) primes p and q. It is assumed that p and q are large enough that factoring n is computationally infeasible (see below).

A *unit* modulo n is a positive integer less than n and relatively prime to n (i.e. divisible by neither p nor q). The set of all units modulo n is denoted by U_n. The modular product and modular quotient of units are also units. If a is a unit and k an integer, then $a^k \bmod n$ is also a unit.

If a positive integer less than n is selected without knowledge of p or q, it is virtually certain to be a unit. (Indeed, generating a nonunit modulo n is as difficult as factoring n.)

Exponentiation.

The *universal exponent* of an RSA modulus $n = pq$ is defined to be

$$\lambda(n) := \mathrm{LCM}(p - 1, q - 1).$$

If c and d are congruent modulo $\lambda(n)$, then

$$a^c \equiv a^d \pmod{n}$$

for every unit a. It follows that, if

$$de \equiv 1 \pmod{\lambda(n)},$$

then

$$a^{de} \equiv a \pmod{n}$$

for every unit a.

IF-based cryptography.

Given an RSA modulus n, a pair of integers d, e satisfying $de \equiv 1 \pmod{\lambda(n)}$ can be taken as the components of a key pair. Traditionally, e denotes the public key and d the private key. Given n and e, it is known that finding d is as difficult as factoring n. This in turn is believed to be necessary for computing a unit a given $a^e \bmod n$. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the integer factorization problem. Thus it is an example of *IF-based cryptography.* The difficulty of the integer factorization problem is discussed in Annex D.4.3.

A.1.4 Modular Square Roots

The Legendre symbol.

If $p > 2$ is prime, and a is any integer, then the *Legendre symbol* $\left(\frac{a}{p}\right)$ is defined as follows. If p divides a, then $\left(\frac{a}{p}\right) = 0$. If p does not divide a, then $\left(\frac{a}{p}\right)$ equals 1 if a is a square modulo p and –1 otherwise. (Despite the similarity in notation, a Legendre symbol should not be confused with a rational fraction; the distinction must be made from the context.)

The Jacobi symbol.

The *Jacobi symbol* $\left(\frac{a}{n}\right)$ is a generalization of the Legendre symbol. If $n > 1$ is odd with prime factorization

$$n = \prod_{i=1}^{t} p_i^{e_i} ,$$

and a is any integer, then the Jacobi symbol is defined to be

$$\left(\frac{a}{n}\right) := \prod_{i=1}^{t} \left(\frac{a}{p_i}\right)^{e_i} ,$$

where the symbols $\left(\frac{a}{p_i}\right)$ are Legendre symbols. (Despite the similarity in notation, a Jacobi symbol should not be confused with a rational fraction; the distinction must be made from the context.)

The values of the Jacobi symbol are ±1 if a and n are relatively prime and 0 otherwise. The values 1 and –1 are achieved equally often (unless n is a square, in which case the value –1 does not occur at all).

Algorithms for computing Legendre and Jacobi symbols are given in A.2.3.

Square roots modulo a prime.

Let p be an odd prime, and let g be an integer with $0 \le g < p$. A *square root modulo p* of g is an integer z with $0 \le z < p$ and

$$z^2 \equiv g \pmod{p}.$$

The number of square roots modulo p of g is $1+J$, where J is the Jacobi symbol $\left(\frac{g}{p}\right)$.

If $g = 0$, then there is one square root modulo p, namely $z = 0$. If $g \neq 0$, then g has either 0 or 2 square roots modulo p. If z is one square root, then the other is $p - z$.

A procedure for computing square roots modulo a prime is given in A.2.5.

RW Moduli.

If $n = pq$ is an RSA modulus with $p \equiv q \equiv 3 \pmod 4$ and $p \not\equiv q \pmod 8$, then n is called an *RW modulus.*

Denote by J_n the set of units u such that the Jacobi symbol $\left(\frac{u}{n}\right)$ equals 1. The set J_n comprises precisely half of the units. If f is a unit, then either f or $f/2 \bmod n$ is in J_n.

Exponentiation.

Let $\lambda(n)$ be the universal exponent of n (see Annex A.1.3). If c and d are congruent modulo $\lambda(n)/2$, then

$$a^c \equiv \pm a^d \pmod{n}$$

for every a in J_n. It follows that, if

$$de \equiv 1 \pmod{\lambda(n)/2},$$

then

$$a^{de} \equiv \pm a \pmod{n}$$

for every unit a in J_n.

IF-based cryptography.

Given an RW modulus n, a pair of integers d, e satisfying $de \equiv 1 \pmod{\lambda(n)/2}$ can be taken as the components of a key pair. Traditionally, e denotes the public key and d the private key. Given n and e, it is known that finding d is as difficult as factoring n. This in turn is necessary for computing a unit a given $a^e \bmod n$. For this reason, public-key cryptography based on key pairs of this type provides another example of IF-based cryptography. The difficulty of the integer factorization problem is discussed in Annex D.4.3.

A.2 Integer and Modular Arithmetic: Algorithms

A.2.1 Modular Exponentiation

Modular exponentiation can be performed efficiently by the *binary method* outlined below.

Input: a positive integer v, a modulus m, and an integer g modulo m.

Output: $g^v \bmod m$.

1. Let $v = v_r v_{r-1} \ldots v_1 v_0$ be the binary representation of v, where the most significant bit v_r of v is 1.
2. Set $x \leftarrow g$.
3. For i from $r - 1$ downto 0 do
   3.1 Set $x \leftarrow x^2 \bmod m$.
   3.2 If $v_i = 1$ then set $x \leftarrow gx \bmod m$.
4. Output x.

There are several modifications that improve the performance of this algorithm**.** These methods are summarized in [Gor98].

A.2.2 The Extended Euclidean Algorithm

The following algorithm computes efficiently the G.C.D. d of m and h. If m and h are relatively prime, the algorithm also finds the quotient g/h modulo m.

Input: an integer $m > 1$ and integers g and $h > 0$. (If only the G.C.D. of m and h is desired, no input g is required.)

Output: the G.C.D. d of m and h and, if $d = 1$, the integer c with $0 < c < m$ and $c \equiv g/h \pmod m$.

1. If $h = 1$ then output $d := 1$ and $c := g$ and stop.
2. Set $r_0 \leftarrow m$.
3. Set $r_1 \leftarrow h \bmod m$.
4. Set $s_0 \leftarrow 0$.
5. Set $s_1 \leftarrow g \bmod m$.
6. While $r_1 > 0$
   6.1 Set $q \leftarrow \lfloor r_0 / r_1 \rfloor$.
   6.2 Set $r_2 \leftarrow r_0 - q r_1 \bmod m$
   6.3 Set $s_2 \leftarrow s_0 - q s_1 \bmod m$
   6.4 Set $r_0 \leftarrow r_1$
   Set $r_1 \leftarrow r_2$
   Set $s_0 \leftarrow s_1$
   Set $s_1 \leftarrow s_2$
7. Output $d := r_0$.
8. If $r_0 = 1$ then output $c := s_0$

If m is prime, the quotient exists provided that $h \not\equiv 0 \pmod m$, and can be found efficiently using exponentiation via

$$c := g\, h^{m-2} \bmod m.$$

A.2.3 Evaluating Jacobi Symbols

The following algorithm efficiently computes the Jacobi symbol.

Input: an integer a and an odd integer $n > 1$.

Output: the Jacobi symbol $\left(\frac{a}{n}\right)$.

1. Set $x \leftarrow a$, $y \leftarrow n$, $J \leftarrow 1$
2. While $y > 1$
   - 2.1 Set $x \leftarrow (x \bmod y)$
   - 2.2 If $x > y/2$ then
     - 2.2.1 Set $x \leftarrow y - x$.
     - 2.2.2 If $y \equiv 3 \pmod 4$ then set $J \leftarrow -J$
   - 2.3 If $x = 0$ then set $x \leftarrow 1$, $y \leftarrow 0$, $J \leftarrow 0$
   - 2.4 While 4 divides x
     - 2.4.1 Set $x \leftarrow x/4$
   - 2.5 If 2 divides x then
     - 2.5.1 Set $x \leftarrow x/2$.
     - 2.5.2 If $y \equiv \pm 3 \pmod 8$ then set $J \leftarrow -J$
   - 2.6 If $x \equiv 3 \pmod 4$ and $y \equiv 3 \pmod 4$ then set $J \leftarrow -J$
   - 2.7 Switch x and y
3. Output J

If n is equal to a prime p, the Jacobi symbol can also be found efficiently using exponentiation via

$$\left(\frac{a}{p}\right) := a^{(p-1)/2} \bmod p.$$

A.2.4 Generating Lucas Sequences

Let $\mathcal{P}$ and $\mathcal{Q}$ be nonzero integers. The *Lucas sequences* U_k, and V_k for $\mathcal{P}$, $\mathcal{Q}$ are defined by

$$U_0 = 0,\ U_1 = 1, \text{ and } U_k = \mathcal{P}U_{k-1} - \mathcal{Q}U_{k-2} \text{ for } k \geq 2,$$
$$V_0 = 2,\ V_1 = \mathcal{P}, \text{ and } V_k = \mathcal{P}V_{k-1} - \mathcal{Q}V_{k-2} \text{ for } k \geq 2.$$

This recursion is adequate for computing U_k and V_k for small values of k. For large k, one can compute U_k and V_k modulo an odd prime p using the following algorithm.

Input: an odd prime p, integers $\mathcal{P}$ and $\mathcal{Q}$, and a positive integer k.

Output: $U_k \bmod p$ and $V_k \bmod p$.

1. Set $\Delta \leftarrow \mathcal{P}^2 - 4\mathcal{Q}$.
2. Let $k = k_r k_{r-1} \ldots k_1 k_0$ be the binary representation of k, where the leftmost bit k_r of k is 1.
3. Set $U \leftarrow 1$, $V \leftarrow \mathcal{P}$.
4. For i from $r - 1$ downto 0 do
   - 4.1 Set $(U, V) \leftarrow (UV \bmod p, (V^2 + \Delta U^2)/2 \bmod p)$.
   - 4.2 If $k_i = 1$ then set $(U, V) \leftarrow ((\mathcal{P}U + V)/2 \bmod p, (\mathcal{P}V + \Delta U)/2 \bmod p)$.
5. Output U and V.

NOTE—To perform the modular divisions of an integer n by 2 (needed in Step 4 of this algorithm), one can simply divide by 2 the integer n or $n + p$ (whichever is even). (The integer division by 2 can be accomplished by shifting the binary expansion of the dividend by one bit.)

A.2.5 Finding Square Roots Modulo a Prime

The following algorithm computes a square root z modulo p of $g \neq 0$.

Input: an odd prime p, and an integer g with $0 < g < p$.

Output: a square root modulo p of g if one exists. (In Case 3, the message "no square roots exist" is returned if none exists.)

I. *$p \equiv 3 \pmod 4$, that is $p = 4k + 3$ for some positive integer k. (See [Leh69].)*
   1. Compute (via A.2.1) and output $z := g^{k+1} \bmod p$.

II. *$p \equiv 5 \pmod 8$, that is $p = 8k + 5$ for some positive integer k. (See [Atk92].)*
   1. Compute $\gamma := (2g)^k \bmod p$ via A.2.1
   2. Compute $i := 2g\gamma^2 \bmod p$
   3. Compute and output $z := g\gamma(i - 1) \bmod p$

III *$p \equiv 1 \pmod 4$, that is $p = 4k + 1$ for some positive integer k. (See [Leh69].)*
   1. Set $\mathcal{Z} \leftarrow g$
   2. Generate random $\mathcal{P}$ with $0 < \mathcal{P} < p$.
   3. Compute via A.2.4 the Lucas sequence elements

$$U := U_{2k+1} \pmod p, \quad V := V_{2k+1} \pmod p.$$

   4. If $U = 0$ then output $z := V/2 \bmod p$ and stop.
   5. If $V = 0$ then output the message "no square roots exist" and stop.
   6. Go to Step 2.

NOTE—The modular division by 2 in Step 4 of case III can be carried out in the same way as in A.2.4.

In cases I and II, the algorithm produces a solution z provided that one exists. If it is unknown whether a solution exists, then the output z should be checked by comparing $w := z^2 \bmod p$ with g. If $w = g$, then z is a solution; otherwise no solutions exist. In case III, the algorithm performs the determination of whether or not a solution exists.

A.2.6 Finding Square Roots Modulo a Power of 2

If $r > 2$ and $a < 2^r$ is a positive integer congruent to 1 modulo 8, then there is a unique positive integer b less than 2^{r-2} such that $b^2 \equiv a \pmod{2^r}$. The number b can be computed efficiently using the following algorithm. The binary representations of the integers a, b, h are denoted as

$$a = a_{r-1}...a_1a_0,$$
$$b = b_{r-1}...b_1b_0,$$
$$h = h_{r-1}...h_1h_0.$$

Input: an integer $r > 2$, and a positive integer $a \equiv 1 \pmod 8$ less than 2^r.

Output: the positive integer b less than 2^{r-2} such that $b^2 \equiv a \pmod{2^r}$.

1. Set $h \leftarrow 1$.
2. Set $b \leftarrow 1$.

3. For j from 2 to $r - 2$ do
   If $h_{j+1} \neq a_{j+1}$ then
      Set $b_j \leftarrow 1$.
      If $j < r/2$
         then $h \leftarrow (h + 2^{j+1}b - 2^{2j}) \bmod 2^r$.
         else $h \leftarrow (h + 2^{j+1}b) \bmod 2^r$.
4. If $b_{r-2} = 1$ then set $b \leftarrow 2^{r-1} - b$.
5. Output b.

A.2.7 Computing the Order of a Given Integer Modulo a Prime

Let p be a prime and let g satisfy $1 < g < p$. The following algorithm determines the order of g modulo p. The algorithm is efficient only for small p.

Input: a prime p and an integer g with $1 < g < p$.

Output: the order k of g modulo p.

1. Set $b \leftarrow g$ and $j \leftarrow 1$.
2. Set $b \leftarrow gb \bmod p$ and $j \leftarrow j + 1$.
3. If $b > 1$ then go to Step 2.
4. Output j.

A.2.8 Constructing an Integer of a Given Order Modulo a Prime

Let p be a prime and let T divide $p - 1$. The following algorithm generates an element of $GF(p)$ of order T. The algorithm is efficient only for small p.

Input: a prime p and an integer T dividing $p - 1$.

Output: an integer u having order T modulo p.

1. Generate a random integer g between 1 and p.
2. Compute via A.2.7 the order k of g modulo p.
3. If T does not divide k then go to Step 1.
4. Output $u := g^{k/T} \bmod p$.

A.2.9 An Implementation of IF Signature Primitives

The following algorithm is an implementation of the IFSP-RSA1, IFSP-RSA2, and IFSP-RW signature primitives. Assuming e is small, it is more efficient for RW signatures than the primitive given in 8.2.8, because it combines the modular exponentiation and Jacobi symbol calculations. The algorithm requires that the primes p and q be known to the user; thus the private key can have either the second or third form specified in 8.1.3. (Note that this algorithm cannot be applied to IF signature verification since it requires knowledge of p and q, which is equivalent to knowledge of the private key.)

One-Time Computation: if the private key is the triple (p, q, d) (see Section 8.1.3), then the remaining components d_1, d_2, c can be computed via

$$c := q^{-1} \bmod p \text{ via A.2.2}$$
$$d_1 := d \bmod (p - 1)$$
$$d_2 := d \bmod (q - 1).$$

NOTE—For the smallest values of e, the values of d_1 and d_2 can be written down explicitly as follows.

RSA: if $e = 3$, then

$$d_1 = (2p - 1) / 3 \text{ and } d_2 = (2q - 1) / 3.$$

RW: if $e = 2$, then

$$d_1 = (p + 1)/4 \text{ and } d_2 = (3q - 1)/4$$

if $p \equiv 3 \pmod 8$, $q \equiv 7 \pmod 8$ and d odd, or
if $p \equiv 7 \pmod 8$, $q \equiv 3 \pmod 8$ and d even;

$$d_1 = (3p - 1)/4 \text{ and } d_2 = (q + 1)/4$$

if $p \equiv 3 \pmod 8$, $q \equiv 7 \pmod 8$ and d even, or
if $p \equiv 7 \pmod 8$, $q \equiv 3 \pmod 8$ and d odd.

In the RW case, one also computes via A.2.1

$$w_1 := \exp(2, p - 1 - d_1) \bmod p$$
$$w_2 := \exp(2, q - 1 - d_2) \bmod q.$$

Input: the (fixed) integers p, q, e, c, d_1, d_2, and (for RW) w_1 and w_2; the (per-message) integer f modulo pq. (It is unnecessary to store both p and d_1, if one can be conveniently computed from the other, as in the cases $e = 2$ or $e = 3$. The same remark holds for q and d_2.)

Output: the signature s of f.

The steps marked with an asterisk below are to be implemented only for RW and not for RSA.

1. Set $f_1 \leftarrow f \bmod p$.
2. Compute $j_1 := \exp(f_1, d_1) \bmod p$ via A.2.1.
*3. Compute $i_1 := \exp(j_1, e) \bmod p$.
*4. If $i_1 = f_1$ then set $u_1 \leftarrow 0$ else set $u_1 \leftarrow 1$.
5. Set $f_2 \leftarrow f \bmod q$.
6. Compute $j_2 := \exp(f_2, d_2) \bmod q$ via A.2.1.
*7. Compute $i_2 := \exp(j_2, e) \bmod q$.
*8. If $i_2 = f_2$ then set $u_2 \leftarrow 0$ else set $u_2 \leftarrow 1$.
*9. If $u_1 \neq u_2$ then compute

$$t_1 := j_1 w_1 \bmod p$$
$$t_2 := j_2 w_2 \bmod q$$

and go to Step 11.

10. Set

$$t_1 \leftarrow j_1, t_2 \leftarrow j_2$$

11. Compute

$$h := (t_1 - t_2)\, c \bmod p$$
$$t := t_2 + hq$$

12. If IFSP-RSA1, output $s := t$. Else (that is, if IFSP-RSA2 or IFSP-RW) output $s := (\min t, pq - t)$.

NOTE—Since the exponents are fixed in Steps 2 and 6, the exponentiations can be made more efficient using *addition chains* rather than the binary method A.2.1; see [Gor98].

A.3 Binary Finite Fields: Overview

A.3.1 Finite Fields

A *finite field* (or *Galois field*) is a set with finitely many elements in which the usual algebraic operations (addition, subtraction, multiplication, division by nonzero elements) are possible, and in which the usual algebraic laws (commutative, associative, distributive) hold. The *order* of a finite field is the number of elements it contains. If $q > 1$ is an integer, then a finite field of order q exists if q is a prime power and not otherwise.

The finite field of a given order is unique, in the sense that any two fields of order q display identical algebraic structure. Nevertheless, there are often many ways to represent a field. It is traditional to denote the finite field of order q by F_q or $GF(q)$; this Standard uses the latter notation for typographical reasons. It should be borne in mind that the expressions "the field $GF(q)$" and "the field of order q" usually imply a choice of field representation.

Although finite fields exist of every prime-power order, there are two kinds that are commonly used in cryptography.

— When q is a prime p, the field $GF(p)$ is called a *prime finite field*. The field $GF(p)$ is typically represented as the set of integers modulo p. See Annex A.1.2 for a discussion of these fields.
— When $q = 2^m$ for some m, the field $GF(2^m)$ is called a *binary finite field*. Unlike the prime field case, there are many common representations for binary finite fields. These fields are discussed below (A.3.3 to A.3.9).

These are the two kinds of finite fields considered in this Standard.

A.3.2 Polynomials over Finite Fields

A *polynomial over* $GF(q)$ is a polynomial with coefficients in $GF(q)$. Addition and multiplication of polynomials over $GF(q)$ are defined as usual in polynomial arithmetic, except that the operations on the coefficients are performed in $GF(q)$.

A polynomial over the prime field $GF(p)$ is commonly called a *polynomial modulo p*. Addition and multiplication are the same as for polynomials with integer coefficients, except that the coefficients of the results are reduced modulo p.

Example: Over the prime field $GF(7)$,

$$(t^2 + 4t + 5) + (t^3 + t + 3) = t^3 + t^2 + 5t + 1$$
$$(t^2 + 3t + 4)(t + 4) = t^3 + 2t + 2.$$

A *binary polynomial* is a polynomial modulo 2.

Example: Over the field $GF(2)$,

$$(t^3 + 1) + (t^3 + t) = t + 1$$
$$(t^2 + t + 1)(t + 1) = t^3 + 1.$$

A polynomial over *GF* (*q*) is *reducible* if it is the product of two smaller degree polynomials over *GF* (*q*); otherwise it is *irreducible*. For instance, the above examples show that $t^3 + 2t + 2$ is reducible over *GF* (7) and that the binary polynomial $t^3 + 1$ is reducible.

Every nonzero polynomial over *GF* (*q*) has a unique representation as the product of powers of irreducible polynomials. (This result is analogous to the fact that every positive integer has a unique representation as the product of powers of prime numbers.) The degree-1 factors correspond to the roots of the polynomial.

Polynomial congruences.

Modular reduction and congruences can be defined among polynomials over *GF* (*q*), in analogy to the definitions for integers given in A.1.1. To reduce a polynomial *a* (*t*) modulo a nonconstant polynomial *m* (t), one divides *a* (*t*) by *m* (*t*) by long division of polynomials and takes the remainder *r* (*t*). This operation is written

$$r(t) := a(t) \bmod m(t).$$

The remainder *r* (*t*) must either equal 0 or have degree smaller than that of *m* (*t*).

If *m* (*t*) = *t* – *c* for some element *c* of *GF* (*q*), then *a* (*t*) mod *m* (*t*) is just the constant *a* (*c*).

Two polynomials *a* (*t*) and *b* (*t*) are said to be *congruent modulo m* (*t*) if they have the same result upon reduction modulo *m* (*t*). This relationship is written

$$a(t) \equiv b(t) \pmod{m(t)}.$$

One can define addition, multiplication, and exponentiation of polynomials (to integral powers) modulo *m* (*t*), analogously to how they are defined for integer congruences in A.1.1. In the case of a prime field *GF* (*p*), each of these operations involves both reduction of the polynomials modulo *m* (*t*) and reduction of the coefficients modulo *p*.

A.3.3 Binary Finite Fields

If *m* is a positive integer, the *binary finite field GF* (2^m) consists of the 2^m possible bit strings of length *m*. Thus, for example,

$$GF(2^3) = \{000, 001, 010, 011, 100, 101, 110, 111\}.$$

The integer *m* is called the *degree* of the field.

For *m* = 1, the field *GF* (2) is just the set {0, 1} of integers modulo 2. The addition and multiplication operations are given by

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Addition.

For *m* > 1, addition of two elements is implemented by bitwise addition modulo 2. Thus, for example,

$$(11001) + (10100) = (01101)$$

Multiplication.

There is more than one way to implement multiplication in $GF(2^m)$. To specify a multiplication rule, one chooses a *basis representation* for the field. The basis representation is a rule for interpreting each bit string; the multiplication rule follows from this interpretation.

There are two common families of basis representations; *polynomial basis* representations and *normal basis* representations.

A.3.4 Polynomial Basis Representations

In a *polynomial basis representation*, each element of $GF(2^m)$ is represented by a different binary polynomial of degree less than m. More explicitly, the bit string $(a_{m-1} \ldots a_2\, a_1\, a_0)$ is taken to represent the binary polynomial

$$a_{m-1}\, t^{m-1} + \ldots + a_2 t^2 + a_1 t + a_0.$$

The *polynomial basis* is the set

$$B = \{t^{m-1}, \ldots, t^2, t, 1\}.$$

The addition of bit strings, as defined in A.3.3, corresponds to addition of binary polynomials.

Multiplication is defined in terms of an irreducible binary polynomial $p(t)$ of degree m, called the *field polynomial* for the representation. The product of two elements is simply the product of the corresponding polynomials, reduced modulo $p(t)$.

There is a polynomial basis representation for $GF(2^m)$ corresponding to each irreducible binary polynomial $p(t)$ of degree m. Irreducible binary polynomials exist of every degree. Roughly speaking, every one out of m binary polynomials of degree m is irreducible.

Trinomials and pentanomials.

The reduction of polynomials modulo $p(t)$ is particularly efficient if $p(t)$ has a small number of terms. The irreducibles with the least number of terms are the *trinomials* $t^m + t^k + 1$. Thus it is a common practice to choose a trinomial for the field polynomial, provided that one exists.

If an irreducible trinomial of degree m does not exist, then the next best polynomials are the *pentanomials* $t^m + t^a + t^b + t^c + 1$. For every m up to 1000, there exists either an irreducible trinomial or pentanomial of degree m. Section A.8 provides a table with an example for each m up to 1000, along with algorithms for constructing other irreducibles.

A.3.5 Normal Basis Representations

Normal bases.

A *normal basis* for $GF(2^m)$ is a set of the form

$$B = \{\, \theta,\ \theta^2,\ \theta^{2^2},\ \ldots,\ \theta^{2^{m-1}} \,\}$$

with the property that no subset of B adds to 0. (In the language of linear algebra, the elements of B are said to be *linearly independent.*) There exist normal bases for $GF(2^m)$ for every positive integer m.

The representation of $GF(2^m)$ via the normal basis B is carried out by interpreting the bit string $(a_0\ a_1\ a_2\ \ldots\ a_{m-1})$ as the element

$$a_0\theta + a_1\theta^2 + a_2\theta^{2^2} + \ldots + a_{m-1}\theta^{2^{m-1}}.$$

All of the elements of a normal basis B satisfy the same irreducible binary polynomial $p(t)$. This polynomial is called the *field polynomial* for the basis. An irreducible binary polynomial is called a *normal polynomial* if it is the field polynomial for a normal basis.

Gaussian normal bases.

Normal basis representations have the computational advantage that squaring an element can be done very efficiently (see Annex A.4.1). Multiplying distinct elements, on the other hand, can be cumbersome in general. For this reason, it is common to specialize to a class of normal bases, called *Gaussian normal bases*, for which multiplication is both simpler and more efficient.

Gaussian normal bases for $GF(2^m)$ exist whenever m is not divisible by 8. They include the *optimal normal bases*, which have the most efficient multiplication possible in a normal basis.

The *type* of a Gaussian normal basis is a positive integer measuring the complexity of the multiplication operation with respect to that basis. The smaller the type, the more efficient the multiplication. For a given m and T, the field $GF(2^m)$ can have at most one Gaussian normal basis of type T. Thus it is proper to speak of *the* type T Gaussian normal basis over $GF(2^m)$. See [ABV89] for more information on Gaussian normal bases. A table of Gaussian normal bases is given in Annex A.8.1

The Gaussian normal bases of types 1 and 2 have the most efficient multiplication rules of all normal bases. For this reason, they are called *optimal* normal bases. The type 1 Gaussian normal bases are called Type I optimal normal bases, and the type 2 Gaussian normal bases are called Type II optimal normal bases. See [Men93a] for more information on optimal normal bases.

A.3.6 Checking for a Gaussian Normal Basis

If $m > 1$ is not divisible by 8, the following algorithm [ABV89] tests for the existence of a Gaussian normal basis for $GF(2^m)$ of given type. (See also the table in A.8.1.)

Input: an integer $m > 1$ not divisible by 8; a positive integer T.

Output: if a type T Gaussian normal basis for $GF(2^m)$ exists, the message "True"; otherwise "False."

1. Set $p \leftarrow Tm + 1$.
2. If p is not prime then output "False" and stop.
3. Compute via A.2.7 the order k of 2 modulo p.
4. Set $h \leftarrow Tm / k$
5. Compute $d := \mathrm{GCD}(h, m)$ via A.2.2
6. If $d = 1$ then output "True"; else output "False."

Example: Let $m = 4$ and $T = 3$. Then $p = 13$, and 2 has order $k = 12$ modulo 13. Since $h = 1$ is relatively prime to $m = 4$, there does exist a Gaussian normal basis of type 3 for $GF(2^4)$.

A.3.7 The Multiplication Rule for a Gaussian Normal Basis

The following procedure produces the rule for multiplication with respect to a given Gaussian normal basis. (See Annex A.6 for a more general method applicable to all normal bases.)

Input: integers $m > 1$ and T for which there exists a type T Gaussian normal basis B for $GF(2^m)$.

Output: an explicit formula for the first coordinate of the product of two elements with respect to B.

