Mathematical routines for the NIST prime elliptic curves

April 05, 2010

Contents

1

Disclaimer, Introduction and Notation

DISCLAIMER OF LIABILITY. THE U.S. GOVERNMENT (USG) USED ITS BEST EFFORTS TO VERIFY THE ACCURACY OF THE INFORMATION FURNISHED BY IT HEREUNDER, BUT THE USG SHALL NOT BE LIABLE FOR DAMAGES ARISING OUT OF OR RESULTING FROM ANY COPYRIGHT INFRINGEMENT SUIT, PATENT INFRINGEMENT SUIT, ANY OTHER SUIT, OR ANYTHING MADE AVAILABLE HEREUNDER OR THE USE THEREOF NOR BE LIABLE TO ANY USER OF ANYTHING MADE AVAILABLE HEREUNDER FOR CONSEQUENTIAL DAMAGES UNDER ANY CIRCUMSTANCES.

Described herein are routines for implementing primitives for elliptic curve cryptography on the NIST elliptic curves P–192, P–224, P–256, P–384, and P–521 given in [FIPS186-2]. Also included are specialized routines for field arithmetic over the relevant prime fields and example calculations.

No attempt has been made to present elliptic curves in a mathematically rigorous fashion. For a thorough treatment, the reader is referred to [Sil]. Let $p > 3$ be an odd prime. Let $GF(p)$ be the field of p elements (the integers from 0 to $p-1$ with arithmetic modulo p). For the purposes of this document, an **elliptic curve defined over** $GF(p)$ is the set of pairs (x, y) of elements of $GF(p)$ which satisfy an equation of the form $y^2 = x^3 + ax + b \pmod{p}$, where $a, b \in GF(p)$ are parameters of the curve, along with a so–called "point at infinity". Any two points R, S on an elliptic curve can be added to one another to obtain a third point $R + S$ on the curve. Though not defined here, the addition operation is implemented by the routines in

Section 2.2. Given an integer d and an elliptic curve point S, the result of adding S to itself d times is denoted by dS. Further elaborations are made in Section 2.2.

The NIST curves all have the parameter $a = -3 \equiv p - 3 \pmod{p}$. The routines `ec_decompress` (2.2.4), `ec_is_point_affine` (2.2.5), and `ec_double` (2.2.6) make implicit use of this fact. *T*hroughout this document it is assumed that the elliptic curve parameter a is equivalent to $-3 \pmod{p}$.

Notation:

Lowercase	A scalar: a bit, a word of some prescribed bit-size, or an element of a prime field.
$a\|\|b$	Concatenation of scalars a and b regarded as bitstrings.
Uppercase	A point on an elliptic curve, e.g., R, S, T.
$R + S$	Elliptic curve sum of the points R and S.
$R - S$	Elliptic curve difference of the points R and S.
Subscripted uppercase	A coordinate of a point on an elliptic curve, e.g., R_x, R_y, R_z.
p	An odd prime number > 3.
$GF(p)$	The field of p elements, as represented by the integers $0, 1, \ldots, p - 1$.
n	Bit-length of p, i.e., $\lceil \log_2 p \rceil$.
q	Order of an elliptic curve.
b	Elliptic curve parameter. $y^2 = x^3 - 3x + b$.
p_n	Prime field for NIST elliptic curve P–n.
$t \leftarrow x + y$	Addition modulo p.
$t \leftarrow x - y$	Subtraction modulo p.
$t \leftarrow x \cdot y$	Multiplication modulo p.
$t \leftarrow r^{2^j}$	Successive squaring modulo p.
G	Base point for an elliptic curve group.
$\mathcal{O}$	Point at infinity on an elliptic curve.

2

Common Routines

2.1 Field arithmetic

The ultimate requirement is that multiple-precision arithmetic routines of the appropriate sizes be available for addition, subtraction, multiplication, squaring, comparison, modular reduction, modular inversion, and modular square root. All field operations are done modulo an associated prime, hence support for signed integers is not necessary. Since optimal implementations of these routines will vary over both platform and time, the following descriptions are abstract. Specialized pseudocode for modular reduction and the taking of modular square roots is given in Section 3.

- `mp_set` (r, c): accept the n–bit nonnegative integer c and set r equal to c.

- `mp_add` (r, c, d): accept two n–bit nonnegative integers c, d and set r equal to the (at most) $(n + 1)$–bit nonnegative integer $c + d$.

- `mp_sub` (r, c, d): accept two n–bit nonnegative integers c, d and if $c \geq d$, set r equal to the n–bit nonnegative integer $c - d$, else raise an ERROR.