1. Set $p \leftarrow Tm + 1$
2. Generate via A.2.8 an integer u having order T modulo p
3. Compute the sequence $F(1), F(2), \ldots, F(p-1)$ as follows:
   - 3.1 Set $w \leftarrow 1$
   - 3.2 For j from 0 to $T - 1$ do
     - Set $n \leftarrow w$
     - For i from 0 to $m - 1$ do
       - Set $F(n) \leftarrow i$
       - Set $n \leftarrow 2n \bmod p$
     - Set $w \leftarrow uw \bmod p$
4. If T is even, then set $J \leftarrow 0$, else set

$$J := \sum_{k=1}^{m/2} (a_{k-1}\, b_{m/2+k-1} + a_{m/2+k-1}\, b_{k-1})$$

5. Output the formula

$$c_0 = J + \sum_{k=1}^{p-2} a_{F(k+1)}\, b_{F(p-k)}$$

Example: For the type 3 normal basis for $GF(2^4)$, the values of F are given by

$F(1) = 0$	$F(5) = 1$	$F(9) = 0$
$F(2) = 1$	$F(6) = 1$	$F(10) = 2$
$F(3) = 0$	$F(7) = 3$	$F(11) = 3$
$F(4) = 2$	$F(8) = 3$	$F(12) = 2$

Therefore, after simplifying one obtains

$$c_0 = a_0 (b_1 + b_2 + b_3) + a_1 (b_0 + b_2) + a_2 (b_0 + b_1) + a_3 (b_0 + b_3).$$

Here c_0 is the first coordinate of the product

$$(*) \qquad (c_0\, c_1 \ldots c_{m-1}) = (a_0\, a_1 \ldots a_{m-1}) \times (b_0\, b_1 \ldots b_{m-1}).$$

The other coordinates of the product are obtained from the formula for c_0 by cycling the subscripts modulo m. Thus

$$c_1 = a_1 (b_2 + b_3 + b_0) + a_2 (b_1 + b_3) + a_3 (b_1 + b_2) + a_0 (b_1 + b_0),$$
$$c_2 = a_2 (b_3 + b_0 + b_1) + a_3 (b_2 + b_0) + a_0 (b_2 + b_3) + a_1 (b_2 + b_1),$$
$$c_3 = a_3 (b_0 + b_1 + b_2) + a_0 (b_3 + b_1) + a_1 (b_3 + b_0) + a_2 (b_3 + b_2).$$

A.3.8 A Multiplication Algorithm for a Gaussian Normal Basis

The formulae given in A.3.7 for $c_0, \ldots, c_{m-1}$ can be implemented in terms of a single expression. For

$$\underline{u} = (u_0\, u_1 \ldots u_{m-1}), \underline{v} = (v_0\, v_1 \ldots v_{m-1}),$$

let

$$F(\underline{u}, \underline{v})$$

be the expression with

$$c_0 = F(\underline{a}, \underline{b}).$$

Then the product (*) can be computed as follows.

1. Set $(u_0\ u_1 \ldots u_{m-1}) \leftarrow (a_0\ a_1 \ldots a_{m-1})$
2. Set $(v_0\ v_1 \ldots v_{m-1}) \leftarrow (b_0\ b_1 \ldots b_{m-1})$
3. For k from 0 to $m - 1$ do
   3.1 Compute
   $$c_k := F(\underline{u}, \underline{v})$$
   3.2 Set $u \leftarrow$ `LeftShift`(u) and $v \leftarrow$ `LeftShift`(v), where `LeftShift` denotes the circular left shift operation.
4. Output $c := (c_0\ c_1 \ldots c_{m-1})$.

In the above example,

$$F(\underline{u}, \underline{v}) := u_0 (v_1 + v_2 + v_3) + u_1 (v_0 + v_2) + u_2 (v_0 + v_1) + u_3 (v_0 + v_3).$$

If

$$a := (1000) = \theta \text{ and } b := (1101) = \theta + \theta^2 + \theta^8,$$

then

$$c_0 = F((1000), (1101)) = 0,$$
$$c_1 = F((0001), (1011)) = 0,$$
$$c_2 = F((0010), (0111)) = 1,$$
$$c_3 = F((0100), (1110)) = 0,$$

so that $c = ab = (0010)$.

A.3.9 Binary Finite Fields *(cont'd)*

Exponentiation.

If k is a positive integer and α is an element of $GF(2^m)$, then *exponentiation* is the operation of computing α^k. Section A.4.3 contains an efficient method for exponentiation.

Division.

If α and $\beta \neq 0$ are elements of the field $GF(2^m)$, then the *quotient* α/β is the element γ such that $\alpha = \beta\gamma$.

In the finite field $GF(2^m)$, modular division is possible for any denominator other than 0. The set of nonzero elements of $GF(2^m)$ is denoted $GF(2^m)^*$.

Section A.4.4 contains an efficient method for division.

Orders.

The *order* of an element γ of $GF(2^m)^*$ is the smallest positive integer v such that $\gamma^v = 1$. The order always exists and divides $2^m - 1$. If k and l are integers, then $\gamma^k = \gamma^l$ in $GF(2^m)$ if and only if $k \equiv l \pmod{v}$.

Generators.

If v divides $2^m - 1$, then there exists an element of $GF(2^m)^*$ having order v. In particular, there always exists an element γ of order $2^m - 1$ in $GF(2^m)^*$. Such an element is called a *generator* for $GF(2^m)^*$ because every element of $GF(2^m)^*$ is some power of γ.

Exponentiation and discrete logarithms.

Suppose that the element γ of $GF(2^m)^*$ has order v. Then an element η of $GF(2^m)^*$ satisfies $\eta = \gamma^l$ for some l if and only if $\eta^v = 1$. The exponent l is called the *discrete logarithm* of η (with respect to the base γ). The discrete logarithm is an integer modulo v.

DL-based cryptography.

Suppose that γ is an order-r element of $GF(2^m)^*$ where r is prime. Then a key pair can be defined as follows.

— The private key s is an integer modulo r.
— The corresponding public key w is an element of $GF(2^m)^*$ defined by $w := \gamma^s$.

It is necessary to compute a discrete logarithm in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the discrete logarithm problem. Thus it is an example of *DL-based cryptography*. The difficulty of the discrete logarithm problem is discussed in Annex D.4.1.

Field extensions.

Suppose that d and m are positive integers with d dividing m, and let **K** be a field $GF(2^m)$. Then there are precisely 2^d elements α of **K** satisfying

$$\alpha^{2^d} = \alpha.$$

The set **F** all of such elements forms a field $GF(2^d)$. The field **F** is called a *subfield* of **K**, and **K** is called an *extension field* of **F**.

Suppose that γ is an element of $GF(2^m)$ of prime order r. The prime r is said to be a *primitive* factor of $2^m - 1$ if r does not divide $2^d - 1$ for any $d < m$ dividing m. It follows from the previous paragraph that r is primitive if and only if γ is not an element of any proper subfield of $GF(2^m)$. (A *proper* subfield of a field **F** is any subfield except **F** itself).

Example.

Let **K** be the field $GF(2^4)$ given by the polynomial basis with field polynomial $p(t) := t^4 + t + 1$. Since

$$(t^2 + t)^3 = (t^2 + t + 1)^3 = 1,$$

then $\mathbf{F} = \{0, 1, t^2 + t, t^2 + t + 1\}$ forms the field $GF(2^2)$.

A.3.10 Parameters for Common Key Sizes

When selecting domain parameters for DL-based cryptography over binary fields, it is necessary to begin by choosing the following:

— The degree *m* of the field (so that the field has 2^m elements)
— The prime number *r* which is to serve as the order of the base point

These two numbers are related by the condition that *r* must be a primitive divisor of $2^m - 1$ (see Annex A.3.9). Moreover, it is a common practice to choose the two numbers to provide comparable levels of security. (See Annex D.4.1.4, Note 1.) Each parameter depends on the size of the symmetric keys which the DL system is being used to protect.

The following table lists several common symmetric key lengths. For each length is given a set of parameters of *m* and *r* which can be used in conjunction with symmetric keys of that length.

Key Size	*m*	*r*
40	189	2076174855442583929707535 27
56	384	4424998269453035935564731643147706 89
64	506	2822551529460330847604262086149015242689
80	1024	74556028256478842083373957362004549187833663426 57
112	2068	16284563554104115475096133741501580512316567430523445131616185803842288-3778013.
128	2880	19194878188585855612908061936944281464039294965346491767953330250249 20-8842371201

A.4 Binary Finite Fields: Algorithms

The following algorithms perform operations in a finite field $GF(2^m)$ having 2^m elements. The elements of $GF(2^m)$ can be represented either via a polynomial basis modulo the irreducible polynomial $p(t)$ or via a normal basis $\{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\}$.

A.4.1 Squaring and Square Roots

Polynomial Basis:

If

$$\alpha = \alpha_{m-1}t^{m-1} + \ldots + \alpha_2 t^2 + \alpha_1 t + \alpha_0,$$

then

$$\alpha^2 = \alpha_{m-1}t^{2m-2} + \ldots + \alpha_2 t^4 + \alpha_1 t^2 + \alpha_0 \bmod p(t).$$

To compute $\sqrt{\alpha}$, take α and square $m - 1$ times.

Normal Basis:

If α has representation

$$\alpha = (\alpha_0\ \alpha_1 \ldots \alpha_{m-1}),$$

then

$$\alpha^2 = (\alpha_{m-1}\ \alpha_0\ \alpha_1 \ldots \alpha_{m-2})$$

and

$$\sqrt{\alpha} = (\alpha_1 \ldots \alpha_{m-2}\ \alpha_{m-1}\ \alpha_0).$$

A.4.2 The Squaring Matrix

If it is necessary to perform frequent squarings in a fixed polynomial basis, it can be more efficient to use a *squaring matrix.* This is an *m*-by-*m* matrix with coefficients in *GF* (2).

Suppose that the field polynomial is

$$p(t) := t^m + c_{m-1}\, t^{m-1} + \ldots + c_1\, t + c_0.$$

Let $d_{0,j} \leftarrow c_j$ for $0 \le j < m$, and compute

$$\begin{aligned} t^m &= d_{0,m-1}\, t^{m-1} + \ldots + d_{0,1}\, t + d_{0,0} \\ t^{m+1} &= d_{1,m-1}\, t^{m-1} + \ldots + d_{1,1}\, t + d_{1,0} \\ &\cdots \\ t^{2m-2} &= d_{m-2,m-1}\, t^{m-1} + \ldots + d_{m-2,1}\, t + d_{m-2,0} \end{aligned}$$

by repeated multiplication by *t*. Define the matrix

$$S := \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & \ddots & \vdots \\ s_{m,1} & \cdots & s_{m,m} \end{pmatrix}$$

where

$$s_{i,j} := \begin{cases} d_{m-2i,m-j} & \text{if } i \le \lfloor m/2 \rfloor \\ 1 & \text{if } i > \lfloor m/2 \rfloor \text{ and } 2i = j + m \\ 0 & \text{otherwise.} \end{cases}$$

If $(a_0\ a_1 \ldots a_{m-1})$ represents the field element α, then the representation for α^2 is

$$(b_0\ b_1 \ldots b_{m-1}) = (a_0\ a_1 \ldots a_{m-1})\ S.$$

A.4.3 Exponentiation

Exponentiation can be performed efficiently by the *binary method* outlined below.

Input: a positive integer *k*, a field *GF* (2^m) and a field element α.

Output: α^k.

1. Let $k = k_r\ k_{r-1} \ldots k_1\ k_0$ be the binary representation of *k*, where the most significant bit k_r of *k* is 1.
2. Set $x \leftarrow \alpha$.

3. For i from $r - 1$ downto 0 do
   3.1 Set $x \leftarrow x^2$.
   3.2 If $k_i = 1$ then set $x \leftarrow \alpha\, x$.
4. Output x.

There are several modifications that improve the performance of this algorithm. These methods are summarized in [Gor98].

A.4.4 Division

The quotient α / β can be computed directly (i.e. in one step by an algorithm with inputs α and β), or indirectly (by computing the multiplicative inverse β^{-1} and then multiplying it by α). There are two common methods of performing division in a finite field $GF(2^m)$, one direct and one indirect.

Method I: the Extended Euclidean Algorithm.

This algorithm produces the quotient directly (also it can be used for multiplicative inversion of β, and so for indirect division, by using as input 1 in place of α). By $\lfloor r_0(t) / r_1(t) \rfloor$ is meant the quotient upon polynomial division, dropping any remainder.

Input: a field $GF(2^m)$, and field elements α and $\beta \neq 0$.

Output: $\gamma := \alpha / \beta$.

1. Set $r_0(t) \leftarrow p(t)$
2. Set $r_1(t) \leftarrow \beta$
3. Set $s_0(t) \leftarrow 0$
4. Set $s_1(t) \leftarrow \alpha$
5. While $r_1(t) \neq 0$
   5.1 Set $q(t) \leftarrow \lfloor r_0(t) / r_1(t) \rfloor$.
   5.2 Set $r_2(t) \leftarrow r_0(t) + q(t) r_1(t)$
   5.3 Set $s_2(t) \leftarrow s_0(t) + q(t) s_1(t)$
   5.4 Set $r_0(t) \leftarrow r_1(t)$
       Set $r_1(t) \leftarrow r_2(t)$
   5.5 Set $s_0(t) \leftarrow s_1(t)$
       Set $s_1(t) \leftarrow s_2(t)$
6. Output $\gamma := s_0(t)$

NOTES

1—An efficient hardware implementation of this procedure is described in [Ber68].

2—The Extended Euclidean Algorithm uses a polynomial basis representation for $GF(2^m)$. If a normal basis representation is being used, then one can divide using this algorithm only by converting the inputs α and β to a polynomial basis representation, performing the division, and converting the output γ back to normal basis form. (See Annex A.7.1 for methods of converting between different basis representations.)

Method II: Exponentiation.

The multiplicative inverse of β can be found efficiently in either basis representation via

$$\beta^{-1} = \beta^{k},$$

where k is any positive integer satisfying

$$k \equiv -1 \pmod{r},$$

where r is the order of β. In particular, it is always the case that

$$\beta^{-1} = \beta^{2^m-2}.$$

If a general-purpose exponentiation algorithm such as A.4.3 is used, then the best choice is $k := r - 1$. However, there is also a specialized algorithm [ITT86] for exponentiating to the power $k = 2^m - 2$, which is more efficient than the generic method. The efficiency improvements are especially significant when squaring can be done quickly, e.g. in a normal basis representation. The procedure is given below.

Input: a field GF (2^m) and a nonzero field element β.

Output: the reciprocal β^{-1}.

1. Let $m - 1 = b_r\ b_{r-1}\ ...\ b_1\ b_0$ be the binary representation of $m - 1$, where the most significant bit b_r of $m - 1$ is 1.
2. Set $\eta \leftarrow \beta$ and $k \leftarrow 1$
3. For i from $r - 1$ downto 0 do
   - 3.1 Set $\mu \leftarrow \eta$
   - 3.2 For $j = 1$ to k do
     - 3.2.1 Set $\mu \leftarrow \mu^2$
   - 3.3 Set $\mu \leftarrow \mu\eta$ and $k \leftarrow 2k$
   - 3.4 If $b_i = 1$, then set $\eta \leftarrow \eta^2\beta$ and $k \leftarrow k + 1$
4. Output η^2.

A.4.5 Trace

If α is an element of $GF(2^m)$, the *trace* of α is

$$\mathrm{Tr}(\alpha) = \alpha + \alpha^2 + \alpha^{2^2} + ... + \alpha^{2^{m-1}}.$$

The value of Tr (α) is 0 for half the elements of $GF(2^m)$, and 1 for the other half.

The trace can be computed efficiently as follows.

Normal Basis:

If α has representation ($\alpha_0\,\alpha_1 ... \alpha_{m-1}$), then

$$\mathrm{Tr}(\alpha) = \alpha_0 \oplus \alpha_1 \oplus ... \oplus \alpha_{m-1}.$$

Polynomial Basis:

The basic algorithm inputs $\alpha \in GF(2^m)$ and outputs $T = \mathrm{Tr}(\alpha)$.

1. Set $T \leftarrow \alpha$.
2. For i from 1 to $m - 1$ do
   - 2.1 $T \leftarrow T^2 + \alpha$.

3. Output T.

If many traces are to be computed with respect to a fixed polynomial basis

$$\{t^{m-1}, \ldots, t, 1\},$$

then it is more efficient to compute and store the element

$$\tau = (\tau_{m-1} \ldots \tau_1 \tau_0)$$

where each coordinate

$$\tau_j = \mathrm{Tr}\,(t^j)$$

is computed via the basic algorithm. Subsequent traces can be computed via

$$\mathrm{Tr}\,(\alpha) = \alpha \cdot \tau,$$

where the "dot product" of the bit strings is given by bitwise AND (or bitwise multiplication).

A.4.6 Half-Trace

If m is odd, the *half-trace* of $\alpha \in GF\,(2^m)$ is

$$\mathrm{HfTr}\,(\alpha) = \alpha + \alpha^{2^2} + \alpha^{2^4} + \ldots + \alpha^{2^{m-1}}.$$

The following algorithm inputs $\alpha \in GF\,(2^m)$ and outputs $H = \mathrm{HfTr}\,(\alpha)$

1. Set $H \leftarrow \alpha$.
2. For i from 1 to $(m-1)/2$ do
   2.1 $H \leftarrow H^{2^2}$.
   2.2 $H \leftarrow H^{2^2} + \alpha$.
3. Output H.

A.4.7 Solving Quadratic Equations over *GF* (2^m)

If β is an element of $GF\,(2^m)$, then the equation

$$z^2 + z = \beta$$

has $2 - 2T$ solutions over $GF\,(2^m)$, where $T = \mathrm{Tr}\,(\beta)$. Thus, there are either 0 or 2 solutions. If z is one solution, then the other solution is $z + 1$. In the case $\beta = 0$, the solutions are 0 and 1.

The following algorithms compute a solution if one exists.

Input: a field $GF\,(2^m)$ along with a polynomial or normal basis for representing its elements; an element $\beta \neq 0$.

Output: an element z for which $z^2 + z = \beta$, if such an element exists.

Normal basis:

1. Let $(\beta_0 \beta_1 ... \beta_{m-1})$ be the representation of β
2. Set $z_0 \leftarrow 0$
3. For $i = 1$ to $m - 1$ do
   3.1 Set $z_i \leftarrow z_{i-1} \oplus \beta_i$
4. Output $z \leftarrow (z_0 z_1 ... z_{m-1})$

Polynomial basis:

If m is odd, then compute z := half-trace of β via A.4.6. For m even, proceed as follows.

1. Choose random $\rho \in GF(2^m)$
2. Set $z \leftarrow 0$ and $w \leftarrow \rho$.
3. For i from 1 to $m - 1$ do
   3.1 Set $z \leftarrow z^2 + w^2 \beta$.
   3.2 Set $w \leftarrow w^2 + \rho$.
4. If $w = 0$ then go to Step 1
5. Output z.

If the latter algorithm is to be used repeatedly for the same field, and memory is available, then it is more efficient to precompute and store ρ and the values of w. Any element of trace 1 will serve as ρ, and the values of w depend only on ρ and not on β.

Both of the above algorithms produce a solution z provided that one exists. If it is unknown whether a solution exists, then the output z should be checked by comparing $\gamma := z^2 + z$ with β. If $\gamma = \beta$, then z is a solution; otherwise no solutions exist.

A.5 Polynomials over a Finite Field

The computations below can take place either over a prime field (having a prime number p of elements) or over a binary field (having 2^m elements).

A.5.1 Exponentiation Modulo a Polynomial

If k is a positive integer and $f(t)$ and $m(t)$ are polynomials with coefficients in the field $GF(q)$, then $f(t)^k \bmod m(t)$ can be computed efficiently by the *binary method* outlined below.

Input: a positive integer k, a field $GF(q)$, and polynomials $f(t)$ and $m(t)$ with coefficients in $GF(q)$.

Output: the polynomial $f(t)^k \bmod m(t)$.

1. Let $k = k_r k_{r-1} ... k_1 k_0$ be the binary representation of k, where the most significant bit k_r of k is 1.
2. Set $u(t) \leftarrow f(t) \bmod m(t)$.
3. For i from $r - 1$ downto 0 do
   3.1 Set $u(t) \leftarrow u(t)^2 \bmod m(t)$.
   3.2 If $k_i = 1$ then set $u(t) \leftarrow u(t) f(t) \bmod m(t)$.
4. Output $u(t)$.

There are several modifications that improve the performance of this algorithm. These methods are summarized in [Gor98].

A.5.2 G.C.D.'s over a Finite Field

If $f(t)$ and $g(t) \neq 0$ are two polynomials with coefficients in the field $GF(q)$, then there is a unique monic polynomial $d(t)$ of largest degree which divides both $f(t)$ and $g(t)$. The polynomial $d(t)$ is called the *greatest common divisor* or *G.C.D.* of $f(t)$ and $g(t)$. The following algorithm computes the G.C.D. of two polynomials.

Input: a finite field $GF(q)$ and two polynomials $f(t)$, $g(t) \neq 0$ over $GF(q)$.

Output: $d(t) = \text{GCD}(f(t), g(t))$.

1. Set $a(t) \leftarrow f(t)$, $b(t) \leftarrow g(t)$.
2. While $b(t) \neq 0$
   - 2.1 Set $c(t) \leftarrow$ the remainder when $a(t)$ is divided by $b(t)$.
   - 2.2 Set $a(t) \leftarrow b(t)$.
   - 2.3 Set $b(t) \leftarrow c(t)$.
3. Set $\alpha \leftarrow$ the leading coefficient of $a(t)$.
4. Set $d(t) \leftarrow \alpha^{-1} a(t)$.
5. Output $d(t)$.

A.5.3 Factoring Polynomials over *GF* (*p*) (Special Case)

Let $f(t)$ be a polynomial with coefficients in the field $GF(p)$, and suppose that $f(t)$ factors into distinct irreducible polynomials of degree d. (This is the special case needed in A.14.) The following algorithm finds a random degree-d factor of $f(t)$ efficiently.

Input: a prime $p > 2$, a positive integer d, and a polynomial $f(t)$ which factors modulo p into distinct irreducible polynomials of degree d.

Output: a random degree-d factor of $f(t)$.

1. Set $g(t) \leftarrow f(t)$.
2. While $\deg(g) > d$
   - 2.1 Choose $u(t) \leftarrow$ a random monic polynomial of degree $2d - 1$.
   - 2.2 Compute via A.5.1
     $$c(t) := u(t)^{(p^d-1)/2} \bmod g(t).$$
   - 2.3 Set $h(t) \leftarrow \text{GCD}(c(t) - 1, g(t))$.
   - 2.4 If $h(t)$ is constant or $\deg(g) = \deg(h)$ then go to Step 2.1.
   - 2.5 If $2\deg(h) > \deg(g)$ then set $g(t) \leftarrow g(t) / h(t)$; else $g(t) \leftarrow h(t)$.
3. Output $g(t)$.

A.5.4 Factoring Polynomials over *GF* (2) (Special Case)

Let $f(t)$ be a polynomial with coefficients in the field $GF(2)$, and suppose that $f(t)$ factors into distinct irreducible polynomials of degree d. (This is the special case needed in A.14.) The following algorithm finds a random degree-d factor of $f(t)$ efficiently.

Input: a positive integer d, and a polynomial $f(t)$ which factors modulo 2 into distinct irreducible polynomials of degree d.

Output: a random degree-d factor of $f(t)$.

1. Set $g(t) \leftarrow f(t)$.
2. While $\deg(g) > d$

2.1 Choose $u(t) \leftarrow$ a random monic polynomial of degree $2d - 1$.
2.2 Set $c(t) \leftarrow u(t)$.
2.3 For i from 1 to $d - 1$ do
2.3.1 $c(t) \leftarrow c(t)^2 + u(t) \bmod g(t)$.
2.4 Compute $h(t) := \text{GCD}(c(t), g(t))$ via A.5.2.
2.5 If $h(t)$ is constant or $\deg(g) = \deg(h)$ then go to Step 2.1.
2.6 If $2 \deg(h) > \deg(g)$ then set $g(t) \leftarrow g(t) / h(t)$; else $g(t) \leftarrow h(t)$.
3. Output $g(t)$.

A.5.5 Checking Polynomials over *GF* (2^r) for Irreducibility

If $f(t)$ is a polynomial with coefficients in the field $GF(2^r)$, then $f(t)$ can be tested efficiently for irreducibility using the following algorithm.

Input: a polynomial $f(t)$ with coefficients in $GF(2^r)$.

Output: the message "True" if $f(t)$ is irreducible; the message "False" otherwise.

1. Set $d \leftarrow$ degree of $f(t)$.
2. Set $u(t) \leftarrow t$.
3. For i from 1 to $\lfloor d/2 \rfloor$ do
3.1 For j from 1 to r do
Set $u(t) \leftarrow u(t)^2 \bmod f(t)$
Next j
3.2 Set $g(t) \leftarrow \text{GCD}(u(t) + t,\ f(t))$.
3.3 If $g(t) \neq 1$ then output "False" and stop.
3. Output "True."

A.5.6 Finding a Root in *GF* (2^m) of an Irreducible Binary Polynomial

If $f(t)$ is an irreducible polynomial modulo 2 of degree d dividing m, then $f(t)$ has d distinct roots in the field $GF(2^m)$. A random root can be found efficiently using the following algorithm.

Input: an irreducible polynomial modulo 2 of degree d, and a field $GF(2^m)$, where d divides m.

Output: a random root of $f(t)$ in $GF(2^m)$.

1. Set $g(t) \leftarrow f(t)$.
2. While $\deg(g) > 1$
2.1 Choose random $u \in GF(2^m)$.
2.2 Set $c(t) \leftarrow ut$.
2.3 For i from 1 to $m - 1$ do
2.3.1 $c(t) \leftarrow c(t)^2 + ut \bmod g(t)$.
2.4 Set $h(t) \leftarrow \text{GCD}(c(t), g(t))$.
2.5 If $h(t)$ is constant or $\deg(g) = \deg(h)$ then go to Step 2.1.
2.6 If $2 \deg(h) > \deg(g)$ then set $g(t) \leftarrow g(t) / h(t)$; else $g(t) \leftarrow h(t)$.
3. Output $g(0)$.

A.5.7 Embedding in an Extension Field

Given a field $\mathbf{F} = GF(2^d)$, the following algorithm embeds $\mathbf{F}$ into an extension field $\mathbf{K} = GF(2^{de})$.

Input: integers d and e; a (polynomial or normal) basis B for $\mathbf{F} = GF(2^d)$ with field polynomial $p(t)$; a (polynomial or normal) basis for $\mathbf{K} = GF(2^{de})$.

Output: an embedding of **F** into **K**; that is a function taking each $\alpha \in \mathbf{F}$ to a corresponding element β of **K**.

1. Compute via A.5.6 a root $\lambda \in \mathbf{K}$ of $p(t)$.
2. If B is a polynomial basis then output

$$\beta := a_{m-1}\lambda^{m-1} + \ldots + a_2\lambda^2 + a_1\lambda + a_0,$$

where $(a_{m-1} \ldots a_1\ a_0)$ is the bit string representing α with respect to B.

3. If B is a normal basis then output

$$\beta := a_0\lambda + a_1\lambda^2 + a_2\lambda^{2^2} + \cdots + a_{m-1}\lambda^{2^{m-1}},$$

where $(a_0\ a_1 \ldots a_{m-1})$ is the bit string representing α with respect to B.

A.6 General Normal Bases for Binary Fields

The algorithms in this section allow computation with any normal basis. (In the case of Gaussian normal bases, the algorithms of A.3 are more efficient.)

A general normal basis is specified by its field polynomial $p(t)$ (see Annex A.3.5). The rule for multiplication with respect to a general normal basis is most easily described via a *multiplication matrix*. The multiplication matrix is an m-by-m matrix with entries in $GF(2)$. A description of how to multiply using the multiplication matrix is given in A.6.4.

A.6.1 Checking for a Normal Basis

The following algorithm checks whether an irreducible polynomial $p(t)$ over $GF(2)$ is normal, and if so, outputs two matrices to be used later in deriving the multiplication rule for the corresponding normal basis.

Input: an irreducible polynomial $p(t)$ degree m over $GF(2)$.

Output: if $p(t)$ is normal, two m-by-m matrices over $GF(2)$. If not, the message "not normal."