- `mp_mul` (r, c, d): accept two n–bit nonnegative integers c, d and set r equal to the $2n$–bit nonnegative integer $c \times d$.

- `mp_sqr` (r, c): accept an n–bit nonnegative integer c and set r equal to the $2n$–bit nonnegative integer c^2.

- `mp_cmp` (c, d): accept two n–bit nonnegative integers c, d and return 1 if $c > d$, return 0 if $c = d$, and return -1 if $c < d$.

In each of the elliptic curve routines all arithmetic operations should be performed modulo the associated prime modulus p. A description of the requisite multiple–precision reduction routine is

- `mp_mod` (r, c, m): accept a $2n$–bit nonnegative integer c and an n–bit nonnegative integer m, and set r equal to the n–bit nonnegative integer which is the remainder of the division of c by m, i.e., $c \mod m$.

In this generality an expensive *multiple–precision division* routine is required. However, in the context of this document, modular reductions can always be done with a small number of multiple–precision *additions and/or subtractions*: reducing the result of an addition or subtraction is detailed below, reducing the result of a multiplication (or square) is done with the appropriate routine from Section 3.

Let $0 \leq x, y < p$. Modular addition, $t \leftarrow x + y$, can be implemented as

```
mp_add($t, x, y$)
if mp_cmp($t, p$) $>=$ 0 then
  mp_sub($t, t, p$)
end if
```

modular subtraction, $t \leftarrow x - y$, can be implemented as `mp_sub`(t, p, y), followed by $t \leftarrow t + x$.

Modular inversion can be done efficiently via the binary extended greatest common divisor algorithm, e.g. [HMV, Algorithm 2.22]. The only divisions done are by 2, and hence they can be implemented as right–shifts. It can be used to find an inverse modulo any modulus provided it exists.

2.2 Elliptic curve arithmetic

In the interest of keeping this document somewhat self–contained, routines for basic elliptic curve operations are included. See also [IEEE-P1363, Section A.10].

The routine `ec_twin_mult` (2.2.12) is not currently in any standards document.

Let S be a non–infinite point on an elliptic curve with equation $y^2 = x^3 - 3x + b \pmod{p}$. The **affine representation** of S is a pair $S = (S_x, S_y)$ of elements of $GF(p)$ which satisfy the defining equation.

The **compressed representation** of S, denoted by $\overline{S}$, is the affine x–coordinate, S_x, along with the least significant bit $(S_y)_0$ of the affine y–coordinate, S_y. In this document, the scheme given in [ANSI-X9.63, 4.3.6] is used to represent compression. Given $\overline{S}$, the affine representation of S can be recovered by taking a modular square root (Routine 2.2.4).

A **projective representation** [1] of S is a triple $S = (S_x, S_y, S_z)$ of elements of $GF(p)$, such that $S_y^2 = S_x^3 - 3S_xS_z^4 + bS_z^6 \pmod{p}$. Such a representation is not unique: if $0 \neq \lambda \in GF(p)$, then $(\lambda^2 S_x, \lambda^3 S_y, \lambda S_z)$ is another projective representation. Given an affine representation (S_x, S_y), a convenient projective representation is $(S_x, S_y, 1)$ (see Routine 2.2.1). Given a projective representation (S_x, S_y, S_z) of $S \neq \mathcal{O}$, the affine representation is recovered via Routine 2.2.2 as

$$\text{Affine}(S_x, S_y, S_z) = \left(\frac{S_x}{S_z^2}, \frac{S_y}{S_z^3}\right).$$

There is another point on an elliptic curve, the **point at infinity**, or the **identity**, sometimes denoted by $\mathcal{O}$. This point has no affine representation. In projective coordinates it is any triple of the form $(\lambda^2, \lambda^3, 0)$, where λ is a nonzero element of $GF(p)$. Most often it will be seen as $(1, 1, 0)$. For any point S on an elliptic curve, it is the case that $S - S = 0S = \mathcal{O}$.

An elliptic curve over $GF(p)$ has an **order**, which is the smallest positive integer q such that for any point S, $qS = \mathcal{O}$. The order of each of the NIST elliptic curves is prime. Thus, if $S \neq \mathcal{O}$ is a point on a NIST curve and $0 < d < q$, then $dS \neq \mathcal{O}$. Each NIST curve specifies a **base point**, denoted by G, for use as a public parameter in various cryptographic schemes.

For efficiency reasons (mainly to avoid divisions), most computations on elliptic curve points are done using the projective representation. The following routines will be described.