1. Compute

$$t = a_{0,0} + a_{0,1}t + a_{0,2}t^2 + \ldots + a_{0,m-1}t^{m-1} \pmod{p(t)}$$

$$t^2 = a_{1,0} + a_{1,1}t + a_{1,2}t^2 + \ldots + a_{1,m-1}t^{m-1} \pmod{p(t)}$$

$$t^4 = a_{2,0} + a_{2,1}t + a_{2,2}t^2 + \ldots + a_{2,m-1}t^{m-1} \pmod{p(t)}$$

$$\ldots$$

$$t^{2^{m-1}} = a_{m-1,0} + a_{m-1,1}t + a_{m-1,2}t^2 + \ldots + a_{m-1,m-1}t^{m-1} \pmod{p(t)}$$

modulo 2 via repeated squaring.

2. Set

$$A := \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,m-1} \end{pmatrix}.$$

3. Attempt to compute the inverse B of the matrix A over $GF(2)$.
4. If A is not invertible, output the message "not normal" and stop; otherwise output A and B.

A.6.2 Finding a Normal Basis

If m is a multiple of 8, then the Gaussian normal basis construction of A.3 does not apply. In this case (or in any case where a non-Gaussian normal basis is desired), it is possible to find a normal basis by searching for a normal polynomial of degree m over $GF(2)$.

The procedure is to produce an irreducible polynomial of degree m over $GF(2)$, test it for normality via A.6.1, and repeat until a normal polynomial is found. To produce an irreducible, one can use the table in Annex A.8.1, the random search routine in A.8.2, or one of the techniques in A.8.3 through A.8.5.

The probability of a randomly chosen irreducible polynomial of degree m over $GF(2)$ being normal is at least 0.2 if $m \le 2000$, and is usually much higher. Thus one expects to have to try no more than about five irreducibles before finding a normal polynomial.

A.6.3 Computing the Multiplication Matrix

The following algorithm computes the multiplication matrix of a basis B that has been deemed "normal" by A.6.1.

Input: a normal polynomial

$$p(t) = t^m + c_{m-1}\, t^{m-1} + \ldots + c_1\, t + c_0$$

over $GF(2)$; the matrices A and B computed by A.6.1.

Output: the multiplication matrix M for $p(t)$.

1. Set

$$C := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ c_0 & c_1 & c_2 & \cdots & c_{m-1} \end{pmatrix}.$$

2. Compute $D := ACB$ over $GF(2)$. (We denote by $d_{i,j}$ the $(i, j)^{\text{th}}$ entry of D, for $0 \le i,j < m$.)
3. Set $\mu_{i,j} := d_{j-i,-i}$ for each i and j, where each subscript is regarded as an integer modulo m.
4. Output

$$M := \begin{pmatrix} \mu_{0,0} & \mu_{0,1} & \cdots & \mu_{0,m-1} \\ \mu_{1,0} & \mu_{1,1} & \cdots & \mu_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{m-1,0} & \mu_{m-1,1} & \cdots & \mu_{m-1,m-1} \end{pmatrix}.$$

Example: Let $p(t) = t^4 + t^3 + t^2 + t + 1$. Then

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

so that

$$B = A^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Also

$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

so that

$$D = ACB = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Therefore

$$M = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

In the case of a Gaussian normal basis, the multiplication matrix M can be written down immediately from the explicit multiplication rule. For $0 \le i < m$ and $0 \le j < m$, $\mu_{i,j}$ is 1 if $a_i b_j$ appears in the formula for c_0, and 0 otherwise.

Example: If B is the type 3 normal basis over $GF(2^4)$, then it was found in A.3.7 that, if

$$(c_0\ c_1\ \ldots\ c_{m-1}) = (a_0\ a_1 \ldots\ a_{m-1}) \times (b_0\ b_1\ \ldots\ b_{m-1}),$$

then

$$c_0 = a_0 (b_1 + b_2 + b_3) + a_1 (b_0 + b_2) + a_2 (b_0 + b_1) + a_3 (b_0 + b_3).$$

Thus the multiplication matrix for B is

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

A.6.4 Multiplication

The following algorithm implements the multiplication of two elements of a field $GF(2^m)$ represented in terms of a normal basis.

Input: the multiplication matrix M for the field $GF(2^m)$; field elements $a = (a_0\ a_1\ \ldots\ a_{m-1})$ and $b = (b_0\ b_1\ \ldots\ b_{m-1})$.

Output: the product $c = (c_0\ c_1\ \ldots\ c_{m-1})$ of a and b.

1. Set $x \leftarrow a$.
2. Set $y \leftarrow b$.
3. For k from 0 to $m - 1$ do
   3.1 Compute via matrix multiplication

$$c_k := x\, M\, y^{\mathrm{tr}}$$

   where y^{tr} denotes the matrix transpose of the vector y.
   3.2 Set $x \leftarrow$ `LeftShift`(x) and $y \leftarrow$ `LeftShift`(y), where `LeftShift` denotes the circular left shift operation.
4. Output $c = (c_0\ c_1\ \ldots\ c_{m-1})$.

Example: Let $\theta \in GF(2^4)$ be a root of the normal polynomial $p(t) = t^4 + t^3 + t^2 + t + 1$. We find the product of the elements

$$a = (1000) = \theta$$

and

$$b = (1101) = \theta + \theta^2 + \theta^8.$$

The multiplication matrix for $p(t)$ was found in A.6.3 to be

$$M = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

Thus

$$c_0 = (1000)\, M \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = 0,$$

$$c_1 = (0001)\, M \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = 1,$$

$$c_2 = (0010)\, M \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = 1,$$

$$c_3 = (0100)\, M \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} = 1,$$

so that $c = ab = (0111)$.

A.7 Basis Conversion for Binary Fields

In order to use DL and EC protocols over a finite field $\mathbf{F} = GF(2^m)$, it is necessary for the users to agree on a common field degree m but not on a common basis representation of $\mathbf{F}$. If the users do not agree on the basis, however, it is necessary for one or both users to perform a *change of basis*, i.e. to convert incoming and/or outgoing field elements between one's own representation and that of the other user's.

A.7.1 The Change-of-Basis Matrix

Suppose that a user has received a field element that is represented in terms of the basis B_0, while the user himself uses basis B_1 for his own computations. This user must now perform a change of basis from B_0 to B_1 on this element. This can be accomplished using a *change-of-basis* matrix from B_0 to B_1. This is an m-by-m matrix with entries in $GF(2)$.

Suppose that Γ is the change-of-basis matrix from B_0 to B_1, and that $\underline{u}$ is the bit string representing $\alpha \in GF(2^m)$ in terms of B_0. Then

$$\underline{v} = \underline{u}\,\Gamma$$

is the bit string representing α in terms of B_1.

If Γ is the change-of-basis matrix from B_0 to B_1, then the inverse Γ^{-1} of the matrix over $GF(2)$ is the change-of-basis matrix from B_1 to B_0. Thus, given a bit string v representing $\alpha \in GF(2^m)$ in terms of B_1, the bit string

$$\underline{u} = \underline{v}\,\Gamma^{-1}$$

represents α in terms of B_0.

If Γ is the change-of-basis matrix from B_0 to B_1, and Γ' is the change-of-basis matrix from B_1 to B_2, then the matrix product $\Gamma'' = \Gamma\,\Gamma'$ is the change-of-basis matrix from B_0 to B_2.

The algorithm for calculating the change-of-basis matrix is given in A.7.3. A special case is treated in A.7.4.

Example: Suppose that B_0 is the Type I optimal normal basis for $GF(2^4)$ and B_1 is the polynomial basis with field polynomial $t^4 + t + 1$. Then the change-of-basis matrix from B_0 to B_1 is

$$\Gamma = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

(see Annex A.7.3). If α is the element of $GF(2^4)$ represented by (1001) with respect to B_0, then its representation with respect to B_1 is

$$(1001)\,\Gamma = (0100).$$

The inverse of Γ over $GF(2)$ is

$$\Gamma^{-1} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

If β is the element of $GF(2^4)$ represented by (1101) with respect to B_1, then its representation with respect to B_0 is

$$(1101)\,\Gamma^{-1} = (0111).$$

A.7.2 The Field Polynomial of a Gaussian Normal Basis

In order to compute the change-of-basis matrix for conversion to or from a normal basis B, it is necessary to compute the field polynomial for that basis (see Annex A.3.5).

If $GF(2^m)$ has a type T Gaussian normal basis over $GF(2)$, the following algorithm efficiently produces its field polynomial.

Input: positive integers m and T for which there exists a type T Gaussian normal basis for $GF(2^m)$ over $GF(2)$.

Output: the field polynomial $p(t)$ for the basis.

I. *$T = 1$. (Type I optimal normal basis)*
   1. Set $p(t) \leftarrow t^m + t^{m-1} + \ldots + t + 1$.
   2. Output $p(t)$.

II. *$T = 2$. (Type II optimal normal basis)*
   1. Set $q(t) \leftarrow 1$.
   2. Set $p(t) \leftarrow t + 1$.
   3. For $i = 1$ to $m - 1$ do
      $r(t) \leftarrow q(t)$
      $q(t) \leftarrow p(t)$
      $p(t) := t\,q(t) + r(t)$
   4. Output $p(t)$.

III. *$T \geq 3$. (Sub-optimal Gaussian normal basis)*
   1. Set $p \leftarrow T\,m + 1$.
   2. Generate via A.2.8 an integer u having order T modulo p.
   3. For k from 1 to m do
      3.1 Compute

$$e_k = \sum_{j=0}^{T-1} \exp\left(\frac{2^k u^j \pi i}{p}\right).$$

   4. Compute the polynomial

$$f(t) := \prod_{k=1}^{m} (t - e_k).$$

      (The polynomial $f(t)$ has integer coefficients.)
   5. Output $p(t) := f(t) \bmod 2$.

NOTE—The complex numbers e_k must be computed with sufficient accuracy to identify each coefficient of the polynomial $f(t)$. Since each such coefficient is an integer, this means that the error incurred in calculating each coefficient should be less than 1/2.

Example: Let $m = 4$ and $T = 3$. By the example of A.3.6, there exists a Gaussian normal basis of type 3 for $GF(2^4)$ over $GF(2)$. Now $p = 13$, and $u = 3$ has order $T = 3$ modulo $p = 13$.

The complex numbers e_k are

$$e_1 := e^{2\pi i/13} + e^{6\pi i/13} - e^{5\pi i/13}$$
$$\approx 0.6513878188659973233 + 0.522415803456407715i$$

$$e_2 := e^{4\pi i/13} + e^{12\pi i/13} + e^{10\pi i/13}$$
$$\approx -1.1513878188659973233 + 1.7254221884220093641i$$

$$e_3 := e^{8\pi i/13} - e^{11\pi i/13} - e^{7\pi i/13}$$
$$\approx 0.6513878188659973233 - 0.522415803456407715\,i$$

$$e_4 := -e^{\pi i/13} - e^{3\pi i/13} - e^{9\pi i/13}$$
$$\approx -1.1513878188659973233 - 1.7254221884220093641\,i$$

Thus

$$f(t) := (t - e_1)(t - e_2)(t - e_3)(t - e_4) = t^4 + t^3 + 2t^2 - 4t + 3$$

so that $p(t) := t^4 + t^3 + 1$ is the field polynomial for the basis.

A.7.3 Computing the Change-of-Basis Matrix

The following algorithm efficiently calculates the change-of-basis matrix from a (polynomial or normal) basis B_0 to one's own (polynomial or normal) basis B_1.

Input: a field degree m; a (polynomial or normal) basis B_0 with field polynomial $p_0(u)$; a (polynomial or normal) basis B_1 with field polynomial $p_1(t)$. (Note: in the case of a Gaussian normal basis, the field polynomial can be computed via A.7.2.)

Output: the change-of-basis matrix Γ from B_0 to B_1.

1. Let u be a root of $p_0(u)$ represented with respect to B_1. (u can be computed via A.5.6.)
2. Define the elements $e_0, \ldots, e_{m-1}$ via

$$e_j = \begin{cases} t^{m-1-j} & \text{if } B_1 = \{t^{m-1}, \ldots, t^2, t, 1\} \\ \theta^{2^j} & \text{if } B_1 = \{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\}. \end{cases}$$

3. Compute the elements $\gamma_{i,j}$ for $0 \le i < m$, $0 \le j < m$ as follows.
   — If B_0 is a polynomial basis, then compute

$$1 = \sum_{j=0}^{m-1} \gamma_{m-1,j} e_j$$
$$u = \sum_{j=0}^{m-1} \gamma_{m-2,j} e_j$$
$$\vdots$$
$$u^{m-2} = \sum_{j=0}^{m-1} \gamma_{1,j} e_j$$
$$u^{m-1} = \sum_{j=0}^{m-1} \gamma_{0,j} e_j$$

   by repeated multiplication by u.
   — If B_0 is a normal basis, then compute

$$u = \sum_{j=0}^{m-1} \gamma_{0,j} e_j$$
$$u^2 = \sum_{j=0}^{m-1} \gamma_{1,j} e_j$$
$$u^{2^2} = \sum_{j=0}^{m-1} \gamma_{2,j} e_j$$
$$\vdots$$
$$u^{2^{m-1}} = \sum_{j=0}^{m-1} \gamma_{m-1,j} e_j$$

by repeated squaring.

5. Output

$$\Gamma \leftarrow \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,m-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,m-1} \end{pmatrix}$$

Example: Suppose that B_0 is the Type I optimal normal basis for $GF(2^4)$, and B_1 is the polynomial basis modulo $p_1(t) = t^4 + t + 1$. Then a root of $p_0(u) = u^4 + u^3 + u^2 + u + 1$ is given by $u = t^3 + t^2$. By repeated squaring modulo $p_1(t)$,

$$u = t^3 + t^2$$
$$u^2 = t^3 + t^2 + t + 1$$
$$u^4 = t^3 + t$$
$$u^8 = t^3$$

so that

$$\Gamma := \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Example: Suppose that B_0 is polynomial basis modulo $p_0(u) = u^5 + u^4 + u^2 + u + 1$, and B_1 is the polynomial basis modulo $p_1(t) = t^5 + t^2 + 1$. Then a root of $p_0(u)$ is given by $u = t + 1$. Thus

$$1 = 1$$
$$u = t + 1$$
$$u^2 = t^2 + 1$$
$$u^3 = t^3 + t^2 + t + 1$$
$$u^4 = t^4 + 1,$$

so that

$$\Gamma := \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Example: Suppose that B_0 is polynomial basis modulo $p_0(u) = u^5 + u^2 + 1$, and B_1 is the Type II optimal normal basis for $GF(2^5)$. Then a root of $p_0(u)$ is given by $u = \theta^2 + \theta^4 + \theta^8 + \theta^{16}$. Thus

$$\begin{aligned} 1 &= \theta + \theta^2 + \theta^4 + \theta^8 + \theta^{16} \\ u &= \theta^2 + \theta^4 + \theta^8 + \theta^{16} \\ u^2 &= \theta + \theta^4 + \theta^8 + \theta^{16} \\ u^3 &= \theta + \theta^4 + \theta^{16} \\ u^4 &= \theta + \theta^2 + \theta^8 + \theta^{16}, \end{aligned}$$

so that

$$\Gamma := \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

NOTE—If both B_0 and B_1 are normal bases, then it is sufficient to perform the first computation

$$u = \sum_{j=0}^{m-1} \gamma_{0,j} e_j$$

and omit the rest. This expression provides the top row of the matrix Γ, and subsequent rows are obtained by right cyclic shifts.

Example: Suppose that B_0 is the type 3 Gaussian normal basis for $GF(2^4)$, and B_1 is the Type I optimal normal basis for $GF(2^4)$. Then a root of $p_0(u) = u^4 + u^3 + 1$ is given by $u = \theta + \theta^4 + \theta^8$. Thus

$$\Gamma := \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

A.7.4 Conversion to a Polynomial Basis

If one is representing elements of $GF(2^m)$ with respect to the normal basis B_1, and it is desired to perform division via the Euclidean algorithm (see Annex A.4.4), then it is necessary to construct a change-of-basis matrix from some polynomial basis B_0 to the basis B_1. If another means of division is not available, it is

necessary to construct this matrix using an algorithm other than that of A.7.3 (which requires division in its first step). This can be done as follows.

Input: a field degree m; a normal basis B_1 with field polynomial $p(t)$. (Note: if B_1 is a Gaussian normal basis, then $p(t)$ can be computed via A.7.2.)

Output: the change-of-basis matrix Γ from B_0 to B_1, where B_0 is the polynomial basis with field polynomial $p(t)$.

If B_1 is a Type I optimal normal basis, then the change-of-basis matrix is

$$\Gamma := \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,m-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,m-1} \end{pmatrix}$$

where

$$\gamma_{i,j} = \begin{cases} 1 \text{ if } i = m-1 \\ 1 \text{ if } 2^j \equiv -(i+2) (\mathrm{mod}\, m+1) \\ 0 \text{ otherwise} \end{cases}$$

In the general case, the algorithm of A.7.3 can be used, with Step 1 replaced by

1′. Set $u \leftarrow \theta$ with respect to $B_1 = \{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\}$.

A.8 Bases for Binary Fields: Tables and Algorithms

A.8.1 Basis Table

The following table provides irreducible polynomials modulo 2 that are minimal, in the sense that they have as few terms as possible and that those terms are of the smallest possible degree. More precisely, if an irreducible binary trinomial $t^m + t^k + 1$ exists, then the minimal possible value of k is listed; if no such trinomial exists, then a pentanomial $t^m + t^a + t^b + t^c + 1$ is listed. In the latter case, the value of a is minimal; the value of b is minimal for the given a; and c is minimal for the given a and b. In addition, for each m not divisible by 8 is given the minimal type among Gaussian normal bases for $GF(2^m)$.

The *optimal* normal bases are type 1 and type 2 Gaussian normal bases (see Annex A.3.5). In the cases where both type 1 and type 2 bases exist, they are both listed in the table.

m	Coefficients	GB Type	m	coefficients	GB Type	m	coefficients	GB Type	m	coefficients	GB Type
2	1	1,2	252	15	3	502	8, 5, 4	10	752	13, 10, 3	-
3	1	2	253	46	10	503	3	6	753	158	16
4	1	1	254	7, 2, 1	2	504	15, 14, 6	-	754	19	10
5	2	2	255	52	6	505	156	10	755	12, 10, 1	2
6	1	2	256	10, 5, 2	-	506	23	5	756	45	1

7	1	4	257	12	6	507	13, 6, 3	4	757	7, 6, 1	16
8	4, 3, 1	-	258	71	5	508	9	1	758	233	6
9	1	2	259	10, 6, 2	10	509	8, 7, 3	2	759	98	4
10	3	1	260	15	5	510	69	3	760	11, 6, 5	-
11	2	2	261	7, 6, 4	2	511	10	6	761	3	2
12	3	1	262	9, 8, 4	3	512	8, 5, 2	-	762	83	10
13	4, 3, 1	4	263	93	6	513	26	4	763	16, 14, 9	22
14	5	2	264	9, 6, 2	-	514	67	33	764	6, 5, 3	3
15	1	4	265	42	4	515	14, 7, 4	2	765	9, 7, 4	2
16	5, 3, 1	-	266	47	6	516	21	3	766	18, 10, 2	6
17	3	6	267	8, 6, 3	8	517	12, 10, 2	4	767	168	6
18	3	1,2	268	25	1	518	33	14	768	19, 17, 4	-
19	5, 2, 1	10	269	7, 6, 1	8	519	79	2	769	120	10
20	3	3	270	53	2	520	12, 8, 4	-	770	12, 8, 4	5
21	2	10	271	58	6	521	32	32	771	17, 15, 6	2
22	1	3	272	9, 3, 2	-	522	39	1	772	7	1
23	5	2	273	23	2	523	13, 6, 2	10	773	10, 8, 6	6
24	4, 3, 1	-	274	67	9	524	167	5	774	185	2
25	3	4	275	11, 10, 9	14	525	6, 4, 1	8	775	93	6
26	4, 3, 1	2	276	63	3	526	97	3	776	14, 8, 6	-
27	5, 2, 1	6	277	12, 6, 3	4	527	47	6	777	29	16
28	1	1	278	5	2	528	11, 6, 2	-	778	375	21
29	2	2	279	5	4	529	42	24	779	10, 8, 3	2
30	1	2	280	9, 5, 2	-	530	10, 6, 4	2	780	13	13
31	3	10	281	93	2	531	10, 5, 4	2	781	17, 16, 2	16
32	7, 3, 2	-	282	35	6	532	1	3	782	329	3
33	10	2	283	12, 7, 5	6	533	4, 3, 2	12	783	68	2
34	7	9	284	53	3	534	161	7	784	12, 8, 4	-
35	2	2	285	10, 7, 5	10	535	8, 6, 2	4	785	92	2
36	9	1	286	69	3	536	7, 5, 3	-	786	12, 10, 3	1
37	6, 4, 1	4	287	71	6	537	94	8	787	7, 6, 3	6
38	6, 5, 1	6	288	11, 10, 1	-	538	195	6	788	16, 14, 4	11
39	4	2	289	21	12	539	10, 5, 4	12	789	5, 2, 1	14
40	5, 4, 3	-	290	5, 3, 2	5	540	9	1	790	9, 6, 1	3
41	3	2	291	12, 11, 5	6	541	13, 10, 4	18	791	30	2
42	7	5	292	37	1	542	8, 6, 1	3	792	9, 7, 3	-
43	6, 4, 3	4	293	11, 6, 1	2	543	16	2	793	253	6
44	5	9	294	33	3	544	8, 3, 1	-	794	143	14
45	4, 3, 1	4	295	48	16	545	122	2	795	7, 4, 1	10
46	1	3	296	7, 3, 2	-	546	8, 2, 1	1	796	9, 4, 1	1
47	5	6	297	5	6	547	13, 7, 4	10	797	12, 10, 4	6
48	5, 3, 2	-	298	11, 8, 4	6	548	10, 5, 3	5	798	53	6
49	9	4	299	11, 6, 4	2	549	16, 4, 3	14	799	25	22
50	4, 3, 2	2	300	5	19	550	193	7	800	9, 7, 1	-
51	6, 3, 1	2	301	9, 5, 2	10	551	135	6	801	217	12
52	3	1	302	41	3	552	12, 6, 2	-	802	15, 13, 9	6
53	6, 2, 1	2	303	1	2	553	39	4	803	14, 9, 2	2
54	9	3	304	11, 2, 1	-	554	10, 8, 7	2	804	75	5
55	7	12	305	102	6	555	10, 9, 4	4	805	8, 7, 2	6
56	6, 4, 2	-	306	7, 3, 1	2	556	153	1	806	21	11
57	4	10	307	8, 4, 2	4	557	7, 6, 5	6	807	7	14
58	19	1	308	15	15	558	73	2	808	14, 3, 2	-
59	7, 4, 2	12	309	10, 6, 4	2	559	34	4	809	15	2

60	1	1	310	93	6	560	11, 9, 6	-	810	159	2
61	5, 2, 1	6	311	7, 5, 3	6	561	71	2	811	12, 10, 8	10
62	29	6	312	9, 7, 4	-	562	11, 4, 2	1	812	29	3
63	1	6	313	79	6	563	14, 7, 3	14	813	10, 3, 1	4
64	4, 3, 1	-	314	15	5	564	163	3	814	21	15
65	18	2	315	10, 9, 1	8	565	11, 6, 1	10	815	333	8
66	3	1	316	63	1	566	153	3	816	10, 6, 4	-
67	5, 2, 1	4	317	7, 4, 2	26	567	28	4	817	52	6
68	9	9	318	45	11	568	15, 7, 6	-	818	119	2
69	6, 5, 2	2	319	36	4	569	77	12	819	16, 9, 7	20
70	5, 3, 1	3	320	4, 3, 1	-	570	67	5	820	123	1
71	6	8	321	31	12	571	10, 5, 2	10	821	15, 11, 2	8
72	10, 8, 4	-	322	67	6	572	12, 8, 1	5	822	17	3
73	25	4	323	10, 3, 1	2	573	10, 6, 4	4	823	9	10
74	35	2	324	51	5	574	13	3	824	11, 6, 4	-
75	6, 3, 1	10	325	10, 5, 2	4	575	146	2	825	38	6
76	21	3	326	10, 3, 1	2	576	13, 4, 3	-	826	255	1
77	6, 5, 2	6	327	34	8	577	25	4	827	12, 10, 7	14
78	6, 5, 3	7	328	8, 3, 1	-	578	23, 22, 16	6	828	189	1
79	9	4	329	50	2	579	12, 9, 7	10	829	4, 3, 1	10
80	9, 4, 2	-	330	99	2	580	237	3	830	17, 10, 7	14
81	4	2	331	10, 6, 2	6	581	13, 7, 6	8	831	49	2
82	6, 4, 2	1	332	89	3	582	85	3	832	13, 5, 2	-
83	7, 4, 2	2	333	2	24	583	130	4	833	149	2
84	5	5	334	5, 2, 1	7	584	14, 6, 2	-	834	15	2
85	8, 2, 1	12	335	10, 7, 2	12	585	88	2	835	14, 7, 5	6
86	21	2	336	7, 4, 1	-	586	7, 5, 2	1	836	10, 9, 2	15
87	13	4	337	55	10	587	11, 6, 1	14	837	8, 6, 5	6
88	7, 6, 2	-	338	4, 3, 1	2	588	35	11	838	61	7
89	38	2	339	16, 10, 7	8	589	10, 4, 3	4	839	54	12
90	27	2	340	45	3	590	93	11	840	11, 5, 1	-
91	8, 5, 1	6	341	10, 8, 6	8	591	9, 6, 4	6	841	144	12
92	21	3	342	125	6	592	13, 6, 3	-	842	47	5
93	2	4	343	75	4	593	86	2	843	11, 10, 7	6
94	21	3	344	7, 2, 1	-	594	19	17	844	105	13
95	11	2	345	22	4	595	9, 2, 1	6	845	2	8
96	10, 6, 4	-	346	63	1	596	273	3	846	105	2
97	6	4	347	11, 10, 3	6	597	14, 12, 9	4	847	136	30
98	11	2	348	103	1	598	7, 6, 1	15	848	11, 4, 1	-
99	6, 3, 1	2	349	6, 5, 2	10	599	30	8	849	253	8
100	15	1	350	53	2	600	9, 5, 2	-	850	111	6
101	7, 6, 1	6	351	34	10	601	201	6	851	13, 10, 5	6
102	29	6	352	13, 11, 6	-	602	215	5	852	159	1
103	9	6	353	69	14	603	6, 4, 3	12	853	10, 7, 1	4
104	4, 3, 1	-	354	99	2	604	105	7	854	7, 5, 3	18
105	4	2	355	6, 5, 1	6	605	10, 7, 5	6	855	29	8
106	15	1	356	10, 9, 7	3	606	165	2	856	19, 10, 3	-
107	9, 7, 4	6	357	11, 10, 2	10	607	105	6	857	119	8
108	17	5	358	57	10	608	19, 13, 6	-	858	207	1
109	5, 4, 2	10	359	68	2	609	31	4	859	17, 15, 4	22
110	33	6	360	5, 3, 2	-	610	127	10	860	35	9
111	10	20	361	7, 4, 1	30	611	10, 4, 2	2	861	14	28
112	5, 4, 3	-	362	63	5	612	81	1	862	349	31