- `ec_projectify` (R, S): accept an affine point S and set R equal to its projective representation.

- `ec_affinify` (R, S): accept a projective point S and set R equal to its affine representation.

- `ec_compress` (R, S): accept an affine point S and set R equal to its compressed representation, $\overline{S}$.

[1] There are actually many ways to define projective representation on elliptic curves. The one used here gives rise to efficient arithmetic.

- `ec_decompress` $(R, \overline{S})$: accept a compressed point *sbar* and set R equal to its uncompressed representation.
- `ec_is_point_affine` (S): accept an affine point $S = (S_x, S_y)$ and return **true** if it is on the curve, (i.e., if $S_y^2 = S_x^3 - 3S_x + b \pmod{p}$), otherwise return **false**.
- `ec_double` (R, S): accept a projective point S and set R equal to the projective point $2S$. Routine 2.2.6 performs no checks on its inputs.
- `ec_add` (R, S, T): accept two distinct, non–infinite, projective points S, T and set R equal to the projective point $S + T$. Routine 2.2.7 performs no checks on its inputs.
- `ec_full_add` (R, S, T): accept two projective points S, T and set R equal to the projective point $S + T$. Routine 2.2.8 checks whether one of S or T is the point at infinity or whether $S == T$, and if so, takes the appropriate action.
- `ec_full_sub` (R, S, T): accept two projective points S, T and set R equal to the projective point $S - T$. Routine 2.2.9 checks whether one of S or T is the point at infinity or whether $S == T$, and if so, takes the appropriate action.
- `ec_mult` (R, d, S): accept a projective point S, an integer $0 \leq d < p$ and [2] set R equal to the projective point cS.
- `ec_twin_mult` (R, d_0, S, d_1, T): accept two projective points S, T, two integers $0 \leq d_0, d_1 < p$, and set R equal to the projective point $d_0S + d_1T$.

Before proceeding to the routines there are some notes:

1. In the routines `ec_decompress`(2.2.4), `ec_is_point_affine`(2.2.5), and `ec_double`(2.2.6), there are small multiplies, $3t$, $4t$, $8t$. These can be implemented as additions:

$3t = (t + t) + t$	2 additions
$-3t = ((p - t) + (p - t)) + (p - t)$	1 subtraction, 2 additions
$4t = (t + t) + (t + t)$	2 additions
$8t = ((t + t) + (t + t)) + ((t + t) + (t + t))$	3 additions

[2] The NIST curves all have order less than their defining prime p, thus this restriction on the range of d is reasonable in this context.

followed by subtractions to reduce modulo p. The descriptions of `mp_add` and `mp_sub` in Section 2.1 only account for addition or subtraction of two values. It is up to the implementer to resolve any potential overflow problems that could occur.

2. In `ec_add`(2.2.7) line 37 the computation $t_2 = t_2/2$ could be done with Routine 2.1.1. However, it can be done with significantly less effort. If t_2 is even then $t_2/2 \mod p$ is the same as $t_2/2$, i.e., t_2 right-shifted by one bit. If t_2 is odd, then $t_2 + p$ must be even, so one can use the fact that $t_2 \mod p = (t_2 + p) \mod p$, and add p and then right-shift by one bit. Since $t_2 + p < 2p$, it follows that $(t_2 + p)/2 < p$, i.e., the result is already reduced modulo p.

3. In `ec_decompress`(2.2.4) the check in line 5 is necessary if one uses a modular square root routine from Section 3, because such a routine returns a spurious answer when the argument has no modular square root.

4. In `ec_add`(2.2.7) specified triples are sometimes returned. The triple $(1, 1, 0)$ represents the point at infinity $\mathcal{O}$. The triple $(0, 0, 0)$ is used as a marker to indicate that the routine `ec_double`(2.2.6) should be called. The routine `ec_add` cannot be used to add a point to itself because doubling is a different procedure than adding. The routine `ec_full_add`(2.2.8), is designed to handle this case as well as the special case, (not handled by `ec_add`), when one of the two points is the point at infinity $\mathcal{O}$.

5. There are many ways to implement the `ec_mult` routine (2.2.10). See, for instance, [HMV], Section 3.3 or [G].

6. In `ec_mult`(2.2.10) the ensuring that S_z is 1 in lines 10–13 and the calling of `ec_full_add` (or `ec_full_sub`) with the arguments as ordered in lines 20 and 23 allow one to avoid the branch in lines 2–8 of `ec_add`(2.2.7), saving a p–modular square and 4 p–modular multiplies per call.