113	9	2	363	8, 5, 3	4	613	19, 10, 4	10	863	6, 3, 2	6
114	5, 3, 2	5	364	9	3	614	45	2	864	21, 10, 6	-
115	8, 7, 5	4	365	9, 6, 5	24	615	211	2	865	1	4
116	4, 2, 1	3	366	29	22	616	19, 10, 3	-	866	75	2
117	5, 2, 1	8	367	21	6	617	200	8	867	9, 5, 2	4
118	33	6	368	7, 3, 2	-	618	295	1,2	868	145	19
119	8	2	369	91	10	619	9, 8, 5	4	869	11, 7, 6	12
120	4, 3, 1	-	370	139	6	620	9	3	870	301	2
121	18	6	371	8, 3, 2	2	621	12, 6, 5	6	871	378	6
122	6, 2, 1	6	372	111	1	622	297	3	872	12, 10, 8	-
123	2	10	373	8, 7, 2	4	623	68	12	873	352	2
124	19	3	374	8, 6, 5	3	624	11, 6, 5	-	874	12, 4, 2	9
125	7, 6, 5	6	375	16	2	625	133	36	875	12, 8, 1	12
126	21	3	376	8, 7, 5	-	626	251	21	876	149	1
127	1	4	377	41	14	627	13, 8, 4	20	877	6, 5, 4	16
128	7, 2, 1	-	378	43	1,2	628	223	7	878	12, 9, 8	15
129	5	8	379	10, 8, 5	12	629	6, 5, 2	2	879	11	2
130	3	1	380	47	5	630	7, 4, 2	14	880	8, 6, 2	-
131	8, 3, 2	2	381	5, 2, 1	8	631	307	10	881	78	18
132	17	5	382	81	6	632	9, 2, 1	-	882	99	1
133	9, 8, 2	12	383	90	12	633	101	34	883	17, 16, 12	4
134	57	2	384	12, 3, 2	-	634	39	13	884	173	27
135	11	2	385	6	6	635	14, 10, 4	8	885	8, 7, 1	28
136	5, 3, 2	-	386	83	2	636	217	13	886	13, 9, 8	3
137	21	6	387	8, 7, 1	4	637	14, 9, 1	4	887	147	6
138	8, 7, 1	1	388	159	1	638	6, 5, 1	2	888	19, 18, 10	-
139	8, 5, 3	4	389	10, 9, 5	24	639	16	2	889	127	4
140	15	3	390	9	3	640	8, 6, 2	-	890	183	5
141	10, 4, 1	8	391	28	6	641	11	2	891	12, 4, 1	2
142	21	6	392	6, 4, 2	-	642	119	6	892	31	3
143	5, 3, 2	6	393	7	2	643	11, 3, 2	12	893	11, 8, 6	2
144	7, 4, 2	-	394	135	9	644	11, 6, 5	3	894	173	3
145	52	10	395	11, 6, 5	6	645	11, 8, 4	2	895	12	4
146	71	2	396	25	11	646	249	6	896	7, 5, 3	-
147	14	6	397	12, 7, 6	6	647	5	14	897	113	8
148	27	1	398	7, 6, 2	2	648	8, 4, 2	-	898	207	21
149	10, 9, 7	8	399	26	12	649	37	10	899	18, 15, 5	8
150	53	19	400	5, 3, 2	-	650	3	2	900	1	11
151	3	6	401	152	8	651	14	2	901	13, 7, 6	6
152	6, 3, 2	-	402	171	5	652	93	1	902	21	3
153	1	4	403	9, 8, 5	16	653	10, 8, 7	2	903	35	4
154	15	25	404	65	3	654	33	14	904	12, 7, 2	-
155	62	2	405	13, 8, 2	4	655	88	4	905	117	6
156	9	13	406	141	6	656	7, 5, 4	-	906	123	1
157	6, 5, 2	10	407	71	8	657	38	10	907	12, 10, 2	6
158	8, 6, 4	2	408	5, 3, 2	-	658	55	1	908	143	21
159	31	22	409	87	4	659	15, 4, 2	2	909	14, 4, 1	4
160	5, 3, 2	-	410	10, 4, 3	2	660	11	1	910	15, 9, 7	18
161	18	6	411	12, 10, 3	2	661	12, 11, 4	6	911	204	2
162	27	1	412	147	3	662	21	3	912	7, 5, 1	-
163	7, 6, 3	4	413	10, 7, 6	2	663	107	14	913	91	6
164	10, 8, 7	5	414	13	2	664	11, 9, 8	-	914	4, 2, 1	18
165	9, 8, 3	4	415	102	28	665	33	14	915	8, 6, 3	10

166	37	3	416	9, 5, 2	-	666	10, 7, 2	22	916	183	3
167	6	14	417	107	4	667	18, 7, 3	6	917	12, 10, 7	6
168	10, 8, 2	-	418	199	1	668	147	11	918	77	10
169	34	4	419	15, 5, 4	2	669	5, 4, 2	4	919	36	4
170	11	6	420	7	1	670	153	6	920	14, 9, 6	-
171	6, 5, 2	12	421	5, 4, 2	10	671	15	6	921	221	6
172	1	1	422	149	11	672	11, 6, 5	-	922	7, 6, 5	10
173	8, 5, 2	2	423	25	4	673	28	4	923	16, 14, 13	2
174	13	2	424	9, 7, 2	-	674	11, 7, 4	5	924	31	5
175	6	4	425	12	6	675	6, 3, 1	22	925	16, 15, 7	4
176	11, 3, 2	-	426	63	2	676	31	1	926	365	6
177	8	4	427	11, 6, 5	16	677	8, 4, 3	8	927	403	4
178	31	1	428	105	5	678	15, 5, 3	10	928	10, 3, 2	-
179	4, 2, 1	2	429	10, 8, 7	2	679	66	10	929	11, 4, 3	8
180	3	1	430	12, 10, 6	3	680	22, 8, 6	-	930	31	2
181	7, 6, 1	6	431	120	2	681	11, 9, 3	22	931	10, 9, 4	10
182	81	3	432	13, 4, 3	-	682	171	6	932	177	3
183	56	2	433	33	4	683	11, 6, 1	2	933	16, 6, 1	2
184	9, 8, 7	-	434	12, 10, 8	9	684	209	3	934	22, 6, 5	3
185	24	8	435	12, 9, 5	4	685	4, 3, 1	4	935	417	2
186	11	2	436	165	13	686	197	2	936	15, 13, 12	-
187	7, 6, 5	6	437	6, 2, 1	18	687	13	10	937	217	6
188	6, 5, 2	5	438	65	2	688	14, 4, 2	-	938	207	2
189	6, 5, 2	2	439	49	10	689	14	12	939	7, 5, 4	2
190	8, 7, 6	10	440	4, 3, 1	-	690	79	2	940	10, 7, 1	1
191	9	2	441	7	2	691	13, 6, 2	10	941	11, 6, 1	6
192	7, 2, 1	-	442	7, 5, 2	1	692	299	5	942	45	10
193	15	4	443	10, 6, 1	2	693	15, 8, 2	6	943	24	6
194	87	2	444	81	5	694	169	3	944	12, 11, 9	-
195	8, 3, 2	6	445	7, 6, 4	6	695	177	18	945	77	8
196	3	1	446	105	6	696	16, 14, 8	-	946	16, 12, 6	1
197	9, 4, 2	18	447	73	6	697	267	4	947	9, 6, 5	6
198	9	22	448	11, 6, 4	-	698	215	5	948	189	7
199	34	4	449	134	8	699	15, 10, 1	4	949	8, 3, 2	4
200	5, 3, 2	-	450	47	13	700	75	1	950	13, 12, 10	2
201	14	8	451	16, 10, 1	6	701	16, 4, 2	18	951	260	16
202	55	6	452	6, 5, 4	11	702	37	14	952	16, 9, 7	-
203	8, 7, 1	12	453	15, 6, 4	2	703	12, 7, 1	6	953	168	2
204	27	3	454	8, 6, 1	19	704	8, 3, 2	-	954	131	49
205	9, 5, 2	4	455	38	26	705	17	6	955	7, 6, 3	10
206	10, 9, 5	3	456	16, 4, 2	-	706	12, 11, 8	21	956	305	15
207	43	4	457	16	30	707	15, 8, 5	6	957	9, 6, 5	6
208	8, 6, 2	-	458	203	6	708	15	1	958	13, 9, 4	6
209	6	2	459	12, 5, 2	8	709	4, 3, 1	4	959	143	8
210	7	1,2	460	19	1	710	12, 10, 2	3	960	12, 9, 3	-
211	11, 10, 8	10	461	7, 6, 1	6	711	92	8	961	18	16
212	105	5	462	73	10	712	5, 4, 3	-	962	15, 8, 5	14
213	6, 5, 2	4	463	93	12	713	41	2	963	20, 9, 6	4
214	73	3	464	16, 8, 4	-	714	23	5	964	103	9
215	23	6	465	31	4	715	7, 4, 1	4	965	15, 4, 2	2
216	7, 3, 1	-	466	14, 11, 6	1	716	183	5	966	201	7
217	45	6	467	11, 6, 1	6	717	16, 7, 1	18	967	36	16
218	11	5	468	27	21	718	165	15	968	9, 5, 2	-

219	8, 4, 1	4	469	9, 5, 2	4	719	150	2	969	31	4
220	7	3	470	9	2	720	9, 6, 4	-	970	11, 7, 2	9
221	8, 6, 2	2	471	1	8	721	9	6	971	6, 2, 1	6
222	5, 4, 2	10	472	10, 8, 2	-	722	231	26	972	7	5
223	33	12	473	200	2	723	16, 10, 4	2	973	13, 6, 4	6
224	9, 8, 3	-	474	191	5	724	207	13	974	9, 8, 7	2
225	32	22	475	9, 8, 4	4	725	9, 6, 5	2	975	19	2
226	10, 6, 4	1	476	9	5	726	5	2	976	8, 6, 2	-
227	10, 9, 4	24	477	16, 15, 7	46	727	180	4	977	15	8
228	113	9	478	121	7	728	4, 3, 2	-	978	9, 3, 1	6
229	10, 4, 1	12	479	104	8	729	58	24	979	178	4
230	8, 7, 6	2	480	15, 9, 6	-	730	147	13	980	8, 7, 6	9
231	26	2	481	138	6	731	8, 6, 2	8	981	12, 6, 5	32
232	8, 4, 2	-	482	9, 6, 5	5	732	343	11	982	177	15
233	74	2	483	9, 6, 4	2	733	8, 7, 2	10	983	230	14
234	31	5	484	105	3	734	11, 6, 1	3	984	12, 8, 2	-
235	9, 6, 1	4	485	17, 16, 6	18	735	44	8	985	222	10
236	5	3	486	81	10	736	13, 8, 6	-	986	3	2
237	7, 4, 1	10	487	94	4	737	5	6	987	16, 13, 12	6
238	73	7	488	4, 3, 1	-	738	347	5	988	121	7
239	36	2	489	83	12	739	18, 16, 8	4	989	10, 4, 2	2
240	8, 5, 3	-	490	219	1	740	135	3	990	161	10
241	70	6	491	11, 6, 3	2	741	9, 8, 3	2	991	39	18
242	95	6	492	7	13	742	85	15	992	17, 15, 13	-
243	8, 5, 1	2	493	10, 5, 3	4	743	90	2	993	62	2
244	111	3	494	17	3	744	13, 11, 1	-	994	223	10
245	6, 4, 1	2	495	76	2	745	258	10	995	15, 12, 2	14
246	11, 2, 1	11	496	16, 5, 2	-	746	351	2	996	65	43
247	82	6	497	78	20	747	10, 6, 4	6	997	12, 6, 3	4
248	14, 12, 10	-	498	155	9	748	19	7	998	101	2
249	35	8	499	11, 6, 5	4	749	7, 6, 1	2	999	59	8
250	103	9	500	27	11	750	309	14	1000	5, 4, 3	-
251	7, 4, 2	2	501	5, 4, 2	10	751	18	6			

A.8.2 Random Search for Other Irreducible Polynomials

The basic method for producing an irreducible polynomial modulo 2 of degree m is as follows: generate a random polynomial (perhaps subject to certain conditions, such as number of terms), test for irreducibility (via A.5.5 with $r = 1$), and repeat until an irreducible is found. The polynomial $f(t)$ should satisfy the following three conditions, since these are necessary for irreducibility modulo 2.

i) $f(t)$ must have an odd number of (nonzero) terms.
ii) The constant term of $f(t)$ must be 1.
iii) There must be at least one odd-degree term.

If a random $f(t)$ of degree at most m satisfies these conditions, then the probability that $f(t)$ is irreducible is roughly $4 / m$. Thus one can expect to find an irreducible after $\approx m / 4$ random tries.

A.8.3 Irreducibles from Other Irreducibles

If $f(t)$ is an irreducible of degree m, then so are the following five polynomials.

$$g(t) = t^m f(1/t)$$
$$h(t) = g(t+1)$$
$$j(t) = t^m h(1/t)$$
$$k(t) = j(t+1)$$
$$l(t) = t^m k(1/t) = f(t+1)$$

Example:

$$f(t) = t^5 + t^2 + 1$$
$$g(t) = t^5 + t^3 + 1$$
$$h(t) = t^5 + t^4 + t^3 + t^2 + 1$$
$$j(t) = t^5 + t^3 + t^2 + t + 1$$
$$k(t) = t^5 + t^4 + t^3 + t + 1$$
$$l(t) = t^5 + t^4 + t^2 + t + 1$$

Another construction is as follows. Suppose that $f(t)$ is an irreducible of odd degree m. Define the polynomials $h_0(t)$, $h_1(t)$, $h_2(t)$ by

$$f(t) = h_0(t^3) + t\,h_1(t^3) + t^2 h_2(t^3).$$

Then

$$g(t) = h_0(t)^3 + t\,h_1(t)^3 + t^2 h_2(t)^3 + t\,h_0(t)\,h_1(t)\,h_2(t)$$

is an irreducible of degree m.

More generally, if r is an odd number relatively prime to $2^m - 1$, then define $h_0(t)$, …, $h_{r-1}(t)$ by

$$f(t) = \sum_{k=0}^{r-1} t^k h_k(t^r).$$

Define the functions

$$w_k(t) = t^{k/r} h_k(t).$$

Let M be the r-by-r matrix with entries

$$M_{i,j} = w_{i-j}(t)$$

where the subscripts of w are taken modulo r. Then the determinant of M is an irreducible of degree m.

A.8.4 Irreducibles of Even Degree

If $m = 2d$, then one can produce an irreducible of degree m more efficiently by generating an irreducible of degree d and using the following *degree-doubling* technique.

If

$$p(t) = t^d + t^{d-1} + 1 + \sum_i t^{k_i}$$

is irreducible, then so is

$$P(t) = t^{2d} + t^{d} + t^{2d-2} + t^{d-1} + 1 + \sum_{i} (t^{2k_i} + t^{k_i}).$$

Note that the sum over i is allowed to be empty.

Examples:

$$p(t) = t^3 + t^2 + 1,$$
$$P(t) = t^6 + t^4 + t^3 + t^2 + 1.$$
$$p(t) = t^5 + t^4 + t^3 + t + 1,$$
$$P(t) = t^{10} + t^8 + t^6 + t^5 + t^4 + t^3 + t^2 + t + 1.$$
$$p(t) = t^5 + t^4 + t^2 + t + 1,$$
$$P(t) = t^{10} + t^8 + t^5 + t + 1.$$

If an irreducible $f(t)$ is not of the form required for degree doubling, then it may be used to construct one that is suitable via the methods of A.8.3. In particular, if $m = 2d$ with d odd, then either $f(t)$ or $f(t + 1)$ is of the suitable form.

Input: an irreducible $f(t)$ of odd degree d.

Output: an irreducible $P(t)$ of degree $2d$.

1. If $f(t)$ has a degree $d - 1$ term then set $p(t) \leftarrow f(t)$; else set $p(t) \leftarrow f(t + 1)$
2. Apply degree doubling to $p(t)$ and output the result.

The degree doubling technique can often be applied more than once. The following algorithm treats a family of cases in which this is possible.

Input: an irreducible $f(t)$ of odd degree d satisfying at least one of the following three conditions:

i) $f(t)$ contains both degree 1 and degree $d - 1$ terms.
ii) $f(t)$ contains a degree 1 term and contains an even number of odd-degree terms in all.
iii) $f(t)$ contains a degree $d - 1$ term and contains an odd number of odd-degree terms in all.

Output: an irreducible of degree $4d$.

1. If $f(t)$ satisfies (iii) then set $p(t) \leftarrow f(t)$; else set $p(t) \leftarrow t^d f(1 / t)$
2. Apply degree doubling to $p(t)$ to produce an irreducible $P(t)$ of degree $2d$.
3. Set $g(t) \leftarrow P(t + 1)$.
4. Set $h(t) \leftarrow t^{2d} g(1 / t)$.
5. Apply degree doubling to $h(t)$ and output the result.

A.8.5 Irreducible Trinomials

A search for irreducible trinomials should take into account the following restrictions on which trinomials can be irreducible modulo 2.

Suppose the trinomial to be checked is $f(t) = t^m + t^k + 1$, where $m > k > 0$. (Note that, since $f(t)$ is irreducible if and only if $t^m + t^{m-k} + 1$ is, then only half the possible k need to be checked. For instance, it is sufficient to check for $k \le m / 2$.) Then $f(t)$ is reducible in all of the following cases.

1. m and k are both even.
2. m is a multiple of 8.
3. $m = hn$, and $k = 2h$ or $(n – 2)h$, for some odd number n, h such that $n \equiv \pm h \pmod 8$.
4. $k = 2h$ or $m – 2h$ for some h not dividing m, and $m \equiv \pm 3 \pmod 8$.
5. $m = 2n$ where n is odd and $n \equiv k \pmod 4$, *except* for the polynomials $t^{2k} + t^k + 1$ with k a power of 3. (All of these polynomials are *irreducible*.)

A.9 Elliptic Curves: Overview

A.9.1 Introduction

A *plane curve* is defined to be the set of points satisfying an equation $F(x, y) = 0$. The simplest plane curves are lines (whose defining equation has degree 1 in x and y) and conic sections (degree 2 in x and y). The next simplest are the cubic curves (degree 3). Cubic plane curves are called *elliptic curves*, because they arose historically from the problem of computing the circumference of an ellipse.

In cryptography, the elliptic curves of interest are those defined over finite fields. That is, the coefficients of the defining equation $F(x, y) = 0$ are elements of $GF(q)$, and the points on the curve are of the form $P = (x, y)$, where x and y are elements of $GF(q)$. Examples are given below.

The Weierstrass equation.

There are several kinds of defining equations for elliptic curves, but the most common are the *Weierstrass equations.*

— For the prime finite fields $GF(p)$ with $p > 3$, the Weierstrass equation is

$$y^2 = x^3 + ax + b$$

where a and b are integers modulo p for which $4a^3 + 27b^2 \not\equiv 0 \pmod p$.

— For the binary finite fields $GF(2^m)$, the Weierstrass equation is

$$y^2 + xy = x^3 + ax^2 + b$$

where a and b are elements of $GF(2^m)$ with $b \neq 0$.

NOTE—there is another kind of Weierstrass equation over $GF(2^m)$, giving what are called *supersingular* curves. However, these curves are cryptographically weak (see [MOV93]); thus they are omitted from this Standard.

Given a Weierstrass equation, the elliptic curve E consists of the solutions (x, y) over $GF(q)$ to the defining equation, along with an additional element called the *point at infinity* (denoted $\mathcal{O}$). The points other than $\mathcal{O}$ are called *finite* points. The number of points on E (including $\mathcal{O}$) is called the *order* of E and is denoted by $\#E(GF(q))$.

Example: Let E be the curve

$$y^2 = x^3 + 10x + 5$$

over the field $GF(13)$. Then the points on E are

$$\{\mathcal{O}, (1,4), (1,9), (3,6), (3,7), (8,5), (8,8), (10,0), (11,4), (11,9)\}.$$

Thus the order of E is $\#E$ (GF (13)) = 10.

Example: Let E be the curve

$$y^2 + xy = x^3 + (t + 1)\,x^2 + 1$$

over the field GF (2^3) given by the polynomial basis with field polynomial $t^3 + t + 1 = 0$. Then the points on E are

$$\{\mathcal{O}, ((000), (001)),$$
$$((010), (100)), ((010), (110)), ((011), (100)), ((011), (111)),$$
$$((100), (001)), ((100), (101)), ((101), (010)), ((101), (111)),$$
$$((110), (000)), ((110), (110)), ((111), (001)), ((111), (110))\}.$$

Thus the order of E is $\#E$ (GF (2^3)) = 14.

For more information on elliptic curve cryptography, see [Men93b].

A.9.2 Operations on Elliptic Curves

There is an *addition operation* on the points of an elliptic curve which possesses the algebraic properties of ordinary addition (e.g. commutativity and associativity). This operation can be described geometrically as follows.

Define the *inverse* of the point $P = (x, y)$ to be

$$-P = \begin{cases} (x, -y) \text{ if } q = p \text{ prime}, \\ (x, x + y) \text{ if } q = 2^m. \end{cases}$$

Then the *sum* $P + Q$ of the points P and Q is the point R with the property that P, Q, and $-R$ lie on a common line.

The point at infinity.

The point at infinity $\mathcal{O}$ plays a role analogous to that of the number 0 in ordinary addition. Thus

$$P + \mathcal{O} = P,$$
$$P + (-P) = \mathcal{O}$$

for all points P.

Full addition.

When implementing the formulae for elliptic curve addition, it is necessary to distinguish between *doubling* (adding a point to itself) and *adding* two distinct points that are not inverses of each other, because the formulae are different in the two cases. Besides this, there are also the special cases involving $\mathcal{O}$. By *full addition* is meant choosing and implementing the appropriate formula for the given pair of points. Algorithms for full addition are given in A.10.1, A.10.2 and A.10.8.

Scalar multiplication.

Elliptic curve points can be added but not *multiplied*. It is, however, possible to perform *scalar multiplication*, which is another name for repeated addition of the same point. If n is a positive integer and P a point on an elliptic curve, the scalar multiple nP is the result of adding n copies of P. Thus, for example, $5P = P + P + P + P + P$.

The notion of scalar multiplication can be extended to zero and the negative integers via

$$0P = \mathcal{O}, \qquad (-n)\,P = n\,(-P).$$

A.9.3 Elliptic Curve Cryptography

Orders.

The *order* of a point P on an elliptic curve is the smallest positive integer r such that $rP = \mathcal{O}$. The order always exists and divides the order of the curve $\#E(GF(q))$. If k and l are integers, then $kP = lP$ if and only if $k \equiv l \pmod{r}$.

Elliptic curve discrete logarithms.

Suppose that the point G on E has prime order r where r^2 does not divide the order of the curve $\#E(GF(q))$. Then a point P satisfies $P = lG$ for some l if and only if $rP = \mathcal{O}$. The coefficient l is called the *elliptic curve discrete logarithm* of P (with respect to the base point G). The elliptic curve discrete logarithm is an integer modulo r.

EC-based cryptography.

Suppose that the base point G on E has order r as described in the preceding paragraph. Then a key pair can the defined as follows.

— The private key s is an integer modulo r.
— The corresponding public key W is a point on E defined by $W := sG$.

It is necessary to compute an elliptic curve discrete logarithm in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the elliptic curve discrete logarithm problem. Thus it is an example of *EC-based cryptography*. The difficulty of the elliptic curve discrete logarithm problem is discussed in Annex D.4.2.

A.9.4 Analogies with DL

The discrete logarithm problem in finite fields $GF(q)^*$ and the elliptic curve discrete logarithm are in some sense the same abstract problem in two different settings. As a result, the primitives and schemes of DL and EC based cryptography are closely analogous to each other. The following table makes these analogies explicit.

	DL	EC
Setting	$GF(q)^*$	curve E over $GF(q)$
Basic operation	multiplication in $GF(q)$	addition of points
Main operation	exponentiation	scalar multiplication
Base element	generator g	base point G

Base element order	prime r	prime r
Private key	s (integer modulo r)	s (integer modulo r)
Public key	w (element of $GF(q)$)	W (point on E)

A.9.5 Curve Orders

The most difficult part of generating EC parameters is finding a base point of prime order. Generating such a point requires knowledge of the curve order $n = \#E(GF(q))$. Since r must divide n, one has the following problem: *given a field* $\mathbf{F} = GF(q)$, *find an elliptic curve defined over* $\mathbf{F}$ *whose order is divisible by a sufficiently large prime* r. (Note that "sufficiently large" is defined in terms of the desired security; see Annex A.9.3 and D.4.2.) This section discusses this problem.

Basic facts.

— If n is the order of an elliptic curve over $GF(q)$, then the *Hasse bound* is

$$q - 2\sqrt{q} + 1 \le n \le q + 2\sqrt{q} + 1.$$

Thus the order of an elliptic curve over $GF(q)$ is approximately q.

— If q is a prime p, let n be the order of the curve $y^2 = x^3 + ax + b$, where a and b are both nonzero. Then if $\lambda \neq 0$, the order of the curve $y^2 = x^3 + a\lambda^2 x + b\lambda^3$ is n if λ is a square modulo p and $2p + 2 - n$ otherwise. (This fact allows one to replace a given curve by one with the same order and satisfying some extra condition, such as $a = p - 3$ which will be used in A.10.4.) In the case $b = 0$, there are four possible orders; in the case $a = 0$, there are six. (The formulae for these orders can be found in Step 6 of A.14.2.3.)

— If $q = 2^m$, let n be the order of the curve $y^2 + xy = x^3 + ax^2 + b$, where a and b are both nonzero. Then if $\lambda \neq 0$, the order of the curve $y^2 + xy = x^3 + (a + \lambda)x^2 + b$ is n if λ has trace 0 and $2^{m+1} + 2 - n$ otherwise (see Annex A.4.5). (This fact allows one to replace a given curve by one with the same order and satisfying some extra condition, such as $a = 0$ which will be used in A.10.7.)

— If $q = 2^m$, then the curves $y^2 + xy = x^3 + ax^2 + b$ and $y^2 + xy = x^3 + a^2x^2 + b^2$ have the same order.

Near primality.

Given a trial division bound l_{max}, we say that the positive integer k is *smooth* if every prime divisor of k is at most l_{max}. Given large positive integers r_{min} and r_{max}, we say that u is *nearly prime* if $u = kr$ for some prime r in the interval $r_{min} \le r \le r_{max}$ and some smooth integer k. (The requirement that k be smooth is omitted in most definitions of near primality. It is included here to guarantee that there exists an efficient algorithm to check for near primality.) In the case in which a prime order curve is desired, the bound l_{max} is set to 1.

NOTE—since all elliptic curves over $GF(q)$ have order at most $u_{max} = q + 2\sqrt{q} + 1$, then r_{max} should be no greater than u_{max}. (If no maximum is desired, e.g., as in ANSI X9.62 [ANS98e], then one takes $r_{max} \leftarrow u_{max}$.) Moreover, if r_{min} is close to u_{max}, then there will be a small number of possible curves to choose from, so that finding a suitable one will be more difficult. If a prime-order curve is desired, a convenient choice is $r_{min} = q + \sqrt{q}$.

Finding curves of appropriate order.

Given a field size q and lower and upper bounds r_{min} and r_{max} for base point order, we wish to find an elliptic curve E over $GF(q)$ and a prime r in the interval $r_{min} \le r \le r_{max}$ which is the order of a point on E.

Since we cannot factor large numbers in general, we must choose a trial division bound l_{max} and restrict our search to nearly prime curve orders (see Annex A.15.3). There are four approaches to selecting such a curve:

1. Select a curve at random, compute its order directly, and repeat the process until an appropriate order is found.
2. Select curve coefficients with particular desired properties, compute the curve order directly, and repeat the process until an appropriate order is found.
3. If $q = 2^m$ where m is divisible by a "small" integer d, then select a curve defined over $GF(2^d)$ and compute its order (over $GF(2^m)$) via A.11.6. Repeat if possible until an appropriate order is found.
4. Search for an appropriate order, and construct a curve of that order.

The first approach is described in A.12.4 and A.12.6 (except for the point-counting algorithm, which is omitted). The second approach is not described since its details depend on the particular desired properties. The third approach is given in A.11.4–A.11.6. The fourth approach is implemented using the *complex multiplication* (or *CM*) method, given in A.14.

A.9.6 Representation of Points

This section discusses the issues involved in choosing representations for points on elliptic curves, for purposes of internal computation and for external communication.

Affine coordinates.

A finite point on E is specified by two elements x, y in $GF(q)$ satisfying the defining equation for E. These are called the *affine coordinates* for the point. The point at infinity $\mathcal{O}$ has no affine coordinates. For purposes of internal computation, it is most convenient to represent $\mathcal{O}$ by a pair of coordinates (x, y) not on E. For $q = 2^m$, the simplest choice is $\mathcal{O} = (0,0)$. For $q = p$, one chooses $\mathcal{O} = (0, 0)$ unless $b = 0$, in which case $\mathcal{O} = (0, 1)$.

Coordinate compression.

The affine coordinates of a point require $2m$ bits to store and transmit, where q is either 2^m or an m-bit prime. This is far more than is needed, however. For purposes of external communication, therefore, it can be advantageous to *compress* one or both of the coordinates.

The y coordinate can always be compressed. The compressed y coordinate, denoted $\tilde{y}$, is a single bit, defined as follows.

— if q is an odd prime, then $\tilde{y} := y \bmod 2$, where y is interpreted as a positive integer less than q. Put another way, $\tilde{y}$ is the rightmost bit of y.

— if q is a power of 2, then $\tilde{y}$ is the rightmost bit of the field element $y\,x^{-1}$ (except when $x = 0$, in which case $\tilde{y} := 0$).

Compression of x coordinates takes place only when $q = 2^m$. Moreover, the x coordinate of a point can be compressed if and only if the point is twice another point. In particular, every point of prime order can be compressed. Therefore, x coordinate compression can be used in all cryptographic algorithms specified in this Standard that are based on an elliptic curve over a binary field.

The compressed x coordinate, denoted $\tilde{x}$, is a bit string one bit shorter than the bit string representing x.

The string $\tilde{x}$ is derived from x by dropping a certain bit from the bit string representing x. If **F** is given by a normal basis, or if m is odd, then the rightmost bit is dropped. If **F** is given by a polynomial basis

$$\{t^{m-1}, \ldots, t, 1\}$$

and m is even, then one drops the rightmost bit corresponding to a basis element of trace 1. In other words, one drops the bit in the location of the rightmost 1 in the vector τ defined in A.4.5.

Examples:

i) If $m = 5$ and $x = (11001)$, then $\tilde{x} = (1100)$.
ii) If **F** is given by the field polynomial $t^6 + t^5 + 1$, then

$$\tau = (111110),$$

so that the compressed form of $x = (a_5\ a_4\ a_3\ a_2\ a_1\ a_0)$ is $\tilde{x} = (a_5\ a_4\ a_3\ a_2\ a_0)$.

The choice of dropped bit depends only on the representation of the field, not on the choice of point.

NOTES

1—Algorithms for decompressing coordinates are given in A.12.8 through A.12.10.

2—There are many other possible ways to compress coordinates; the methods given here are the ones that have appeared in the literature (see [Men95], [Ser98]).

3—It is a common usage (which we adopt within this Standard) to say that the point P is compressed if its y coordinate is compressed but its x coordinate is not. (See E.2.3.1.)

Projective coordinates.

If division within $GF(q)$ is relatively expensive, then it may pay to keep track of numerators and denominators separately. In this way, one can replace division by α with multiplication of the denominator by α. This is accomplished by the *projective coordinates* X, Y, and Z, given by

$$x = \frac{X}{Z^2}, y = \frac{Y}{Z^3}.$$

The projective coordinates of a point are not unique because

$$(X, Y, Z) = (\lambda^2 X, \lambda^3 Y, \lambda Z)$$

for every nonzero $\lambda \in GF(q)$.

The projective coordinates of the point at infinity are $(\lambda^2, \lambda^3, 0)$, where $\lambda \neq 0$.

Other kinds of projective coordinates exist, but the ones given here provide the fastest arithmetic on elliptic curves. (See [CC87].)

The formulae above provide the method for converting a finite point from projective coordinates to affine. To convert from affine to projective, one proceeds as follows.

$$X \leftarrow x, Y \leftarrow y, Z \leftarrow 1.$$

Projective coordinates are well suited for internal computation, but not for external communication since they require so many bits. They are more common over *GF* (*p*) since division tends to be more expensive there.

A.10 Elliptic Curves: Algorithms

A.10.1 Full Addition and Subtraction (prime case)

The following algorithm implements a full addition (on a curve modulo *p*) in terms of affine coordinates.

Input: a prime $p > 3$; coefficients a, b for an elliptic curve E: $y^2 = x^3 + ax + b$ modulo p; points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on E.

Output: the point $P_2 := P_0 + P_1$.

1. If $P_0 = \mathcal{O}$ then output $P_2 \leftarrow P_1$ and stop
2. If $P_1 = \mathcal{O}$ then output $P_2 \leftarrow P_0$ and stop
3. If $x_0 \neq x_1$ then
   - 3.1 set $\lambda \leftarrow (y_0 - y_1) / (x_0 - x_1) \bmod p$
   - 3.2 go to step 7
4. If $y_0 \neq y_1$ then output $P_2 \leftarrow \mathcal{O}$ and stop
5. If $y_1 = 0$ then output $P_2 \leftarrow \mathcal{O}$ and stop
6. Set $\lambda \leftarrow (3x_1^2 + a) / (2y_1) \bmod p$
7. Set $x_2 \leftarrow \lambda^2 - x_0 - x_1 \bmod p$
8. Set $y_2 \leftarrow (x_1 - x_2)\lambda - y_1 \bmod p$
9. Output $P_2 \leftarrow (x_2, y_2)$

The above algorithm requires 3 or 4 modular multiplications and a modular inversion.

To subtract the point $P = (x, y)$, one adds the point $-P = (x, -y)$.

A.10.2 Full Addition and Subtraction (binary case)

The following algorithm implements a full addition (on a curve over *GF* (2^m)) in terms of affine coordinates.

Input: a field *GF* (2^m); coefficients a, b for an elliptic curve E: $y^2 + xy = x^3 + ax^2 + b$ over *GF* (2^m); points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on E.

Output: the point $P_2 := P_0 + P_1$.

1. If $P_0 = \mathcal{O}$, then output $P_2 \leftarrow P_1$ and stop
2. If $P_1 = \mathcal{O}$, then output $P_2 \leftarrow P_0$ and stop
3. If $x_0 \neq x_1$ then
   - 3.1 set $\lambda \leftarrow (y_0 + y_1) / (x_0 + x_1)$
   - 3.2 set $x_2 \leftarrow a + \lambda^2 + \lambda + x_0 + x_1$
   - 3.3 go to step 7
4. If $y_0 \neq y_1$ then output $P_2 \leftarrow \mathcal{O}$ and stop
5. If $x_1 = 0$ then output $P_2 \leftarrow \mathcal{O}$ and stop
6. Set
   - 6.1 $\lambda \leftarrow x_1 + y_1 / x_1$
   - 6.2 $x_2 \leftarrow a + \lambda^2 + \lambda$

7. $y_2 \leftarrow (x_1 + x_2)\,\lambda + x_2 + y_1$
8. $P_2 \leftarrow (x_2, y_2)$

The above algorithm requires 2 general multiplications, a squaring, and a multiplicative inversion.

To subtract the point $P = (x, y)$, one adds the point $-P = (x, x + y)$.

Further Speedup.

The two steps

6.1 $\lambda \leftarrow x_1 + y_1 / x_1$
6.2 $x_2 \leftarrow a + \lambda^2 + \lambda$

require a multiplication and an inversion. If $GF(2^m)$ is represented by a normal basis, this can be improved by replacing the two lines with the following steps.

6.1 $w \leftarrow x_1^2$
6.2 $x_2 \leftarrow w + b / w$
6.3 Solve the equation $\mu^2 + \mu = x_2 + a$ for μ via A.4.7 (normal basis case).
6.4 $z \leftarrow w + y_1$
6.5 If $z = \mu\, x_1$ then $\lambda \leftarrow \mu$ else $\lambda \leftarrow \mu + 1$

NOTE—The determination of whether $z = \mu\, x_1$ can be accomplished by computing one bit of the product $\mu\, x_1$ (any bit for which the corresponding bit of x_1 is 1) and comparing it to the corresponding bit of z.

This routine is more efficient than the ordinary method because it replaces the general multiplication by a multiplication by the constant b. (See [SOM95].)

A.10.3 Elliptic Scalar Multiplication

Scalar multiplication can be performed efficiently by the *addition-subtraction method* outlined below.

Input: an integer n and an elliptic curve point P.

Output: the elliptic curve point nP.

1. If $n = 0$ then output $\mathcal{O}$ and stop.
2. If $n < 0$ the set $Q \leftarrow (-P)$ and $k \leftarrow (-n)$, else set $Q \leftarrow P$ and $k \leftarrow n$.
3. Let $h_l h_{l-1} \ldots h_1 h_0$ be the binary representation of $3k$, where the most significant bit h_l is 1.
4. Let $k_l k_{l-1} \ldots k_1 k_0$ be the binary representation of k.
5. Set $S \leftarrow Q$.
6. For i from $l - 1$ downto 1 do
   Set $S \leftarrow 2S$.
   If $h_i = 1$ and $k_i = 0$ then compute $S \leftarrow S + Q$ via A.10.1 or A.10.2.
   If $h_i = 0$ and $k_i = 1$ then compute $S \leftarrow S - Q$ via A.10.1 or A.10.2.
7. Output S.

There are several modifications that improve the performance of this algorithm. These methods are summarized in [Gor98].

A.10.4 Projective Elliptic Doubling (prime case)

The projective form of the doubling formula on the curve $y^2 = x^3 + ax + b$ modulo p is

$$2\,(X_1, Y_1, Z_1) = (X_2, Y_2, Z_2),$$

where

$$\begin{aligned} M &= 3\,X_1^2 + a\,Z_1^4, \\ Z_2 &= 2Y_1Z_1, \\ S &= 4X_1\,Y_1^2, \\ X_2 &= M^2 - 2S, \\ T &= 8\,Y_1^4, \\ Y_2 &= M\,(S - X_2) - T. \end{aligned}$$

The algorithm `Double` given below performs these calculations.

Input: a modulus p; the coefficients a and b defining a curve E modulo p; projective coordinates (X_1, Y_1, Z_1) for a point P_1 on E.

Output: projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = 2P_1$.

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. If $T_2 = 0$ or $T_3 = 0$ then output (1, 1, 0) and stop.
5. If $a = p - 3$ then
   - $T_4 \leftarrow T_3^2$
   - $T_5 \leftarrow T_1 - T_4$
   - $T_4 \leftarrow T_1 + T_4$
   - $T_5 \leftarrow T_4 \times T_5$
   - $T_4 \leftarrow 3 \times T_5$ $= M$

   else
   - $T_4 \leftarrow a$
   - $T_5 \leftarrow T_3^2$
   - $T_5 \leftarrow T_5^2$
   - $T_5 \leftarrow T_4 \times T_5$
   - $T_4 \leftarrow T_1^2$
   - $T_4 \leftarrow 3 \times T_4$
   - $T_4 \leftarrow T_4 + T_5$ $= M$
6. $T_3 \leftarrow T_2 \times T_3$
7. $T_3 \leftarrow 2 \times T_3$ $= Z_2$
8. $T_2 \leftarrow T_2^2$
9. $T_5 \leftarrow T_1 \times T_2$
10. $T_5 \leftarrow 4 \times T_5$ $= S$
11. $T_1 \leftarrow T_4^2$
12. $T_1 \leftarrow T_1 - 2 \times T_5$ $= X_2$
13. $T_2 \leftarrow T_2^2$
14. $T_2 \leftarrow 8 \times T_2$ $= T$

15. $T_5 \leftarrow T_5 - T_1$
16. $T_5 \leftarrow T_4 \times T_5$
17. $T_2 \leftarrow T_5 - T_2$ $= Y_2$
18. $X_2 \leftarrow T_1$
19. $Y_2 \leftarrow T_2$
20. $Z_2 \leftarrow T_3$

This algorithm requires 10 field multiplications and 5 temporary variables. If a is small enough that multiplication by a can be done by repeated addition, only 9 field multiplications are required. If $a = p - 3$, then only 8 field multiplications are required (see [CC87]). The proportion of elliptic curves modulo p that can be rescaled so that $a = p - 3$ is about 1/4 if $p \equiv 1 \pmod 4$ and about 1/2 if $p \equiv 3 \pmod 4$. (See Annex A.9.5, Basic Facts.)

A.10.5 Projective Elliptic Addition (prime case)

The projective form of the adding formula on the curve $y^2 = x^3 + ax + b$ modulo p, is

$$(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2),$$

where

$$U_0 = X_0 Z_1^2,$$
$$S_0 = Y_0 Z_1^3,$$
$$U_1 = X_1 Z_0^2,$$
$$S_1 = Y_1 Z_0^3,$$
$$W = U_0 - U_1,$$
$$R = S_0 - S_1,$$
$$T = U_0 + U_1,$$
$$M = S_0 + S_1,$$
$$Z_2 = Z_0 Z_1 W,$$
$$X_2 = R^2 - TW^2,$$
$$V = TW^2 - 2X_2,$$
$$2Y_2 = VR - MW^3.$$

The algorithm `Add` given below performs these calculations.

Input: a modulus p; the coefficients a and b defining a curve E modulo p; projective coordinates (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) for points P_0 and P_1 on E, where Z_0 and Z_1 are nonzero.

Output: projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = P_0 + P_1$, unless $P_0 = P_1$. In this case, the triplet (0, 0, 0) is returned. (The triplet (0, 0, 0) is not a valid projective point on the curve, but rather a marker indicating that routine `Double` should be used.)

1. $T_1 \leftarrow X_0$ $= U_0$ (if $Z_1 = 1$)
2. $T_2 \leftarrow Y_0$ $= S_0$ (if $Z_1 = 1$)
3. $T_3 \leftarrow Z_0$
4. $T_4 \leftarrow X_1$
5. $T_5 \leftarrow Y_1$
6. If $Z_1 \neq 1$ then
   $T_6 \leftarrow Z_1$

$T_7 \leftarrow T_6^2$
$T_1 \leftarrow T_1 \times T_7$ = U_0 (if $Z_1 \neq 1$)
$T_7 \leftarrow T_6 \times T_7$
$T_2 \leftarrow T_2 \times T_7$ = S_0 (if $Z_1 \neq 1$)

7. $T_7 \leftarrow T_3^2$
8. $T_4 \leftarrow T_4 \times T_7$ = U_1
9. $T_7 \leftarrow T_3 \times T_7$
10. $T_5 \leftarrow T_5 \times T_7$ = S_1
11. $T_4 \leftarrow T_1 - T_4$ = W
12. $T_5 \leftarrow T_2 - T_5$ = R
13. If $T_4 = 0$ then
    If $T_5 = 0$ then output (0,0,0) and stop
    else output (1, 1, 0) and stop
14. $T_1 \leftarrow 2 \times T_1 - T_4$ = T
15. $T_2 \leftarrow 2 \times T_2 - T_5$ = M
16. If $Z_1 \neq 1$ then
    $T_3 \leftarrow T_3 \times T_6$
17. $T_3 \leftarrow T_3 \times T_4$ = Z_2
18. $T_7 \leftarrow T_4^2$
19. $T_4 \leftarrow T_4 \times T_7$
20. $T_7 \leftarrow T_1 \times T_7$
21. $T_1 \leftarrow T_5^2$
22. $T_1 \leftarrow T_1 - T_7$ = X_2
23. $T_7 \leftarrow T_7 - 2 \times T_1$ = V
24. $T_5 \leftarrow T_5 \times T_7$
25. $T_4 \leftarrow T_2 \times T_4$
26. $T_2 \leftarrow T_5 - T_4$
27. $T_2 \leftarrow T_2 / 2$ = Y_2
28. $X_2 \leftarrow T_1$
29. $Y_2 \leftarrow T_2$
30. $Z_2 \leftarrow T_3$

NOTE—the modular division by 2 in Step 27 can be carried out in the same way as in A.2.4.

This algorithm requires 16 field multiplications and 7 temporary variables. In the case $Z_1 = 1$, only 11 field multiplications and 6 temporary variables are required. (This is the case of interest for elliptic scalar multiplication.)

A.10.6 Projective Elliptic Doubling (binary case)

The projective form of the doubling formula on the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$ uses, not the coefficient b, but rather the field element

$$c := b^{2^{m-2}},$$

computed from b by $m - 2$ squarings. (We have $b = c^4$.) The formula is

$$2\,(X_1, Y_1, Z_1) = (X_2, Y_2, Z_2),$$

where

$$Z_2 = X_1 Z_1^2,$$
$$X_2 = (X_1 + c Z_1^2)^4,$$
$$U = Z_2 + X_1^2 + Y_1 Z_1,$$
$$Y_2 = X_1^4 Z_2 + U X_2.$$

The algorithm `Double` given below performs these calculations.

Input: a field of 2^m elements; the field elements a and c specifying a curve E over $GF(2^m)$; projective coordinates (X_1, Y_1, Z_1) for a point P_1 on E.

Output: projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = 2P_1$.

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. $T_4 \leftarrow c$
5. If $T_1 = 0$ or $T_3 = 0$ then output (1, 1, 0) and stop.
6. $T_2 \leftarrow T_2 \times T_3$
7. $T_3 \leftarrow T_3^2$
8. $T_4 \leftarrow T_3 \times T_4$
9. $T_3 \leftarrow T_1 \times T_3$ $= Z_2$
10. $T_2 \leftarrow T_2 + T_3$
11. $T_4 \leftarrow T_1 + T_4$
12. $T_4 \leftarrow T_4^2$
13. $T_4 \leftarrow T_4^2$ $= X_2$
14. $T_1 \leftarrow T_1^2$
15. $T_2 \leftarrow T_1 + T_2$ $= U$
16. $T_2 \leftarrow T_2 \times T_4$
17. $T_1 \leftarrow T_1^2$
18. $T_1 \leftarrow T_1 \times T_3$
19. $T_2 \leftarrow T_1 + T_2$ $= Y_2$
20. $T_1 \leftarrow T_4$
21. $X_2 \leftarrow T_1$
22. $Y_2 \leftarrow T_2$
23. $Z_2 \leftarrow T_3$

This algorithm requires 5 field squarings, 5 general field multiplications, and 4 temporary variables.

A.10.7 Projective Elliptic Addition (binary case)

The projective form of the adding formula on the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$ is

$$(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2),$$

where

$$U_0 = X_0 Z_1^2,$$
$$S_0 = Y_0 Z_1^3,$$
$$U_1 = X_1 Z_0^2,$$
$$W = U_0 + U_1,$$
$$S_1 = Y_1 Z_0^3,$$
$$R = S_0 + S_1,$$
$$L = Z_0 W$$
$$V = RX_1 + LY_1,$$
$$Z_2 = LZ_1,$$
$$T = R + Z_2,$$
$$X_2 = a Z_2^2 + TR + W^3,$$
$$Y_2 = TX_2 + VL^2.$$

The algorithm `Add` given below performs these calculations.

Input: a field of 2^m elements; the field elements a and b defining a curve E over $GF(2^m)$; projective coordinates (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) for points P_0 and P_1 on E, where Z_0 and Z_1 are nonzero.

Output: projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = P_0 + P_1$, unless $P_0 = P_1$. In this case, the triplet (0, 0, 0) is returned. (The triplet (0, 0, 0) is not a valid projective point on the curve, but rather a marker indicating that routine `Double` should be used.)

1. $T_1 \leftarrow X_0$ = U_0 (if $Z_1 = 1$)
2. $T_2 \leftarrow Y_0$ = S_0 (if $Z_1 = 1$)
3. $T_3 \leftarrow Z_0$
4. $T_4 \leftarrow X_1$
5. $T_5 \leftarrow Y_1$
6. If $a \neq 0$ then
   $T_9 \leftarrow a$
7. If $Z_1 \neq 1$ then
   $T_6 \leftarrow Z_1$
   $T_7 \leftarrow T_6^2$
   $T_1 \leftarrow T_1 \times T_7$ = U_0 (if $Z_1 \neq 1$)
   $T_7 \leftarrow T_6 \times T_7$
   $T_2 \leftarrow T_2 \times T_7$ = S_0 (if $Z_1 \neq 1$)
8. $T_7 \leftarrow T_3^2$
9. $T_8 \leftarrow T_4 \times T_7$ = U_1
10. $T_1 \leftarrow T_1 + T_8$ = W
11. $T_7 \leftarrow T_3 \times T_7$
12. $T_8 \leftarrow T_5 \times T_7$ = S_1
13. $T_2 \leftarrow T_2 + T_8$ = R
14. If $T_1 = 0$ then
    If $T_2 = 0$ then output (0, 0, 0) and stop
    else output (1, 1, 0) and stop
15. $T_4 \leftarrow T_2 \times T_4$
16. $T_3 \leftarrow T_1 \times T_3$ = L (= Z_2 if $Z_1 = 1$)
17. $T_5 \leftarrow T_3 \times T_5$
18. $T_4 \leftarrow T_4 + T_5$ = V

19. $T_5 \leftarrow T_3^2$
20. $T_7 \leftarrow T_4 \times T_5$
21. If $Z_1 \neq 1$ then
    $T_3 \leftarrow T_3 \times T_6$ $= Z_2$ (if $Z_1 \neq 1$)
22. $T_4 \leftarrow T_2 + T_3$ $= T$
23. $T_2 \leftarrow T_2 \times T_4$
24. $T_5 \leftarrow T_1^2$
25. $T_1 \leftarrow T_1 \times T_5$
26. If $a \neq 0$ then
    $T_8 \leftarrow T_3^2$
    $T_9 \leftarrow T_8 \times T_9$
    $T_1 \leftarrow T_1 + T_9$
27. $T_1 \leftarrow T_1 + T_2$ $= X_2$
28. $T_4 \leftarrow T_1 \times T_4$
29. $T_2 \leftarrow T_4 + T_7$ $= Y_2$
30. $X_2 \leftarrow T_1$
31. $Y_2 \leftarrow T_2$
32. $Z_2 \leftarrow T_3$

This algorithm requires 5 field squarings, 15 general field multiplications and 9 temporary variables. If $a = 0$, then only 4 field squarings, 14 general field multiplications and 8 temporary variables are required. (About half of the elliptic curves over $GF(2^m)$ can be rescaled so that $a = 0$. They are precisely the curves with order divisible by 4. See Annex A.9.5, Basic Facts.)

In the case $Z_1 = 1$, only 4 field squarings, 11 general field multiplications, and 8 temporary variables are required. If also $a = 0$, then only 3 field squarings, 10 general field multiplications, and 7 temporary variables are required. (These are the cases of interest for elliptic scalar multiplication.)

A.10.8 Projective Full Addition and Subtraction

The following algorithm `FullAdd` implements a full addition in terms of projective coordinates.

Input: a field of q elements; the field elements a and b defining a curve E over $GF(q)$; projective coordinates (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) for points P_0 and P_1 on E.

Output: projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = P_0 + P_1$.

1. If $Z_0 = 0$ then output $(X_2, Y_2, Z_2) \leftarrow (X_1, Y_1, Z_1)$ and stop.
2. If $Z_1 = 0$ then output $(X_2, Y_2, Z_2) \leftarrow (X_0, Y_0, Z_0)$ and stop.
3. Set $(X_2, Y_2, Z_2) \leftarrow$ `Add`$[(X_0, Y_0, Z_0), (X_1, Y_1, Z_1)]$.
4. If $(X_2, Y_2, Z_2) = (0, 0, 0)$ then set $(X_2, Y_2, Z_2) \leftarrow$ `Double`$[(X_1, Y_1, Z_1)]$
5. Output (X_2, Y_2, Z_2).

An *elliptic subtraction* is implemented as follows:

$$\texttt{Subtract}[(X_0, Y_0, Z_0), (X_1, Y_1, Z_1)] = \texttt{FullAdd}[(X_0, Y_0, Z_0), (X_1, U, Z_1)]$$

where

$$U = \begin{cases} -Y_1 \mod p & \text{if } q = p \\ X_1 Z_1 + Y_1 & \text{if } q = 2^m \end{cases}$$

A.10.9 Projective Elliptic Scalar Multiplication

Input: an integer n and an elliptic curve point $P = (X, Y, Z)$.

Output: the elliptic curve point $nP = (X^*, Y^*, Z^*)$.

1. If $n = 0$ or $Z = 0$ then output (1, 1, 0) and stop.
2. Set
   - 2.1 $X^* \leftarrow X$
   - 2.2 $Z^* \leftarrow Z$
   - 2.3 $Z_1 \leftarrow 1$
3. If $n < 0$ then go to Step 6
4. Set
   - 4.1 $k \leftarrow n$
   - 4.2 $Y^* \leftarrow Y$
5. Go to Step 8
6. Set $k \leftarrow (-n)$
7. If $q = p$ then set $Y^* \leftarrow -Y \pmod p$; else set $Y^* \leftarrow XZ + Y$
8. If $Z^* = 1$ then set $X_1 \leftarrow X^*$, $Y_1 \leftarrow Y^*$; else set $X_1 \leftarrow X^* / (Z^*)^2$, $Y_1 \leftarrow Y^* / (Z^*)^3$
9. Let $h_l h_{l-1} ... h_1 h_0$ be the binary representation of $3k$, where the most significant bit h_l is 1.
10. Let $k_l k_{l-1} ... k_1 k_0$ be the binary representation of k.
11. For i from $l - 1$ downto 1 do
    - 11.1 Set $(X^*, Y^*, Z^*) \leftarrow$ `Double`$[(X^*, Y^*, Z^*)]$.
    - 11.2 If $h_i = 1$ and $k_i = 0$ then set $(X^*, Y^*, Z^*) \leftarrow$ `FullAdd`$[(X^*, Y^*, Z^*), (X_1, Y_1, Z_1)]$.
    - 11.3 If $h_i = 0$ and $k_i = 1$ then set $(X^*, Y^*, Z^*) \leftarrow$ `Subtract`$[(X^*, Y^*, Z^*), (X_1, Y_1, Z_1)]$.
12. Output (X^*, Y^*, Z^*).

There are several modifications that improve the performance of this algorithm. These methods are summarized in [Gor98].

A.11 Functions for Elliptic Curve Parameter and Key Generation

A.11.1 Finding a Random Point on an Elliptic Curve (prime case)

The following algorithm provides an efficient method for finding a random point (other than $\mathcal{O}$) on a given elliptic curve over the finite field $GF(p)$.

Input: a prime $p > 3$ and the parameters a, b of an elliptic curve E modulo p.

Output: a randomly generated point (other than $\mathcal{O}$) on E.

1. Choose random x with $0 \le x < p$.
2. Set $\alpha \leftarrow x^3 + ax + b \bmod p$.
3. If $\alpha = 0$ then output $(x, 0)$ and stop.
4. Apply the appropriate technique from A.2.5 to find a square root modulo p of α or determine that none exist.
5. If the result of Step 4 indicates that no square roots exist, then go to Step 1. Otherwise the output of Step 4 is an integer β with $0 < \beta < p$ such that

$$\beta^2 \equiv \alpha \pmod{p}.$$

6. Generate a random bit μ and set $y \leftarrow (-1)^{\mu} \beta$.
7. Output (x, y).

A.11.2 Finding a Random Point on an Elliptic Curve (binary case)

The following algorithm provides an efficient method for finding a random point (other than $\mathcal{O}$) on a given elliptic curve over the finite field $GF(2^m)$.

Input: a field $GF(2^m)$ and the parameters a, b of an elliptic curve E over $GF(2^m)$.

Output: a randomly generated point (other than $\mathcal{O}$) on E.

1. Choose random x in $GF(2^m)$.
2. If $x = 0$ then output $(0, b^{2^{m-1}})$ and stop.
3. Set $\alpha \leftarrow x^3 + ax^2 + b$.
4. If $\alpha = 0$ then output $(x, 0)$ and stop.
5. Set $\beta \leftarrow x^{-2} \alpha$.
6. Apply the appropriate technique from A.4.7 to find an element z for which $z^2 + z = \beta$ or determine that none exist.
7. If the result of Step 6 indicates that no solutions exist, then go to Step 1. Otherwise the output of Step 6 is a solution z.
8. Generate a random bit μ and set $y \leftarrow (z + \mu) x$.
9. Output (x, y).

A.11.3 Finding a Point of Large Prime Order

If the order $\#E(GF(q)) = u$ of an elliptic curve E is nearly prime, the following algorithm efficiently produces a random point on E whose order is the large prime factor r of $u = kr$. (See Annex A.9.5 for the definition of *nearly prime*.)

Input: a prime r, a positive integer k not divisible by r, and an elliptic curve E over the field $GF(q)$.

Output: if $\#E(GF(q)) = kr$, a point G on E of order r. If not, the message "wrong order."

1. Generate a random point P (not $\mathcal{O}$) on E via A.11.1 or A.11.2.
2. Set $G \leftarrow kP$.
3. If $G = \mathcal{O}$ then go to Step 1.
4. Set $Q \leftarrow rG$.
5. If $Q \neq \mathcal{O}$ then output "wrong order" and stop.
6. Output G.

A.11.4 Curve Orders over Small Binary Fields

If d is "small" (i.e. it is feasible to perform 2^d arithmetic operations), then the order of the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^d)$ can be calculated directly as follows. Let

$$\mu = (-1)^{\mathrm{Tr}(a)}.$$

For each nonzero $x \in GF(2^d)$, let

$$\lambda(x) = \mathrm{Tr}(x + b/x^2).$$

Then

$$\#E(GF(2^d)) = 2^d + 1 + \mu \sum_{x \neq 0} (-1)^{\lambda(x)} .$$

A.11.5 Curve Orders over Extension Fields

Given the order of an elliptic curve E over a finite field $GF(2^d)$, the following algorithm computes the order of E over the extension field $GF(2^{de})$.

Input: positive integers d and e, an elliptic curve E defined over $GF(2^d)$, and the order w of E over $GF(2^d)$.

Output: the order u of E over $GF(2^{de})$.

1. Set $\mathcal{P} \leftarrow 2^d + 1 - w$ and $\mathcal{Z} \leftarrow 2^d$.
2. Compute via A.2.4 the Lucas sequence element V_e.
3. Compute $u := 2^{de} + 1 - V_e$.
4. Output u.

A.11.6 Curve Orders via Subfields

The algorithms of A.11.4 and A.11.5 allow construction of elliptic curves with known orders over $GF(2^m)$, provided that m is divisible by an integer d that is small enough for A.11.4. The following algorithm finds such curves with nearly prime orders when such exist. (See Annex A.9.5 for the definition of *nearly prime*.)

Input: a field $GF(2^m)$; a subfield $GF(2^d)$ for some (small) d dividing m; lower and upper bounds $r_{\min}$ and $r_{\max}$ for the base point order.

Output: elements $a, b \in GF(2^m)$ specifying an elliptic curve E, along with the nearly prime order $n = \#E(GF(2^m))$, if one exists; otherwise, the message "no such curve."

1. Select elements $a_0, b_0 \in GF(2^d)$ such that b_0 has not already been selected. (If all of the b_0's have already been tried, then output the message "no such curve" and stop.) Let E be the elliptic curve $y^2 + xy = x^3 + a_0 x^2 + b_0$.
2. Compute the order $w = \#E(GF(2^d))$ via A.11.4.
3. Compute the order $u = \#E(GF(2^m))$ via A.11.5.
4. Test u for near-primality via A.15.3.
5. If u is nearly prime, then set $\lambda \leftarrow 0$ and $n \leftarrow u$ and go to Step 9.
6. Set $u' = 2^{m+1} + 2 - u$.
7. Test u' for near-primality via A.15.3.
8. If u' is nearly prime, then set $\lambda \leftarrow 1$ and $n \leftarrow u'$, else go to Step 1.
9. Find the elements $a_1, b_1 \in GF(2^m)$ corresponding to a_0 and b_0 via A.5.7.
10. If $\lambda = 0$ then set $\tau \leftarrow 0$. If $\lambda = 1$ and m is odd, then set $\tau \leftarrow 1$. Otherwise, find an element $\tau \in GF(2^m)$ of trace 1 by trial and error using A.4.5.
11. Set $a \leftarrow a_1 + \tau$ and $b \leftarrow b_1$
12. Output n, a, b.

NOTE—It follows from the Basic Facts of A.9.5 that any a_0 can be chosen at any time in Step 1.

A.12 Functions for Elliptic Curve Parameter and Key Validation

A.12.1 The MOV Condition

The *MOV condition* ensures that an elliptic curve is not vulnerable to the reduction attack of Menezes, Okamoto and Vanstone [MOV93].

Before performing the algorithm, it is necessary to select an *MOV threshold.* This is a positive integer B such that taking discrete logarithms over $GF(q^B)$ is judged to be at least as difficult as taking elliptic discrete logarithms over $GF(q)$. It follows from the complexity discussions of D.4.1 and from [Men95] that B should be large enough so that

$$T(mB) \geq m$$

where $2^m \leq q < 2^{m+1}$ and

$$T(n) = \frac{8}{3}\left(3n\left(\log_2(n \ln 2)\right)^2\right)^{1/3} - 17.135872.$$

(As before, ln x denotes the natural logarithm of x.). The constant in this formula follows from [DL95] and [Odl95]. The following table gives the smallest value of B satisfying the above condition, for each q whose bit length m is between 128 and 512.

m	*B*	*m*	*B*	*m*	*B*
128-142	6	281-297	14	407-420	22
143-165	7	298-313	15	421-434	23
166-186	8	314-330	16	435-448	24
187-206	9	331-346	17	449-462	25
207-226	10	347-361	18	463-475	26
227-244	11	362-376	19	476-488	27
245-262	12	377-391	20	489-501	28
263-280	13	392-406	21	502-512	29

Once an appropriate B has been selected, the following algorithm checks the MOV condition for the choice of field size q and base point order r by verifying that q^i is not congruent to 1 modulo r for any $i \leq B$

Input: an MOV threshold B, a prime-power q, and a prime r.

Output: the message "True" if the MOV condition is satisfied for an elliptic curve over $GF(q)$ with a base point of order r; the message "False" otherwise.

1. Set $t \leftarrow 1$.
2. For i from 1 to B do
   2.1 Set $t \leftarrow tq \bmod r$.
   2.2 If $t = 1$ then output "False" and stop.

3. Output "True."

A.12.2 The Weil Pairing

The *Weil pairing* is a function <*P*, *Q*> of pairs *P*, *Q* of points on an elliptic curve *E*. It can be used to determine whether *P* and *Q* are multiples of each other. It will be used in A.12.3.

Let $l > 2$ be prime, and let *P* and *Q* be points on *E* with $lP = lQ = \mathcal{O}$. The following procedure computes the Weil pairing.

— Given three points (x_0, y_0), (x_1, y_1), (u, v) on *E*, define the function $f((x_0, y_0), (x_1, y_1), (u, v))$ by

$$\begin{cases} u - x_1 & \text{if } x_0 = x_1 \text{ and } y_0 = -y_1 \\ (3x_1^2 + a)(u - x_1) - 2y_1(v - y_1) & \text{if } x_0 = x_1 \text{ and } y_0 = y_1 \\ (x_0 - x_1)v - (y_0 - y_1)u - (x_0y_1 - x_1y_0) & \text{if } x_0 \neq x_1 \end{cases}$$

if *E* is the curve $y^2 = x^3 + ax + b$ over *GF* (*p*), and

$$\begin{cases} u + x_1 & \text{if } x_0 = x_1 \text{ and } y_0 = x_1 + y_1 \\ x_1^3 + (x_1^2 + y_1)u + x_1v & \text{if } x_0 = x_1 \text{ and } y_0 = y_1 \\ (x_0 + x_1)v + (y_0 + y_1)u + (x_0y_1 + x_1y_0) & \text{if } x_0 \neq x_1 \end{cases}$$

if *E* is the curve $y^2 + xy = x^3 + ax^2 + b$ over *GF* (2^m).

— Given points *A*, *B*, *C* on *E*, let

$$g(A, B, C) := f(A, B, C) / f(A + B, -A - B, C).$$

— The *Weil function h* is computed via the following algorithm.

Input: a prime $l > 2$; a curve *E*; finite points *D*, *C* on *E* with $lD = lC = \mathcal{O}$.

Output: the Weil function $h(D, C)$.

1. Set $A \leftarrow D$, $B \leftarrow D$, $h \leftarrow 1$, $n \leftarrow l$.
2. Set $n \leftarrow \lfloor n / 2 \rfloor$.
3. Set $h \leftarrow g(B, B, C)^n \times h$.
4. Set $B \leftarrow 2B$.
5. If *n* is odd then
   5.1 If $n = 1$
   then $h \leftarrow g(A, -A, C) \times h$
   else $h \leftarrow g(A, B, C) \times h$.
   5.2 Set $A \leftarrow A + B$.
6. If $n > 1$ then go to Step 2.
7. Output *h*.

— Given points *R* and *S* on *E*, let

$$j(R, S) := h(R, S) / h(S, R).$$

— Given points P and Q on E with $lP = lQ = \mathcal{O}$, the Weil pairing $\langle P, Q \rangle$ is computed as follows:

Choose random points T, U on E and let

$$V = P + T, \quad W = Q + U.$$

Then

$$\langle P, Q \rangle = j(T, U)\ j(U, V)\ j(V, W)\ j(W, T).$$

If, in evaluating $\langle P, Q \rangle$, one encounters $f((x_0, y_0), (x_1, y_1), (u, v)) = 0$, then the calculation fails. In this (unlikely) event, repeat the calculation with newly chosen T and U.

In the case $l = 2$, the Weil pairing is easily computed as follows: $\langle P, Q \rangle$ equals 1 if $P = Q$ and –1 otherwise.

We define $\langle P, \mathcal{O} \rangle = 1$ for all points P.

A.12.3 Verification of Cofactor

Let E be an elliptic curve over $GF(q)$ of order $u = kr$, where r is the prime order of the base point G. The integer k is called the *cofactor*. The DL and EC key agreement schemes provide an option for exponentiation or scalar multiplication of the shared secret value by the cofactor. In order to implement this option, it is necessary to know the cofactor k.

In the DL case, the cofactor is simply $k = (q - 1)/r$. In the EC case, a simple formula exists only if $r > 4\sqrt{q}$ (which is typically the case). In this case, k can be computed directly from the verified parameters q and r (see Annex A.16.8) via the formula

$$k := \left\lfloor \frac{(\sqrt{q} + 1)^2}{r} \right\rfloor.$$

If $r \le 4\sqrt{q}$, then k cannot be computed from q and r alone. Therefore the value of k must be sent along with the EC parameters, and should be verified by the recipient. This verification is accomplished by a rather complex algorithm, as described below.

Torsion points.

If l is a prime, then a *l-torsion point* is a finite point T such that $lT = \mathcal{O}$. The following algorithm (a subroutine of the cofactor verification procedure to be given below) generates an l-torsion point or reports that none exist. The algorithm is probabilistic, but the probability of error can be made as small as desired. (A parameter β is included for this purpose.)

Input: the EC parameters q, a, and b; the putative order $u = \#E(GF(q))$; a prime $l \neq r$; the largest positive integer v such that v such that l^v divides u; and a positive integer β.

Output: an l-torsion point T and a positive integer $\alpha \le v$ such that $T = l^\alpha P$ for some point P or the message "no torsion point found."

1. Set $h \leftarrow kr / l^v$.
2. Set $\mu \leftarrow 0$
3. Set $\mu \leftarrow \mu + 1$

4. If $\mu > \beta$ then output "no torsion point found" and stop
5. Generate a random finite point R via A.11.1 or A.11.2
6. Compute $S := hR$
7. If $S = \mathcal{O}$ then go to Step 3
8. Set $\alpha \leftarrow 1$
9. Set $\alpha \leftarrow \alpha + 1$
10. It $\alpha > v$ then output "no torsion point found" and stop
11. Set $T \leftarrow S$
12. Set $S \leftarrow lS$
13. If $S \neq \mathcal{O}$ then go to Step 9
14. Output T and α

Cofactor verification procedure.

Input: the EC parameters q, a, b, r, and k; an upper bound ε for the probability of error.

Output: "cofactor confirmed" or "cofactor rejected."

1. Set $u \leftarrow kr$.
2. If $u < (\sqrt{q} - 1)^2$ or $u > (\sqrt{q} + 1)^2$ then output "cofactor rejected" and stop.
3. Factor

$$k := l_1^{v_1} \ldots l_s^{v_s}$$

where the l_i's are distinct primes and each v_i is positive.

4. For i from 1 to s do
   - 4.1 Set $l \leftarrow l_i$, $v \leftarrow v_i$.
   - 4.2 Set

$$\beta := \left\lceil \frac{\log(s/\varepsilon)}{v \log l} \right\rceil$$

   - 4.3 Set $j \leftarrow 0$.
   - 4.4 If $q \equiv 1 \pmod{l}$ then set $e \leftarrow 0$, $f \leftarrow 0$, $U \leftarrow \mathcal{O}$, $V \leftarrow \mathcal{O}$.
   - 4.5. Set $j \leftarrow j + 1$
   - 4.6 If $j > v\beta$ then output "cofactor rejected" and stop.
   - 4.7 Generate an l-torsion point T and associated integer α via the subroutine.
   - 4.8 If the output of the subroutine is "no torsion point found" then output "cofactor rejected" and stop.
   - 4.9 If $q \not\equiv 1 \pmod{l}$ then set $\rho \leftarrow \alpha$ else
     - 4.9.1 If $\alpha \leq e$ then go to Step 4.10
     - 4.9.2 Compute the Weil pairing $w := \langle T, V \rangle$ via A.12.2
     - 4.9.3 If $\alpha \geq f$ then go to Step 4.9.6
     - 4.9.4 If $w \neq 1$ then set $e \leftarrow \alpha$, $U \leftarrow T$
     - 4.9.5 Go to Step 4.9.8
     - 4.9.6 If $w \neq 1$ then set $e \leftarrow f$, $U \leftarrow V$
     - 4.9.7 Set $f \leftarrow \alpha$, $V \leftarrow T$
     - 4.9.8 Set $\rho \leftarrow e + f$.
   - 4.10 If $\rho < v$ then go to Step 4.5
5. Output "cofactor confirmed."

A.12.4 Constructing Verifiably Pseudo-Random Elliptic Curves (prime case)

It is common to use an elliptic curve selected at random from the curves of appropriate order. (See Annex A.9.5.) The following algorithm produces a set of elliptic curve parameters for such a curve over a field $GF(p)$, along with sufficient information for others to verify that the curve was indeed chosen pseudo-randomly. (The algorithm is consistent with the one given in [ANS98e].)

See Section 5.5 of this Standard for the conversion routines BS2IP and I2BSP.

It is assumed that the following quantities have been chosen:

— a prime modulus p
— lower and upper bounds r_{min} and r_{max} for the order of the base point
— a cryptographic hash function H with output length B bits, where

$$B \geq \left\lceil \frac{1}{2} \log_2 (r_{\min}) \right\rceil$$

— the bit length L of inputs to H, satisfying $L \geq B$.

We also adopt the following notation:

$$v = \lceil \log_2 p \rceil,$$
$$s = \lfloor (v - 1)/B \rfloor,$$
$$w = v - B\,s - 1.$$

Input: a prime modulus p; lower and upper bounds r_{min} and r_{max} for r; a trial division bound $l_{max} < r_{min}$.

Output: a bit string X; EC parameters $q = p$, a, b, r, and G.

1. Choose an arbitrary bit string X of bit length L.
2. Compute $h := H(X)$.
3. Let W_0 be the bit string obtained by taking the w rightmost bits of h.
4. Convert the length-L bit string X to an integer z via BS2IP.
5. For i from 1 to s do:
   5.1 Convert the integer $(z + i) \bmod (2^L)$ to a length-L bit string X_i via I2BSP.
   5.2 Compute $W_i := H(X_i)$.
6. Let W be the bit string obtained by the concatenation of W_0, W_1, …, W_s as follows:

$$W = W_0 \,||\, W_1 \,||\, \ldots \,||\, W_s.$$

7. Convert the length-$(v - 1)$ bit string W to an integer c via BS2IP.
8. If $c = 0$ or $4c + 27 \equiv 0 \pmod{p}$, then go to Step 1.
9. Choose integers $a, b \in GF(p)$ such that

$$cb^2 \equiv a^3 \pmod{p}.$$

   (The simplest choice is $a = c$ and $b = c$. However, one may want to choose differently for performance reasons; e.g. the condition $a = p - 3$ used in A.10.4.)

10. Compute the order u of the elliptic curve E over $GF(p)$ given by $y^2 = x^3 + ax + b$. (This can be done using the techniques described in [ANS98h].)
11. Test u for near-primality via A.15.3.

12. If u is not nearly prime, then go to Step 1. (Otherwise the output of A.15.3 consists of the integers k, r.)
13. Generate a point G on E of order r via A.11.3.
14. Output X, a, b, r, G.

A.12.5 Verification of Elliptic Curve Pseudo-Randomness (prime case)

The following algorithm verifies the validity of a set of elliptic curve parameters. In addition, it determines whether an elliptic curve over $GF(p)$ was generated using the method of A.12.4.

The quantities B, L, v, s, and w, and the hash function H, are as in A.12.4. See Section 5.5 of this Standard for the conversion routines BS2IP and I2BSP.

Input: a bit string X of length L; EC parameters $q = p$, a, b, r, and $G = (x, y)$.

Output: "True" or "False."

1. Compute $h := H(X)$.
2. Let W_0 be the bit string obtained by taking the w rightmost bits of h.
3. Convert the bit string X to an integer z via BS2IP.
4. For i from 1 to s do:
   4.1 Convert the integer $(z + i) \bmod (2^L)$ to a length-L bit string X_i via I2BSP.
   4.2 Compute $W_i := H(X_i)$.
5. Let W be the bit string obtained by the concatenation of W_0, W_1, …, W_s as follows:

$$W = W_0 \| W_1 \| \ldots \| W_s.$$

6. Convert the length-$(v - 1)$ bit string W to an integer c via BS2IP.
7. Perform the following checks.
   7.1 $c > 0$
   7.2 $(4c + 27 \bmod p) > 0$
   7.3 $cb^2 \equiv a^3 \pmod{p}$
   7.4 $G \neq \mathcal{O}$
   7.5 $y^2 \equiv x^3 + ax + b \pmod{p}$
   7.6 $rG = \mathcal{O}$
8. If all the checks in Step 7 work, then output "True"; otherwise output "False."

A.12.6 Constructing Verifiably Pseudo-Random Elliptic Curves (binary case)

It is common to use an elliptic curve selected at random from the curves of appropriate order. (See Annex A.9.5.) The following algorithm produces a set of elliptic curve parameters for such a curve over a field $GF(2^m)$, along with sufficient information for others to verify that the curve was indeed chosen pseudo-randomly. (The algorithm is consistent with the one given in [ANS98e].)

See Section 5.5 of this Standard for the conversion routines BS2IP, I2BSP, BS2OSP, and OS2FEP.

It is assumed that the following quantities have been chosen:

— a field $GF(2^m)$
— lower and upper bounds r_{min} and r_{max} for the order of the base point
— a cryptographic hash function H with output length B bits, where

$$B \geq \left\lceil \frac{1}{2} \log_2 (r_{\min}) \right\rceil$$

— the bit length L of inputs to H, satisfying $L \geq B$.

We also adopt the following notation:

$$s = \lfloor (m - 1)/B \rfloor,$$
$$w = m - B\,s.$$

Input: a field $GF(2^m)$; lower and upper bounds $r_{\min}$ and $r_{\max}$ for r; a trial division bound $l_{\max} < r_{\min}$.

Output: a bit string X; EC parameters $q = 2^m$, a, b, r, and G.

1. Choose an arbitrary bit string X of bit length L.
2. Compute $h := H(X)$.
3. Let W_0 be the bit string obtained by taking the w rightmost bits of h.
4. Convert the length-L bit string X to an integer z via BS2IP.
5. For i from 1 to s do:
   5.1 Convert the integer $(z + i) \bmod (2^L)$ to a length-L bit string X_i via I2BSP.
   5.2 Compute $W_i := H(X_i)$.
6. Let W be the bit string obtained by the concatenation of W_0, W_1, …, W_s as follows:

$$W = W_0 \,||\, W_1 \,||\, \ldots \,||\, W_s.$$

7. Convert the length-m bit string W to a field element b via BS2OSP and OS2FEP.
8. If $b = 0$ then go to Step 1.
9. Let a be an arbitrary element in $GF(2^m)$. (The simplest choice is $a = 0$, which also allows for the efficient implementation given in A.10.7. However, one may want to choose differently for other reasons, such as other performance issues or the availability of suitable curves.)
10. Compute the order u of the elliptic curve E over $GF(2^m)$ given by $y^2 + xy = x^3 + ax^2 + b$. (This can be done using the techniques described in [ANS98h]. See also [Men93b].)
11. Test u for near-primality via A.15.3.
12. If u is not nearly prime, then go to Step 1. (Otherwise the output of A.15.3 consists of the integers k, r.)
13. Generate a point G on E of order r via A.11.3.
14. Output X, a, b, r, G.

A.12.7 Verification of Elliptic Curve Pseudo-Randomness (binary case)

The following algorithm verifies the validity of a set of elliptic curve parameters. In addition, it determines whether an elliptic curve over $GF(2^m)$ was generated using the method of A.12.6.

The quantities B, L, s, and w, and the hash function H, are as in A.12.6. See Section 5.5 of this Standard for the conversion routines BS2IP, I2BSP, BS2OSP, and OS2FEP.

Input: a bit string X of length L; EC parameters $q = 2^m$, a, b, r, and $G = (x, y)$.

Output: "True" or "False."

1. Compute $h := H(X)$.
2. Let W_0 be the bit string obtained by taking the w rightmost bits of h.
3. Convert the bit string X to an integer z via BS2IP.

4. For i from 1 to s do:
   4.1 Convert the integer $(z + i) \bmod (2^L)$ to a length-L bit string X_i via I2BSP.
   4.2 Compute $W_i := H(X_i)$.
5. Let W be the bit string obtained by the concatenation of W_0, W_1, …, W_s as follows:

$$W = W_0 \| W_1 \| \ldots \| W_s.$$

6. Convert the length-m bit string W to the field element b' via BS2OSP and OS2FEP.
7. Perform the following checks.
   7.1 $b \neq 0$
   7.2 $b = b'$
   7.3 $G \neq \mathcal{O}$
   7.4 $y^2 + xy = x^3 + ax^2 + b$
   7.5 $rG = \mathcal{O}$
8. If all the checks in Step 7 work, then output "True"; otherwise output "False."

A.12.8 Decompression of *y* Coordinates (prime case)

The following algorithm recovers the y coordinate of an elliptic curve point from its compressed form.

Input: a prime number p, an elliptic curve E defined modulo p, the x coordinate of a point (x, y) on E, and the compressed representation $\tilde{y}$ of the y coordinate.

Output: the y coordinate of the point.

1. Compute $g := x^3 + ax + b \bmod p$
2. Find a square root z of g modulo p via A.2.5. If the output of A.2.5 is "no square roots exist," then return an error message and stop.
3. Let $\tilde{z}$ be the rightmost bit of z (in other words, $z \bmod 2$).
4. If $\tilde{z} = \tilde{y}$ then $y \leftarrow z$, else $y \leftarrow p - z$.
5. Output y.

NOTE—when implementing the algorithm from A.2.5, the existence of modular square roots should be checked. Otherwise, a value may be returned even if no modular square roots exist.

A.12.9 Decompression of *y* Coordinates (binary case)

The following algorithm recovers the y coordinate of an elliptic curve point from its compressed form.

Input: a field $GF(2^m)$, an elliptic curve E defined over $GF(2^m)$, the x coordinate of a point (x, y) on E, and the compressed representation $\tilde{y}$ of the y coordinate.

Output: the y coordinate of the point.

1. If $x = 0$ then compute $y := \sqrt{b}$ via A.4.1 and go to Step 7.
2. Compute the field element $\alpha := x^3 + ax^2 + b$ in $GF(2^m)$
3. Compute the element $\beta := \alpha (x^2)^{-1}$ via A.4.4
4. Find a field element z such that $z^2 + z = \beta$ via A.4.7. If the output of A.4.7 is "no solutions exist," then return an error message and stop.
5. Let $\tilde{z}$ be the rightmost bit of z
6. Compute $y := (z + \tilde{z} + \tilde{y})\, x$

7. Output y.

NOTES

1—When implementing the algorithm from A.4.7, the existence of solutions to the quadratic equation should be checked. Otherwise, a value may be returned even if no solutions exist.

2—If both coordinates are compressed, the x coordinate must be decompressed first and then the y coordinate. (See Annex A.12.10.)

A.12.10 Decompression of *x* Coordinates (binary case)

The following algorithm recovers the x coordinate of an elliptic curve point from its compressed form. The statement of the algorithm assumes that the point has prime order, since that is the case of cryptographic interest; but in fact that condition is unnecessarily restrictive. (See Annex A.9.6.)

In addition to the EC parameters and the compact representation of the point, it is necessary to possess the trace Tr (a) of the coefficient a of the curve (see Annex A.4.5). If **F** is represented by a polynomial basis

$$B = \{t^{m-1}, \ldots, t, 1\}$$

and m is even, it is also necessary to possess the smallest positive integer s such that Tr (t^s) = 1, since this indicates which bit has been dropped during compression. If a given set of EC parameters is to be reused, these two quantities can be computed once and stored with the parameters.

Input: a field **F** of 2^m elements; an elliptic curve E defined over **F**; the compact form $\tilde{x}$ of the x coordinate of a point P on E of prime order; the trace of the coefficient a of the curve; if **F** is represented by a polynomial basis $\{t^{m-1}, \ldots, t, 1\}$ and m is even, the smallest positive integer s such that Tr (t^s) = 1.

Output: the x coordinate of P.

*If **F** is represented by a normal basis or m is odd:*

1. Set $x^* := (\tilde{x} \,\|\, 0)$
2. If Tr (x^*) = Tr (a) then set $x \leftarrow x^*$; else set $x := (\tilde{x} \,\|\, 1)$
3. Output x

*If **F** is represented by a polynomial basis and m is even:*

1. Write $\tilde{x} = (u_{m-1} \ldots u_1)$
2. Let x^* be the field element

$$x^* := (w_{m-1} \ldots w_0)$$

where

$$w_j = u_j \text{ for } j > s$$

$$w_s = 0$$

$$w_j = u_{j+1} \text{ for } 0 \le j < s.$$

That is, x^* is the polynomial

$$u_{m-1} t^{m-1} + \ldots + u_{s+1} t^{s+1} + u_s t^{s-1} + \ldots + u_1.$$

3. If Tr (x^*) = Tr (a) then output $x \leftarrow x^*$; else ouput $x \leftarrow x^* + t^s$

A.13 Class Group Calculations

The following computations are necessary for the complex multiplication technique described in A.14.

A.13.1 Overview

A reduced symmetric matrix is one of the form

$$S = \begin{pmatrix} A & B \\ B & C \end{pmatrix}$$

where the integers A, B, C satisfy the following conditions:

i) GCD (A, $2B$, C) = 1,
ii) $|2B| \le A \le C$,
iii) If either $A = |2B|$ or $A = C$, then $B \ge 0$.

We will abbreviate S as [A, B, C] when typographically convenient.

The determinant $D := AC - B^2$ of S will be assumed throughout this section to be positive and *squarefree* (i.e., containing no square factors).

Given D, the *class group* H (D) is the set of all reduced symmetric matrices of determinant D. The *class number* h(D) is the number of matrices in H(D).

The class group is used to construct the *reduced class polynomial*. This is a polynomial w_D (t) with integer coefficients of degree h (D). The reduced class polynomial is used in A.14 to construct elliptic curves with known orders.

A.13.2 Class Group and Class Number

The following algorithm produces a list of the reduced symmetric matrices of a given determinant D. See [Bue89].

Input: a squarefree determinant $D > 0$.

Output: the class group H (D).

1. Let s be the largest integer less than $\sqrt{D/3}$.
2. For B from 0 to s do
   - 2.1 List the positive divisors $A_1, \ldots, A_r$ of $D + B^2$ that satisfy $2B \le A \le \sqrt{D + B^2}$.
   - 2.2 For i from 1 to r do
     - 2.2.1 Set $C \leftarrow (D + B^2) / A_i$
     - 2.2.2 If GCD (A_i, $2B$, C) = 1 then
       list [A_i, B, C].
       if $0 < 2B < A_i < C$ then list [A_i, $-B$, C].
3. Output list.

Example: $D = 71$. We need to check $0 \le B < 5$.

— For $B = 0$, we have $A = 1$, leading to [1,0,71].
— For $B = 1$, we have $A = 2,3,4,6,8$, leading to [3, ±1,24] and [8, ±1,9].
— For $B = 2$, we have $A = 5$, leading to [5, ±2, 15].
— For $B = 3$, we have $A = 8$, but no reduced matrices.
— For $B = 4$, we have no divisors A in the right range.

Thus the class group is

$$H(71) = \{[1,0,71], [3, \pm 1,24], [8, \pm 1,9], [5, \pm 2, 15]\} .$$

and the class number is $h(71) = 7$.

A.13.3 Reduced Class Polynomials

Let

$$F(z) = 1 + \sum_{j=1}^{\infty} (-1)^j \left(z^{(3j^2 - j)/2} + z^{(3j^2 + j)/2} \right)$$
$$= 1 - z - z^2 + z^5 + z^7 - z^{12} - z^{15} + \ldots$$

and

$$\theta = \exp\left(\frac{-\sqrt{D} + Bi}{A} \pi \right).$$

Let

$$f_0(A, B, C) = \theta^{-1/24} F(-\theta) / F(\theta^2),$$
$$f_1(A, B, C) = \theta^{-1/24} F(\theta) / F(\theta^2),$$
$$f_2(A, B, C) = \sqrt{2}\, \theta^{1/12} F(\theta^4) / F(\theta^2).$$

NOTE—since

$$|\theta| < e^{-\pi\sqrt{3}/2} \approx 0.0658287 ,$$

the series $F(z)$ used in computing the numbers $f_J(A, B, C)$ converges as quickly as a power series in $e^{-\pi\sqrt{3}/2}$.

If $[A, B, C]$ is a matrix of determinant D, then its *class invariant* is

$$\mathfrak{C}(A, B, C) = (N \lambda^{-BL} 2^{-I/6} (f_J(A, B, C))^K)^G,$$

where:

$$G = \mathrm{GCD}(D,3),$$

$$I = \begin{cases} 3 & \text{if } D \equiv 1,2,6,7 \pmod 8, \\ 0 & \text{if } D \equiv 3 \pmod 8 \text{ and } D \not\equiv 0 \pmod 3, \\ 2 & \text{if } D \equiv 3 \pmod 8 \text{ and } D \equiv 0 \pmod 3, \\ 6 & \text{if } D \equiv 5 \pmod 8, \end{cases}$$

$$J = \begin{cases} 0 & \text{for } AC \text{ odd}, \\ 1 & \text{for } C \text{ even}, \\ 2 & \text{for } A \text{ even}, \end{cases}$$

$$K = \begin{cases} 2 & \text{if } D \equiv 1,2,6 \pmod 8, \\ 1 & \text{if } D \equiv 3,7 \pmod 8, \\ 4 & \text{if } D \equiv 5 \pmod 8, \end{cases}$$

$$L = \begin{cases} A - C + A^2C & \text{if } AC \text{ odd or } D \equiv 5 \pmod 8 \text{ and } C \text{ even}, \\ A + 2C - AC^2 & \text{if } D \equiv 1,2,3,6,7 \pmod 8 \text{ and } C \text{ even}, \\ A - C + 5AC^2 & \text{if } D \equiv 3 \pmod 8 \text{ and } A \text{ even}, \\ A - C - AC^2 & \text{if } D \equiv 1,2,5,6,7 \pmod 8 \text{ and } A \text{ even}, \end{cases}$$

$$M = \begin{cases} (-1)^{(A^2-1)/8} & \text{if } A \text{ odd}, \\ (-1)^{(C^2-1)/8} & \text{if } A \text{ even}, \end{cases}$$

$$N = \begin{cases} 1 & \text{if } D \equiv 5 \pmod 8 \\ & \quad \text{or } D \equiv 3 \pmod 8 \text{ and } AC \text{ odd} \\ & \quad \text{or } D \equiv 7 \pmod 8 \text{ and } AC \text{ even}, \\ M & \text{if } D \equiv 1,2,6 \pmod 8 \\ & \quad \text{or } D \equiv 7 \pmod 8 \text{ and } AC \text{ odd}, \\ -M & \text{if } D \equiv 3 \pmod 8 \text{ and } AC \text{ even}, \end{cases}$$

$\lambda = e^{\pi iK/24}$.

If $[A_1, B_1, C_1]$, ..., $[A_h, B_h, C_h]$ are the reduced symmetric matrices of (positive squarefree) determinant D, then the *reduced class polynomial* for D is

$$w_D(t) = \prod_{j=1}^{h} (t - \mathbf{C}(A_j, B_j, C_j)).$$

The reduced class polynomial has integer coefficients.

NOTE—The above computations must be performed with sufficient accuracy to identify each coefficient of the polynomial $w_D(t)$. Since each such coefficient is an integer, this means that the error incurred in calculating each coefficient should be less than 1/2.

Example.

$$w_{71}(t) = \left(t - \frac{1}{\sqrt{2}}\mathbf{F}_0(1,0,71)\right)\cdot$$

$$\left(t - \frac{e^{-i\pi/8}}{\sqrt{2}}\mathbf{F}_1(3,1,24)\right)\left(t - \frac{e^{i\pi/8}}{\sqrt{2}}\mathbf{F}_1(3,-1,24)\right)$$

$$\left(t - \frac{e^{-23i\pi/24}}{\sqrt{2}}\mathbf{F}_2(8,1,9)\right)\left(t - \frac{e^{23i\pi/24}}{\sqrt{2}}\mathbf{F}_2(8,-1,9)\right)$$

$$\left(t + \frac{e^{-5i\pi/12}}{\sqrt{2}}\mathbf{F}_0(5,2,15)\right)\left(t + \frac{e^{5i\pi/12}}{\sqrt{2}}\mathbf{F}_0(5,-2,15)\right)$$

$$= (t - 2.13060682983889533005591468688942503...)$$

$$(t - (0.95969178530567025250797047645507504...) +$$

$$(0.34916071001269654799855316293926907...)i)$$

$$(t - (0.95969178530567025250797047645507504...) -$$

$$(0.34916071001269654799855316293926907...)i)$$

$$(t + (0.75613568804001789053564010985317 72...) +$$

$$(0.07375086316308890052407649445676 75...)i)$$

$$(t + (0.75613568804001789053564010985317 72...) -$$

$$(0.07375086316308890052407649445676 75...)i)$$

$$(t + (0.26885951218510002700028771004661 02...) -$$

$$(0.84108577401329800103648634224905292...)i)$$

$$(t + (0.26885951218510002700028771004661 02...) +$$

$$(0. 84108577401329800103648634224905292...)i)$$

$$= t^7 - 2t^6 - t^5 + t^4 + t^3 + t^2 - t - 1.$$

A.14 Complex Multiplication

A.14.1 Overview

If E is a non-supersingular elliptic curve over $GF(q)$ of order u, then

$$Z = 4q - (q + 1 - u)^2$$

is positive by the Hasse bound (see Annex A.9.5). Thus there is a unique factorization

$$Z = DV^2$$

where D is squarefree (i.e. contains no square factors). Thus, for each non-supersingular elliptic curve over $GF(q)$ of order u, there exists a unique squarefree positive integer D such that

(*) $$4q = W^2 + DV^2,$$

(**) $$u = q + 1 \pm W$$

for some W and V.

We say that E has *complex multiplication* by D (or, more properly, by $\sqrt{-D}$). We call D a *CM discriminant* for q.

If one knows D for a given curve E, one can compute its order via (*) and (**). As we shall see, one can construct the curves with CM by small D. Therefore one can obtain curves whose orders u satisfy (*) and (**) for small D. The near-primes are plentiful enough that one can find curves of nearly prime order with small enough D to construct.

Over $GF(p)$, the CM technique is also called the *Atkin-Morain method* (see [Mor91]); over $GF(2^m)$, it is also called the *Lay-Zimmer method* (see [LZ94]). Although it is possible (over $GF(p)$) to choose the order first and then the field, it is preferable to choose the field first since there are fields in which the arithmetic is especially efficient.

There are two basic steps involved: finding an appropriate order, and constructing a curve having that order. More precisely, one begins by choosing the field size q, the minimum point order r_{min}, and trial division bound l_{max}. Given those quantities, we say that D is *appropriate* if there exists an elliptic curve over $GF(q)$ with CM by D and having nearly prime order.

Step 1 (A.14.2 and A.14.3, Finding a Nearly Prime Order):

> Find an appropriate D. When one is found, record D, the large prime r, and the positive integer k such that $u = kr$ is the nearly prime curve order.

Step 2 (A.14.4 and A.14.5, Constructing a Curve and Point):

> Given D, k and r, construct an elliptic curve over $GF(q)$ and a point of order r.

A.14.2 Finding a Nearly Prime Order over *GF* (*p*)

A.14.2.1 Congruence Conditions

A squarefree positive integer D can be a CM discriminant for p only if it satisfies the following congruence conditions. Let

$$K = \left\lfloor \frac{\left(\sqrt{p}+1\right)^2}{r_{\min}} \right\rfloor .$$

— If $p \equiv 3 \pmod 8$, then $D \equiv 2, 3,$ or $7 \pmod 8$.
— If $p \equiv 5 \pmod 8$, then D is odd.
— If $p \equiv 7 \pmod 8$, then $D \equiv 3, 6,$ or $7 \pmod 8$.
— If $K = 1$, then $D \equiv 3 \pmod 8$.
— If $K = 2$ or 3, then $D \not\equiv 7 \pmod 8$.

Thus the possible squarefree D's are as follows:

If $K = 1$, then

D = 3, 11, 19, 35, 43, 51, 59, 67, 83, 91, 107, 115, ….

If $p \equiv 1 \pmod 8$ and $K = 2$ or 3, then

D = 1, 2, 3, 5, 6, 10, 11, 13, 14, 17, 19, 21, ….

If $p \equiv 1 \pmod 8$ and $K \geq 4$, then

D = 1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17, ….

If $p \equiv 3 \pmod 8$ and $K = 2$ or 3, then

D = 2, 3, 10, 11, 19, 26, 34, 35, 42, 43, 51, 58, ….

If $p \equiv 3 \pmod 8$ and $K \geq 4$, then

D = 2, 3, 7, 10, 11, 15, 19, 23, 26, 31, 34, 35, ….

If $p \equiv 5 \pmod 8$ and $K = 2$ or 3, then

D = 1, 3, 5, 11, 13, 17, 19, 21, 29, 33, 35, 37, ….

If $p \equiv 5 \pmod 8$ and $K \geq 4$, then

D = 1, 3, 5, 7, 11, 13, 15, 17, 19, 21, 23, 29, ….

If $p \equiv 7 \pmod 8$ and $K = 2$ or 3, then

D = 3, 6, 11, 14, 19, 22, 30, 35, 38, 43, 46, 51, ….

If $p \equiv 7 \pmod 8$ and $K \geq 4$, then

D = 3, 6, 7, 11, 14, 15, 19, 22, 23, 30, 31, 35, ….

A.14.2.2 Testing for CM Discriminants (prime case)

Input: a prime p and a squarefree positive integer D satisfying the congruence conditions from A.14.2.1.

Output: if D is a CM discriminant for p, an integer W such that

$$4p = W^2 + DV^2$$

for some V. [In the cases D = 1 or 3, the output also includes V.] If not, the message "not a CM discriminant."

1. Apply the appropriate technique from A.2.5 to find a square root modulo p of $-D$ or determine that none exist.
2. If the result of Step 1 indicates that no square roots exist, then output "not a CM discriminant" and stop. Otherwise, the output of Step 1 is an integer B modulo p.
3. Let $A \leftarrow p$ and $C \leftarrow (B^2 + D) / p$.
4. Let $S \leftarrow \begin{pmatrix} A & B \\ B & C \end{pmatrix}$ and $U \leftarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.
5. Until $|2B| \le A \le C$, repeat the following steps.
   5.1 Let $\delta \leftarrow \left\lfloor \frac{B}{C} + \frac{1}{2} \right\rfloor$.
   5.2 Let $T \leftarrow \begin{pmatrix} 0 & -1 \\ 1 & \delta \end{pmatrix}$.
   5.3 Replace U by $T^{-1}U$.
   5.4 Replace S by $T^t S T$, where T^t denotes the transpose of T.
6. If D = 11 and A = 3, let $\delta \leftarrow 0$ and repeat 5.2, 5.3, 5.4.
7. Let X and Y be the entries of U. That is,

$$U = \begin{pmatrix} X \\ Y \end{pmatrix}.$$

8. If D = 1 or 3 then output $W \leftarrow 2X$ and $V \leftarrow 2Y$ and stop.
9. If A = 1 then output $W \leftarrow 2X$ and stop.
10. If A = 4 then output $W \leftarrow 4X + BY$ and stop.
11. Output "not a CM discriminant."

A.14.2.3 Finding a Nearly Prime Order (prime case)

Input: a prime p, a trial division bound l_{max}, and lower and upper bounds r_{min} and r_{max} for base point order.

Output: a squarefree positive integer D, a prime r in the interval $r_{min} \le r \le r_{max}$, and a smooth integer k such that $u = kr$ is the order of an elliptic curve modulo p with complex multiplication by D.

1. Choose a squarefree positive integer D, not already chosen, satisfying the congruence conditions of A.14.2.1.
2. Compute via A.2.3 the Jacobi symbol $J = \left(\frac{-D}{p} \right)$. If J = –1 then go to Step 1.
3. List the odd primes l dividing D.
4. For each l, compute via A.2.3 the Jacobi symbol $J = \left(\frac{p}{l} \right)$. If J = –1 for some l, then go to Step 1.
5. Test via A.14.2.2 whether D is a CM discriminant for p. If the result is "not a CM discriminant," go to Step 1. (Otherwise, the result is the integer W, along with V if D = 1 or 3.)

6. Compile a list of the possible orders, as follows.
   — If $D = 1$, the orders are

$$p + 1 \pm W,\ p + 1 \pm V.$$

   — If $D = 3$, the orders are

$$p + 1 \pm W,\ p + 1 \pm (W + 3V)/2,\ p + 1 \pm (W - 3V)/2.$$

   — Otherwise, the orders are $p + 1 \pm W$.
7. Test each order for near-primality via A.15.3. If any order is nearly prime, output (D, k, r) and stop.
8. Go to Step 1.

Example: Let $p = 2^{192} - 2^{64} - 1$. Then

$$p = 4X^2 - 2XY + \frac{1+D}{4} Y^2 \quad \text{and} \quad p + 1 - (4X - Y) = r$$

where $D = 235$,

$$X = -3103725293761793083595768723 4,$$
$$Y = 5905046152393184521033305113,$$

and r is the prime

$$r = 627710173538668076383578942333772047398677360825518901 5329.$$

Thus there is a curve modulo p of order r having complex multiplication by D.

A.14.3 Finding a Nearly Prime Order over *GF* (2^m)

A.14.3.1 Testing for CM Discriminants (binary case)

Input: a field degree d and a squarefree positive integer $D \equiv 7 \pmod 8$.

Output: if D is a CM discriminant for 2^d, an odd integer W such that

$$2^{d+2} = W^2 + DV^2,$$

for some odd V. If not, the message "not a CM discriminant."

1. Compute via A.2.6 an integer B such that $B^2 \equiv -D \pmod{2^{d+2}}$.
2. Let $A \leftarrow 2^{d+2}$ and $C \leftarrow (B^2 + D) / 2^{d+2}$.
3. Let $S \leftarrow \begin{pmatrix} A & B \\ B & C \end{pmatrix}$ and $U \leftarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.
4. Until $|2B| \le A \le C$, repeat the following steps.
   4.1 Let $\delta \leftarrow \left\lfloor \frac{B}{C} + \frac{1}{2} \right\rfloor$.
   4.2 Let $T \leftarrow \begin{pmatrix} 0 & -1 \\ 1 & \delta \end{pmatrix}$

4.3 Replace U by $T^{-1}U$.
4.4 Replace S by $T^t S T$, where T^t denotes the transpose of T.

5. Let X and Y be the entries of U. That is,

$$U = \begin{pmatrix} X \\ Y \end{pmatrix}.$$

6. If $A = 1$, then output $W \leftarrow X$ and stop.
7. If $A = 4$ and Y is even, then output $W \leftarrow (4X + BY) / 2$ and stop.
8. Output "not a CM discriminant."

A.14.3.2 Finding a Nearly Prime Order (binary case)

Input: a field degree d, a trial division bound l_{max}, and lower and upper bounds r_{min} and r_{max} for base point order.

Output: a squarefree positive integer D, a prime r in the interval $r_{min} \le r \le r_{max}$, and a smooth integer k such that $u = kr$ is the order of an elliptic curve over $GF(2^d)$ with complex multiplication by D.

1. Choose a squarefree positive integer $D \equiv 7 \pmod 8$, not already chosen.
2. Compute $H \leftarrow$ the class group for D via A.13.2.
3. Set $h \leftarrow$ the number of elements in H.
4. If d does not divide h, then go to Step 1.
5. Test via A.14.3.1 whether D is a CM discriminant for 2^d. If the result is "not a CM discriminant," go to Step 1. (Otherwise, the result is the integer W.)
6. The possible orders are $2^d + 1 \pm W$.
7. Test each order for near-primality via A.15.3. If any order is nearly prime, output (D, k, r) and stop.
8. Go to Step 1.

Example: Let $q = 2^{155}$. Then

$$4q = X^2 + DY^2 \quad \text{and} \quad q + 1 - X = 4r$$

where $D = 942679$, $X = 22952987868304682039818 1$, $Y = -3713607550317790374 97$, and r is the prime

$$r = 1141798154164767904846623037312629032935687344 7.$$

Thus there is a curve over $GF(q)$ of order $4r$ having complex multiplication by D.

A.14.4 Constructing a Curve and Point (prime case)

A.14.4.1 Constructing a Curve with Prescribed CM (prime case)

Given a prime p and a CM discriminant D, the following technique produces an elliptic curve $y^2 \equiv x^3 + a_0 x + b_0 \pmod p$ with CM by D. (Note that there are at least two possible orders among curves with CM by D. The curve constructed here will have the proper CM, but not necessarily the desired order. This curve will be replaced in A.14.4.2 by one of the desired order.)

For nine values of D, the coefficients of E can be written down at once:

D	a_0	b_0

1	1	0
2	–30	56
3	0	1
7	–35	98
11	–264	1694
19	–152	722
43	–3440	77658
67	–29480	1948226
163	–8697680	9873093538

For other values of D, the following algorithm may be used.

Input: a prime modulus p and a CM discriminant $D > 3$for p.

Output: a_0 and b_0 such that the elliptic curve

$$y^2 \equiv x^3 + a_0 x + b_0 \pmod{p}$$

has CM by D.

1. Compute $w(t) \leftarrow w_D(t) \bmod p$ via A.13.3.
2. Let W be the output from A.14.2.2.
3. If W is even, then use A.5.3 with $d = 1$ to compute a linear factor $t - s$ of $w_D(t)$ modulo p. Let

   $$V := (-1)^D 2^{4I/K} s^{24/(GK)} \bmod p,$$

   where G, I and K are as in A.13.3. Finally, let

   $$a_0 := -3(V + 64)(V + 16) \bmod p,$$
   $$b_0 := 2(V + 64)^2 (V - 8) \bmod p.$$

4. If W is odd, then use A.5.3 with $d = 3$ to find a cubic factor $g(t)$ of $w_D(t)$ modulo p. Perform the following computations, in which the coefficients of the polynomials are integers modulo p.

   $$V(t) := \begin{cases} -t^{24} \bmod g(t) & \text{if } 3 \nmid D, \\ -256t^8 \bmod g(t) & \text{if } 3 \mid D, \end{cases}$$

   $$a_1(t) := -3(V(t) + 64)(V(t) + 256) \bmod g(t),$$
   $$b_1(t) := 2(V(t) + 64)^2 (V(t) - 512) \bmod g(t),$$

   $$a_3(t) := a_1(t)^3 \bmod g(t),$$
   $$b_2(t) := b_1(t)^2 \bmod g(t).$$

   Now let σ be a nonzero coefficient from $a_3(t)$, and let τ be the corresponding coefficient from $b_2(t)$. Finally, let

$$a_0 := \sigma\tau \bmod p,$$
$$b_0 := \sigma\tau^2 \bmod p.$$

5. Output (a_0, b_0).

Example: If $D = 235$, then

$$w_D(t) = t^6 - 10\,t^5 + 22\,t^4 - 24\,t^3 + 16\,t^2 - 4\,t + 4.$$

If $p = 2^{192} - 2^{64} - 1$, then

$$w_D(t) \equiv (t^3 - (5 + \phi)t^2 + (1 - \phi)t - 2)\,(t^3 - (5 - \phi)t^2 + (1 + \phi)t - 2) \pmod p,$$

where ϕ = 1254098248316315745658220082226751383299177953632927607231. The resulting coefficients are

$$a_0 = -208902381629407921389227212 8,$$
$$b_0 = -367504956274613540540444576026309668372 48.$$

Thus the curve $y^2 \equiv x^3 + a_0x^2 + b_0$ modulo p has CM by $D = 235$.

A.14.4.2 Choosing the Curve and Point (prime case)

Input: EC parameters p, k, and r, and coefficients a_0, b_0 produced by A.14.4.1.

Output: a curve E modulo p and a point G on E of order r, or a message "wrong order."

1. Select an integer ξ with $0 < \xi < p$.
2. If $D = 1$ then set $a \leftarrow a_0\xi \bmod p$ and $b \leftarrow 0$.
   If $D = 3$ then set $a \leftarrow 0$ and $b \leftarrow b_0\xi \bmod p$.
   Otherwise, set $a \leftarrow a_0\xi^2 \bmod p$ and $b \leftarrow b_0\xi^3 \bmod p$.
3. Look for a point G of order r on the curve

   $$y^2 \equiv x^3 + ax + b \pmod p$$

   via A.11.3.

4. If the output of A.11.3 is "wrong order" then output the message "wrong order" and stop.
5. Output the coefficients a, b and the point G.

The method of selecting ξ in the first step of this algorithm depends on the kind of coefficients desired. Two examples follow.

— If $D \neq 1$ or 3, and it is desired that $a = -3$ (see Annex A.10.4), then take ξ to be a solution of the congruence $a_0\xi^2 \equiv -3 \pmod p$, provided one exists. If one does not exist, or if this choice of ξ leads to the message "wrong order," then select another curve as follows. If $p \equiv 3 \pmod 4$ and the result was "wrong order," then choose $p - \xi$ in place of ξ; the result leads to a curve with $a = -3$ and the right order. If no solution ξ exists, or if $p \equiv 1 \pmod 4$, then repeat A.14.4.1 with another root of the reduced class polynomial. The proportion of roots leading to a curve with $a = -3$ and the right order is roughly one-half if $p \equiv 3 \pmod 4$, and one-quarter if $p \equiv 1 \pmod 4$.

— If there is no restriction on the coefficients, then choose ξ at random. If the output is the message "wrong order," then repeat the algorithm until a set of parameters a, b G is obtained. This will

happen for half the values of ξ, unless $D = 1$ (one-quarter of the values) or $D = 3$ (one-sixth of the values).

A.14.5 Constructing a Curve and Point (binary case)

A.14.5.1 Constructing a Curve with Prescribed CM (binary case)

Input: a field $GF(2^m)$, a CM discriminant D for 2^m, and the desired curve order u.

Output: a and b such that the elliptic curve

$$y^2 + xy = x^3 + ax^2 + b$$

over $GF(2^m)$ has order u.

1. Compute $w(t) \leftarrow w_D(t) \bmod 2$ via A.13.3.
2. Use A.14.3.1 to find the smallest divisor d of m greater than $(\log_2 D) - 2$ such that D is a CM discriminant for 2^d.
3. Compute $p(t)$:= a degree d factor modulo 2 of $w(t)$. (If $d = h$, then $p(t)$ is just $w(t)$ itself. If $d < h$, $p(t)$ is found via A.5.4.)
4. Compute α := a root in $GF(2^m)$ of $p(t) = 0$ via A.5.6.
5. If 3 divides D
   then set $b \leftarrow \alpha$
   else set $b \leftarrow \alpha^3$
6. If u is divisible by 4, then set $a \leftarrow 0$
   else if m is odd, then set $a \leftarrow 1$
   else generate (by trial and error using A.4.5) a random element $a \in GF(2^m)$ of trace 1.
7. Output (a, b).

Example: If $D = 942679$, then

$$w_D(t) \equiv 1 + t^2 + t^6 + t^{10} + t^{12} + t^{13} + t^{16} + t^{17} + t^{20} + t^{22} + t^{24} + t^{27} + t^{30} + t^{33} + t^{35} + t^{36} + t^{37} + t^{41} + t^{42} + t^{43} + t^{45} + t^{49} + t^{51} + t^{54} + t^{56} + t^{57} + t^{59} + t^{61} + t^{65} + t^{67} + t^{68} + t^{69} + t^{70} + t^{71} + t^{72} + t^{74} + t^{75} + t^{76} + t^{82} + t^{83} + t^{87} + t^{91} + t^{93} + t^{96} + t^{99} + t^{100} + t^{101} + t^{102} + t^{103} + t^{106} + t^{108} + t^{109} + t^{110} + t^{114} + t^{117} + t^{119} + t^{121} + t^{123} + t^{125} + t^{126} + t^{128} + t^{129} + t^{130} + t^{133} + t^{134} + t^{140} + t^{141} + t^{145} + t^{146} + t^{147} + t^{148} + t^{150} + t^{152} + t^{154} + t^{155} + t^{157} + t^{158} + t^{160} + t^{161} + t^{166} + t^{167} + t^{171} + t^{172} + t^{175} + t^{176} + t^{179} + t^{180} + t^{185} + t^{186} + t^{189} + t^{190} + t^{191} + t^{192} + t^{195} + t^{200} + t^{201} + t^{207} + t^{208} + t^{209} + t^{210} + t^{211} + t^{219} + t^{221} + t^{223} + t^{225} + t^{228} + t^{233} + t^{234} + t^{235} + t^{237} + t^{238} + t^{239} + t^{241} + t^{242} + t^{244} + t^{245} + t^{248} + t^{249} + t^{250} + t^{252} + t^{253} + t^{255} + t^{257} + t^{260} + t^{262} + t^{263} + t^{264} + t^{272} + t^{273} + t^{274} + t^{276} + t^{281} + t^{284} + t^{287} + t^{288} + t^{289} + t^{290} + t^{292} + t^{297} + t^{299} + t^{300} + t^{301} + t^{302} + t^{304} + t^{305} + t^{306} + t^{309} + t^{311} + t^{312} + t^{313} + t^{314} + t^{317} + t^{318} + t^{320} + t^{322} + t^{323} + t^{325} + t^{327} + t^{328} + t^{329} + t^{333} + t^{335} + t^{340} + t^{341} + t^{344} + t^{345} + t^{346} + t^{351} + t^{353} + t^{354} + t^{355} + t^{357} + t^{358} + t^{359} + t^{360} + t^{365} + t^{366} + t^{368} + t^{371} + t^{372} + t^{373} + t^{376} + t^{377} + t^{379} + t^{382} + t^{383} + t^{387} + t^{388} + t^{389} + t^{392} + t^{395} + t^{398} + t^{401} + t^{403} + t^{406} + t^{407} + t^{408} + t^{409} + t^{410} + t^{411} + t^{416} + t^{417} + t^{421} + t^{422} + t^{423} + t^{424} + t^{425} + t^{426} + t^{429} + t^{430} + t^{438} + t^{439} + t^{440} + t^{441} + t^{442} + t^{443} + t^{447} + t^{448} + t^{450} + t^{451} + t^{452} + t^{453} + t^{454} + t^{456} + t^{458} + t^{459} + t^{460} + t^{462} + t^{464} + t^{465} + t^{466} + t^{467} + t^{471} + t^{473} + t^{475} + t^{476} + t^{481} + t^{482} + t^{483} + t^{484} + t^{486} + t^{487} + t^{488} + t^{491} + t^{492} + t^{495} + t^{496} + t^{498} + t^{501} + t^{503} + t^{505} + t^{507} + t^{510} + t^{512} + t^{518} + t^{519} + t^{529} + t^{531} + t^{533} + t^{536} + t^{539} + t^{540} + t^{541} + t^{543} + t^{545} + t^{546} + t^{547} + t^{548} + t^{550} + t^{552} + t^{555} + t^{556} + t^{557} + t^{558} + t^{559} + t^{560} + t^{563} + t^{565} + t^{566} + t^{568} + t^{580} + t^{585} + t^{588} + t^{589} + t^{591} + t^{592} + t^{593} + t^{596} + t^{597} + t^{602} + t^{604} + t^{606} + t^{610} + t^{616} + t^{620} \pmod 2.$$

This polynomial factors into 4 irreducibles over $GF(2)$, each of degree 155. One of these is

$$p(t) = 1 + t + t^2 + t^6 + t^9 + t^{10} + t^{11} + t^{13} + t^{14} + t^{15} + t^{16} + t^{18} + t^{19} + t^{22} + t^{23} + t^{26} + t^{27} + t^{29} + t^{31} + t^{49} + t^{50} + t^{51} + t^{54} + t^{55} + t^{60} + t^{61} + t^{62} + t^{64} + t^{66} + t^{70} + t^{72} + t^{74} + t^{75} + t^{80} + t^{82} + t^{85} + t^{86} + t^{88} + t^{89} + t^{91} + t^{93} + t^{97} + t^{101} +$$

$t^{103} + t^{104} + t^{111} + t^{115} + t^{116} + t^{117} + t^{118} + t^{120} + t^{121} + t^{123} + t^{124} + t^{126} + t^{127} + t^{128} + t^{129} + t^{130} + t^{131} + t^{132} + t^{134} + t^{136} + t^{137} + t^{138} + t^{139} + t^{140} + t^{143} + t^{145} + t^{154} + t^{155}$.

If t is a root of $p(t)$, then the curve

$$y^2+xy = x^3 + t^3$$

over $GF(2^{155})$ has order $4r$, where r is the prime

$$r = 1141798154164767904846623037312629032935687344 7.$$

A.14.5.2 Choosing the Curve and Point (binary case)

Input: a field size $GF(2^m)$, an appropriate D, the corresponding k and r from A.14.3.2.

Output: a curve E over $GF(2^m)$ and a point G on E of order r.

1. Compute a and b via A.14.5.1 with $u = kr$.
2. Find a point G of order r via A.11.3.
3. Output the coefficients a, b and the point G.

A.15 Primality Tests and Proofs

A.15.1 A Probabilistic Primality Test

If n is a large positive integer, the following probabilistic algorithm (the *strong probable prime test* or the *Miller-Rabin test*) will determine whether n is prime or composite, with arbitrarily small probability of error (see [Knu81]).

Input: an odd integer $n > 2$ to be tested, a positive integer t for the number of trials.

Output: the message “prime” or “composite.”

1. Compute v and odd w such that $n – 1 = 2^v w$.
2. For j from 1 to t do
   - 2.1 Choose random a in the interval $0 < a < n$.
   - 2.2 Set $b \leftarrow a^w \bmod n$.
   - 2.3 If $b = 1$ or $n – 1$ go to Step 2.6.
   - 2.4 For i from 1 to $v – 1$ do
     - 2.4.1 Set $b \leftarrow b^2 \bmod n$.
     - 2.4.2 If $b = n – 1$ go to step 2.6.
     - 2.4.3 If $b = 1$ output "composite" and stop.
     - 2.4.4 Next i.
   - 2.5 Output "composite" and stop.
   - 2.6 Next j.
3. Output “prime.”

If the algorithm outputs “composite,” then n is composite. If the algorithm outputs “prime” then n is almost certainly prime.

Testing random numbers for primality.

If a random *k*-bit integer *n* has tested "prime" after *t* trials of the Miller-Rabin test, then the probability that *n* is composite is at most

$$p_{k,t} = 2^{t+4} k(2^{-\sqrt{tk}})\sqrt{\frac{k}{t}}$$

(provided that $t > 1$ and $k \geq 88$; see [DLP93]).

To achieve a probability of error less than 2^{-100} for a random *k*-bit integer, one should choose the number *t* of trials according to the following list.

K	*T*	*k*	*t*	*K*	*t*
160	34	202-208	23	335-360	12
161-163	33	209-215	22	361-392	11
164-166	32	216-222	21	393-430	10
167-169	31	223-231	20	431-479	9
170-173	30	232-241	19	480-542	8
174-177	29	242-252	18	543-626	7
178-181	28	253-264	17	627-746	6
182-185	27	265-278	16	747-926	5
186-190	26	279-294	15	927-1232	4
191-195	25	295-313	14	1233-1853	3
196-201	24	314-334	13	1854-up	2

The values of *t* in this table can be lowered in many cases; see [DLP93] and [Bur96].

Testing fixed numbers for primality.

In *generating* parameters, one can take random numbers and test them for primality according to the rules above. When *verifying* parameters, however, one cannot treat the putative prime number as random. In this case, the Miller-Rabin test should be run $t = 50$ times in order to achieve a probability of error less than 2^{-100}.

A.15.2 Proving Primality

Another application of the complex multiplication algorithm of A.14 is *proving* the primality of an integer deemed "prime" by a probabilistic test such as A.15.1. The *Goldwasser-Kilian-Atkin algorithm* produces a *primality certificate*: a collection

$$\mathbf{C} = \{C_1,...C_s\}$$

in which each component C_i consists of positive integers

$$C_i = (p_i, r_i, a_i, b_i, x_i, y_i),$$

where:

— For all i, (x_i, y_i) is a point of order r_i on the elliptic curve

$$y^2 \equiv x^3 + a_i x + b_i \pmod{p_i},$$

— $\sqrt{r_i} > \sqrt[4]{p_i} + 1$ for all i,
— $p_1 = n$,
— $p_{i+1} = r_i$ for $1 \le i < s$,
— $r_s < \ell_{\max}^2$,
— r_s is proved prime by trial division.

If a primality certificate exists for n, then n is prime by the following result (see [GK86]):

Let p and r be positive integers greater than 3 with $\sqrt{r} > \sqrt[4]{p} + 1$. Let a, b, x, and y be integers modulo p such that (x, y) is a point of order r on the elliptic curve

$$y^2 \equiv x^3 + ax + b \pmod{p}.$$

Then p is prime if r is.

The GKA algorithm can be implemented as follows.

Input: a large odd positive integer n that has tested "prime" in A.15.1, and a trial division bound $l_{\max}$.

Output: a primality certificate for n, or an error message.

1. Set $\mathbf{C} \leftarrow \{\}$.
2. Set $i \leftarrow 0$.
3. Set $r \leftarrow n$.
4. While $r > \ell_{\max}^2$ do
   - 4.1 Set $i \leftarrow i + 1$.
   - 4.2 Set $p \leftarrow r$.
   - 4.3 Set $r_{\min} \leftarrow (\sqrt[4]{p} + 1)^2$.
   - 4.4 Find positive integers D, k, r (via A.14.2.3) such that
     - — $r \ge r_{\min}$,
     - — r tests "prime" in A.15.1,
     - — kr is an order of an elliptic curve over $GF(p)$ with CM by D.
   - 4.5 Set $C_i \leftarrow (p, r, D, k)$.
   - 4.6 Append C_i to $\mathbf{C}$.
5. $s \leftarrow i$.
6. Confirm primality of r by trial division. (If it is found that r is composite, then output an error message and stop.)
7. For i from 1 to s do
   - 7.1 Recover $(p, r, D, k) \leftarrow C_i$.
   - 7.2 Find via A.14.4.2 a curve

$$E : y^2 \equiv x^3 + ax + b \pmod{p}$$

over $GF(p)$ and a point (x,y) on E of order r.

7.3 Set $C_i \leftarrow (p, r, a, b, x, y)$.

8. Output $\mathbf{C}$.

A.15.3 Testing for Near Primality

Given lower and upper bounds r_{min} and r_{max}, and a trial division bound l_{max}, the following procedures test an integer u for near primality in the sense of A.9.5. Let

$$K := \lfloor u/r_{min} \rfloor.$$

— If $K = 1$, then u is nearly prime if and only if it is prime.

— If $K \geq 2$, then the following algorithm tests u for near primality. Note that one can always take $l_{max} \leq K$.

Input: positive integers u, l_{max}, r_{min}, and r_{max}.

Output: if u is nearly prime, a prime r in the interval $r_{min} \leq r \leq r_{max}$ and a smooth integer k such that $u = kr$. If u is not nearly prime, the message "not nearly prime."

1. Set $r \leftarrow u$, $k \leftarrow 1$.
2. For l from 2 to l_{max} do
   2.1 If l is composite then go to Step 2.3.
   2.2 While (l divides r)
      2.2.1 Set $r \leftarrow r / l$ and $k \leftarrow k\, l$.
      2.2.2 If $r < r_{min}$ then output "not nearly prime" and stop.
   2.3 Next l.
3. If $r > r_{max}$ then output "not nearly prime" and stop.
4. Test r for primality via A.15.1 (and A.15.2 if desired).
5. If r is prime then output k and r and stop.
6. Output "not nearly prime."

A.15.4 Generating Random Primes

The following algorithm generates "random" primes. By this is meant that the algorithm searches numbers of a specified bit length, beginning its search at a randomly chosen starting point.

NOTE—Since the primes are not distributed evenly, some primes are likelier than others to be chosen by this method.

The primes p found by this algorithm also satisfy the condition that $p - 1$ be relatively prime to a specified odd positive integer f. Such a condition is required for generating IF parameters. In the case of RSA, f is set equal to the public exponent. In the case of RW, f is set equal to one-half the public exponent. For applications where such a condition is not needed, one takes $f = 1$, since this choice leads to a vacuous condition.

Input: a lower bound p_{min} for p; an upper bound p_{max} for p; an odd positive integer f.

Output: a random prime p from the range $p_{min} \leq p \leq p_{max}$ for which $p - 1$ is relatively prime to f.

1. Generate a random number p from the range $p_{min} \leq p \leq p_{max}$.
2. If p is even then set $p \leftarrow p + 1$.
3. If $p > p_{max}$ then

3.1 Set $p \leftarrow p - p_{max} + p_{min} - 1$
3.2 Go to Step 2.
4. Compute $d := \text{GCD}(p - 1, f)$ via A.2.2
5. If $d = 1$, then
6. Test p for primality via A.15.1 (and A.15.2 if desired).
5.2 If p is prime, then output p and stop.
6. Set $p \leftarrow p + 2$
7. Go to Step 3.

A.15.5 Generating Random Primes with Congruence Conditions

The following algorithm differs from the preceding one only in that it imposes the additional condition on the prime p that $p \equiv a \pmod{r}$ for some specified a and odd prime r. The remarks made at the beginning of A.15.4 apply here as well.

Input: a prime $r > 2$; a positive integer a not divisible by r; a lower bound p_{min} for p; an upper bound p_{max} for p; an odd positive integer f.

Output: a random prime p from the range $p_{min} \le p \le p_{max}$ satisfying $p \equiv a \pmod{r}$ and for which $p - 1$ is relatively prime to f.

1. Generate a random number p from the range $p_{min} \le p \le p_{max}$.
2. Compute $w := p \bmod 2r$
3. Set $p \leftarrow p + 2r - w + a$
4. If p is even then set $p \leftarrow p + r$.
5. If $p > p_{max}$ then
5.1 Set $p \leftarrow p - p_{max} + p_{min} - 1$
5.2 Go to Step 2.
6. Compute $d := \text{GCD}(p - 1, f)$ via A.2.2.
7. If $d = 1$, then
7.1 Test p for primality via A.15.1 (and A.15.2 if desired).
7.2 If p is prime, then output p and stop.
8. Set $p \leftarrow p + 2r$
9. Go to Step 5.

A.15.6 Strong Primes

A large prime p is called *strong* if

— $p \equiv 1 \pmod{r}$ for some large prime r.
— $p \equiv -1 \pmod{s}$ for some large prime s.
— $r \equiv 1 \pmod{t}$ for some large prime t.

The following algorithm efficiently produces a strong prime p such that $p - 1$ is relatively prime to the specified odd positive integer e. (If the latter condition is not wanted, the choice $e = 1$ leads to a vacuous condition.)

Input: a lower bound p_{min} for p; an upper bound p_{max} for p; the desired bit lengths $L(r)$, $L(s)$, and $L(t)$; an odd positive integer e.

Output: a random strong prime p from the interval $p_{min} \le p \le p_{max}$, where the primes r, s, and t have lengths $L(r)$, $L(s)$, and $L(t)$ respectively, where $p - 1$ is relatively prime to e.

1. Generate a random prime t from the interval $2^{L(t)-1} \le t \le 2^{L(t)} - 1$ using A.15.4 with $f = 1$.
2. Generate a random prime r from the interval $2^{L(r)-1} \le r \le 2^{L(r)} - 1$ satisfying $r \equiv 1 \pmod{t}$ using A.15.5 with $f = 1$.
3. Generate a random prime s from the interval $2^{L(s)-1} \le s \le 2^{L(s)} - 1$ using A.15.4 with $f = 1$.
4. Compute $u := 1/s \bmod r$ via A.2.2.
5. Compute $v := 1/r \bmod s$ via A.2.2.
6. Compute $a \leftarrow su - rv \bmod rs$.
7. Generate a random prime p from the interval $p_{min} \le p \le p_{max}$ satisfying $p \equiv a \pmod{rs}$, using A.15.5 with $f = e$.
8. Output p.

If the strong prime p is also to satisfy an additional congruence condition $p \equiv h \pmod{m}$, then the last two steps of the algorithm should be replaced by the following steps.

7. Compute $b := 1 / (rs) \bmod m$ via A.2.2.
8. Compute $k := 1 / m \bmod rs$ via A.2.2.
9. Compute $c \leftarrow akm + bhrs \bmod mrs$.
10. Generate a random prime p from the interval $p_{min} \le p \le p_{max}$ satisfying $p \equiv c \pmod{mrs}$, using A.15.5 with $f = e$.
11. Output p.

A.16 Generation and Validation of Parameters and Keys

A.16.1 An Algorithm for Generating DL Parameters (prime case)

Input: lower and upper bounds p_{min} and p_{max} for the modulus p; lower and upper bounds r_{min} and r_{max} for the generator order r; whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC.

Output: DL parameters $q = p$, r, and g, satisfying $p_{min} \le p \le p_{max}$ and $r_{min} \le r \le r_{max}$; the cofactor k, if desired

1. Generate a random prime r from the interval $r_{min} \le r \le r_{max}$ using A.15.4 with $f = 1$.
2. Generate a random prime p from the interval $p_{min} \le p \le p_{max}$ satisfying the condition $p \equiv 1 \pmod{r}$, using A.15.5 with $f = 1$.
3. Let $k = (p - 1) / r$.
4. If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, check if r divides k. If so, go to Step 1.
5. Choose an integer h, not already chosen, satisfying $1 < h < p - 1$.
6. Compute

$$g := h^k \bmod p$$

   via A.2.1

7. If $g = 1$ then go to Step 5.
8. Output p, r, g. If desired, also output k.

A.16.2 An Algorithm for Validating DL Parameters (prime case)

Input: DL parameters $q = p$, r, and g; the cofactor k (optional); whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC.

Output: "True" or "False."

1. Check that p is an odd integer and $p > 2$. Check primality of p via A.15.1 with $t = 50$.
2. Check that r is an odd integer and $r > 2$. Check primality of r via A.15.1 with $t = 50$.
3. Check that g is an integer such that $1 < g < p$
4. Check that $g^r \equiv 1 \pmod{p}$
5. If k is supplied, check that k is an integer such that $kr = p - 1$
6. If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, check that r does not divide k (if k is not supplied, first set $k \leftarrow (p - 1)/r$)
7. Output "True" if all checks work, and "False" otherwise.

A.16.3 An Algorithm for Generating DL Parameters (binary case)

Input: lower and upper bounds r_{min} and r_{max} for the generator order r; a list of possible field degrees $m_1, \ldots, m_l$; whether polynomial or normal basis is desired; whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC.

Output: DL parameters $q = 2^m$, r, and g (and the cofactor k, if desired), satisfying $r_{min} \le r \le r_{max}$ and $m = m_i$ for some i, if such exist; otherwise, the message "no such parameters."

1. For i from 1 to l do
   - 1.1 Set $m \leftarrow m_i$
   - 1.2 If m does not appear in the list given in Annex A.3.10, then go to Step 1.5
   - 1.3 Set r to the value in the list that corresponds to m.
   - 1.4 If $r_{min} \le r \le r_{max}$ then go to Step 4.
   - 1.5 Next i
2. For i from 1 to l do
   - 2.1 Set $m \leftarrow m_i$
   - 2.2 Obtain the known primitive prime factors of $2^m - 1$ (see Note 1 below)
   - 2.3 Remove those factors r from the list that are not between r_{min} and r_{max}
   - 2.4. If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, remove those prime factors r from the list for which r^2 divides $2^m - 1$.
   - 2.5 If any factors remain, then let r be one of them and go to Step 4.
   - 2.6 Next i
3. Output the message "no such parameters" and stop.
4. Set $k \leftarrow (2^m - 1)/r$.
5. Choose a polynomial or normal basis for the field $GF(2^m)$ (see Note 2 below)
6. Choose an element h of $GF(2^m)$ not already chosen.
7. Compute $g := h^k$ in $GF(2^m)$ via A.4.3.
8. If $g = 1$ then go to Step 6.
9. Output 2^m, r, g and k.

NOTES

1—See Annex A.3.9 for the definition of a primitive prime factor of $2^m - 1$.

The Cunningham tables are published collections of factors of $2^m - 1$ for m up to 1200, even m up to 2400, and $m \equiv 4 \pmod 8$ up to 4800. They are available in print (see [Cun88]); an up to date version is available via FTP from Oxford University at ftp://sable.ox.ac.uk/pub/math/cunningham/.

If $m \le 1200$ is odd, then the primitive prime factors of $2^m - 1$ are listed in Cunningham table 2–, in the entry $n = m$. If $m \le 2400$ is even but not divisible by 4, then the primitive prime factors of $2^m - 1$ are listed in Cunningham table 2+ (Odd), in the entry $n = m / 2$. If $m \le 4800$ is divisible by 4 but not by 8, then the primitive prime factors of $2^m - 1$ are listed in Cunningham table 2LM, in the two entries for $n = m / 2$. If $m \le 2400$ is divisible by 8, then the primitive prime factors of $2^m - 1$ are listed in Cunningham table 2+ (4k), in the entry $n = m / 2$.

In the FTP version, the last three tables are combined into one, called 2+, but the entries are listed in the same notation.

2—A polynomial basis can be obtained from the basis table given in Annex A.8.1 if $m \leq 1000$. Alternatively, irreducible polynomials over $GF(2)$ can be obtained using the methods given in Annex A.8.2 through A.8.5.

If a normal basis is desired and if $m \leq 1000$ is not divisible by 8, then a Gaussian normal basis can be obtained from the basis table given in Annex A.8.1. Alternatively, normal polynomials over $GF(2)$ can be obtained via Annex A.6.2.

A.16.4 An Algorithm for Validating DL Parameters (binary case)

Input: DL parameters $q = 2^m$, r, and g; the cofactor k (optional); the representation for the elements of $GF(2^m)$; whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC.

Output: "True" or "False."

1. If the representation for the field elements is given by a polynomial basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5.
2. If the representation for the field elements is given by a Gaussian normal basis of type T, check that m is a positive integer not divisible by 8 and that T is a positive integer; check that such a basis exists via A.3.6.
3. If the representation for the field elements is given by a general normal basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5 and that it is normal via A.6.1
4. Check that r is an odd integer and $r > 2$. Check primality of r via A.15.1 with $t = 50$.
5. Check via A.2.7 that the order of 2 modulo r is m (otherwise, it will be less than m, which means that r is not a primitive factor of $2^m - 1$—see Annex A.16.2).
6. Check that g is an element of $GF(2^m)$ and that $g \neq 0$ and $g \neq 1$ in $GF(2^m)$.
7. Check that $g^r = 1$ in $GF(2^m)$.
8. If k is supplied, check that k is an integer such that $kr = 2^m - 1$
9. If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, check that r does not divide k (if k is not supplied, first set $k \leftarrow (2^m - 1)/r$)
10. Output "True" if the checks work, and "False" otherwise.

A.16.5 An Algorithm for Generating DL Keys

Input: valid DL parameters q, r and g

Output: a public key w, a private key s

1. Generate an integer s in the range $0 < s < r$ designed to be hard for an adversary to guess (e.g., a random integer).
2. Compute w by exponentiation: $w := g^s$ in $GF(q)$.
3. Output w and s.

A.16.6 Algorithms for Validating DL Public Keys

The following algorithm verifies if a DL public key is valid.

Input: valid DL parameters q, r and g; the public key w.

Output: "True" or "False."

1. If $q = p$ is an odd prime, check that w is an integer such that $1 < w < p$.

2. If $q = 2^m$ is a power of two, check that w is an element of $GF(2^m)$ and that $w \neq 0$ and $w \neq 1$ in $GF(2^m)$.
3. Check that that $w^r = 1$ in $GF(q)$.
4. Output "True" if the checks work, and "False" otherwise.

The following algorithm does not verify the validity of a DL public key, but merely checks if it is in the multiplicative group of $GF(q)$. It may be used in conjunction with DLSVDP-DHC and DLSVDP-MQVC primitives.

Input: valid DL parameters q, r and g; the public key w.

Output: "True" or "False."

1. If $q = p$ is an odd prime, check that w is an integer such that $0 < w < p$.
2. If $q = 2^m$ is a power of two, check that w is an element of $GF(2^m)$ and that $w \neq 0$ in $GF(2^m)$.
3. Output "True" if the checks work, and "False" otherwise.

A.16.7 An Algorithm for Generating EC Parameters

Input: a field size q (where q is an odd prime p or 2^m); lower and upper bounds r_{min} and r_{max} for the base point order r; a trial division bound l_{max}; the representation for the elements of $GF(2^m)$ if $q = 2^m$ (see Note 2 in A.16.3 for information on methods for choosing a basis for $GF(2^m)$); whether or not the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC .

Output: EC parameters q, a, b, r and G; the cofactor k if desired.

1. Use the techniques of A.14 to find a positive integer k, a prime r in the interval $r_{min} \leq r \leq r_{max}$, a curve E over $GF(q)$ of order kr, and a point G of order r.
2. If q is prime and $q = r$ then the curve is anomalous and subject to the reduction attack described in [Sma98] and [SA98]. Go to Step 1.
3. If the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC, check if r divides k. If so, go to Step 1.
4. Check the MOV condition via A.12.1. If the output is "False" then go to Step 1.
5. Output q, a, b, r, and G (and k if desired).

A.16.8 An Algorithm for Validating EC Parameters

Input: EC parameters q (where q is an odd prime p or 2^m), a, b, r, and G; the cofactor k (optional); the representation for the elements of $GF(2^m)$ if $q = 2^m$; whether the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC.

Output: "True" or "False."

1. If $q = p$, check that p is an odd integer and $p > 2$. Check primality of p via A.15.1 with $t = 50$.
2. If $q = 2^m$ and the representation for the field elements is given by a polynomial basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5.
3. If $q = 2^m$ and representation for the field elements is given by a Gaussian normal basis of type T, check that m is a positive integer not divisible by 8 and that T is a positive integer; check that such a basis exists via A.3.6.
4. If $q = 2^m$ and the representation for the field elements is given by a general normal basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5 and that it is normal via A.6.1
5. Check that r is an odd integer and $r > 2$. Check primality of r via A.15.1 with $t = 50$.

6. If $q = p$, then check that a and b are integers such that $0 \le a < p$ and $0 \le b < p$ and $4a^3 + 27b^2 \bmod p$ is not 0.
7. If $q = 2^m$, then check that a and b are elements of $GF(2^m)$ and that $b \ne 0$ in $GF(2^m)$.
8. Check that $G \ne \mathcal{O}$. Let $G = (x, y)$
9. If $q = p$, check that x and y are integers such that $0 \le x < p$ and $0 \le y < p$ and $y^2 \equiv x^3 + ax + b \pmod p$.
10. If $q = 2^m$, check that x and y are elements of $GF(2^m)$ and that $y^2 + xy = x^3 + ax^2 + b$ in $GF(2^m)$.
11. Check that $rG = \mathcal{O}$.
12. If $q = p$, then check that $r \ne p$ (otherwise, the curve is anomalous and subject to the reduction attack described in [Sma98] and [SA98]; see Annex D.4.2 for more information).
13. If the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC, then
    13.1 If the cofactor k is an input, verify that r does not divide k
    13.2 If the cofactor k is not an input, verify that $r > \sqrt{q} + 1$ (see the Note below)
14. Check the MOV condition via A.12.1.
15. If the cofactor k is an input, then verify it via A.12.3.
16. Output "True" if the checks work, and "False" otherwise.

NOTE—If the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC, it is necessary to verify that r does not divide k. If k is not supplied and $r \le \sqrt{q} + 1$, there is no known simple method to verify that (if $r > \sqrt{q} + 1$, then $k < r$, so r does not divide k). Therefore, in this atypical case, the algorithm outputs "False" even though the parameters may be valid.

A.16.9 An Algorithm for Generating EC Keys

Input: valid EC parameters q, a, b, r, and G.

Output: a public key W, a private key s

1. Generate an integer s in the range $0 < s < r$ designed to be hard for an adversary to guess (e.g., a random integer).
2. Compute the point W by scalar multiplication: $W = sG$.
3. Output W and s.

A.16.10 Algorithms for Validating EC Public Keys

The following algorithm verifies if a EC public key is valid.

Input: valid EC parameters q, a, b, r, G and k such that r does not divide k; a public key W.

Output: "True" or "False."

1. Check that $W \ne \mathcal{O}$. Let $W = (x, y)$
2. If $q = p$, check that x and y are integers such that $0 \le x < p$ and $0 \le y < p$ and $y^2 \equiv x^3 + ax + b \pmod p$.
3. If $q = 2^m$, check that x and y are elements of $GF(2^m)$ and that $y^2 + xy = x^3 + ax^2 + b$ in $GF(2^m)$.
4. Check that $rW = \mathcal{O}$.
5. Output "True" if the checks work, and "False" otherwise.

The following algorithm does not verify the validity of an EC public key, but merely checks if it is a non-identity point on the elliptic curve specified by the parameters. It may be used in conjunction with ECSVDP-DHC and ECSVDP-MQVC primitives.

Input: valid EC parameters q, a, b, r, and G; a public key W.

Output: "True" or "False."

1. Check that $W \neq \mathcal{O}$. Let $W = (x, y)$
2. If $q = p$, check that x and y are integers such that $0 \le x < p$ and $0 \le y < p$ and $y^2 \equiv x^3 + ax + b \pmod{p}$.
3. If $q = 2^m$, check that x and y are elements of $GF(2^m)$ and that $y^2 + xy = x^3 + ax^2 + b$ in $GF(2^m)$.
4. Output "True" if the checks work, and "False" otherwise.

A.16.11 An Algorithm for Generating RSA Keys

An *RSA modulus* is a product n of two (large) primes p and q. Given a public verification (or encryption) exponent e, the following algorithm efficiently produces such an RSA modulus along with the secret signing (or decryption) exponent d.

Common choices for e include the Fermat primes 3, 5, 17, 257, and 65537, since these choices lead to particularly efficient exponentiations using the binary method of A.2.1. If a pseudo-random value of e is desired, it should be generated independently of p and q.

The primes produced may be randomly generated, or they may be strong primes (see Annex A.15.6). A large randomly generated prime is virtually certain to be strong enough in practice.

Input: the desired bit length L for the modulus; an odd public exponent $e > 1$.

Output: an RSA modulus n of bit length L, and the secret exponent d.

1. Generate a prime p from the interval

   $$2^{M-1} \le p \le 2^M - 1$$

   where $M = \lfloor (L + 1)/2 \rfloor$, using A.15.4 with $f = e$ (random) or A.15.6 (strong).

2. Generate a prime q from the interval

   $$\left\lfloor \frac{2^{L-1}}{p} + 1 \right\rfloor \le q \le \left\lfloor \frac{2^L}{p} \right\rfloor$$

   using A.15.4 with $f = e$ (random) or A.15.6 (strong).

3. Set $n := pq$.
4. Compute $l := \text{LCM}(p - 1, q - 1)$ via A.1.1 and A.2.2.
5. Compute $d := e^{-1} \bmod l$ via A.2.2.
6. Output n and d.

A.16.12 An Algorithm for Generating RW Keys

A *Rabin-Williams (RW) modulus* is a product n of two (large) primes $p \equiv 3 \pmod 8$ and $q \equiv 7 \pmod 8$. Given a public verification exponent e, the following algorithm efficiently produces such an RW modulus along with the secret signing exponent d.

The usual choice for the verification exponent is $e = 2$, since this choice means that signature verification can be implemented via a modular squaring rather than a modular exponentiation. If a pseudo-random value of e is desired, it should be generated independently of p and q.

The primes produced may be randomly generated, or they may be strong primes (see Annex A.15.6). A large randomly generated prime is virtually certain to be strong enough in practice.

Input: the desired bit length L for the modulus; an even public exponent e.

Output: an RW modulus n of bit length L, and the secret exponent d.

1. Generate a prime $p \equiv 3 \pmod 8$ from the interval

$$2^{M-1} \le p \le 2^M - 1$$

where $M = \lfloor (L + 1)/2 \rfloor$, using A.15.5 with $f = e$ (random) or A.15.6 (strong).

2. Generate a prime $q \equiv 7 \pmod 8$ from the interval

$$\left\lfloor \frac{2^{L-1}}{p} + 1 \right\rfloor \le q \le \left\lfloor \frac{2^L}{p} \right\rfloor$$

using A.15.5 with $f = e$ (random) or A.15.6 (strong).

3. Set $n := pq$.
4. Compute $l := \text{LCM}(p - 1, q - 1) / 2$ via A.1.1 and A.2.2.
5. Compute $d := e^{-1} \bmod l$ via A.2.2.
6. Output n and d.

[[THIS IS THE END OF THE P1363 ANNEX A]]