7. It is common in various schemes to compute a sum of the form $d_0S + d_1T$. The routine `ec_twin_mult` (2.2.12) reduces the amount of computation required for such a calculation to approximately that of a single elliptic curve scalar multiply dS. For details, see [Sol2].

8. In `ec_twin_mult`, it may be worthwhile to affinify and then projectify, (as in 2.2.10, lines 10–13), the points $S, T, S+T, S-T$ in order to save multiplications in the `ec_add` routine, provided it is done carefully. In particular, making four calls to the `ec_affinify` routine is probably *not* cost efficient, due to the cost of the four p–modular inversions (each has cost proportional to n p–modular multiplies). Instead, inverses for the four z–coordinates can be computed with only one p–modular inversion and 11 p–modular multiplies as follows:

$a \leftarrow S_z$	
$b \leftarrow T_z$	
$c \leftarrow (S+T)_z$	
$d \leftarrow (S-T)_z$	
$ab, cd,$	total: 2 multiplies
$abc, abd, acd, bcd,$	total: 6 multiplies
$abcd,$	total: 7 multiplies
$e = (abcd)^{-1}$	total: 7 multiplies and 1 inversion
$a^{-1} = ebcd$	total: 8 multiplies and 1 inversion
$b^{-1} = eacd$	total: 9 multiplies and 1 inversion
$c^{-1} = eabd$	total: 10 multiplies and 1 inversion
$d^{-1} = eabc$	total: 11 multiplies and 1 inversion

Now the inverses can be used to affinify the points at a total cost of 16 p–modular multiplies (4 for each point). Thus the total cost of affinification is 27 p–modular multiplies and 1 field inversion.

Routine 2.1.1 `mp_mod_inv` (r, c, p): Set $r = c^{-1} \pmod p$.

```
$u \leftarrow c, v \leftarrow p$
$x_1 \leftarrow 1, x_2 \leftarrow 0$
while ($u \neq 1$ and $v \neq 1$) do
  while $u$ is even do
    $u \leftarrow u/2$ {can be implemented as a right–shift}
    if $x_1$ is even then
      $x_1 \leftarrow x_1/2$
    else
      $x_1 \leftarrow (x_1 + p)/2$ {do not reduce sum modulo $p$}
    end if
  end while
  while $v$ is even do
    $v \leftarrow v/2$
    if $x_2$ is even then
      $x_2 \leftarrow x_2/2$
    else
      $x_2 \leftarrow (x_2 + p)/2$ {do not reduce sum modulo $p$}
    end if
  end while
  if $u \geq v$ then
    $u \leftarrow u - v, x_1 \leftarrow x_1 - x_2$ {subtractions modulo $p$}
  else
    $v \leftarrow v - u, x_2 \leftarrow x_2 - x_1$ {subtractions modulo $p$}
  end if
end while
if $u == 1$ then
  $r \leftarrow x_1$
else
  $r \leftarrow x_2$
end if
```

Routine 2.2.1 `ec_projectify` (R, S): Create projective R from affine S.

```
$R_x \leftarrow S_x$
$R_y \leftarrow S_y$
$R_z \leftarrow 1$
```

Routine 2.2.2 `ec_affinify` (R, S): Create affine R from projective S.

```
if $S_z == 0$ then
    return ERROR {cannot affinify the point at infinity}
end if
$\lambda \leftarrow S_z^{-1} \mod p$ {use modular inversion routine from Section 3}
$R_x \leftarrow \lambda^2 S_x$
$R_y \leftarrow \lambda^3 S_y$
```

Routine 2.2.3 `ec_compress` (R, S): Set R to compressed $\overline{S}$ from affine S. This implements compression as described in [ANSI-X9.63, 4.3.6].

```
$\overline{R} \leftarrow (2 + (S_y \mod 2)) || S_x$ {|| denotes concatenation of bitstrings.}
```

Routine 2.2.4 `ec_decompress` $(R, \overline{S})$: Create affine R from compressed $\overline{S}$. Access to elliptic curve parameter b is required.

```
$c \leftarrow$ 2 most significant bits of $\overline{S}$
$R_x \leftarrow$ $n$ least significant bits of $\overline{S}$
$t_0 \leftarrow R_x^3 - 3R_x + b \pmod p$
$t_1 \leftarrow \sqrt{t_0} \pmod p$ {use modular square root routine from chapter 3}
if $t_1^2 \neq t_0$ then
    return ERROR {no such point}
end if
if $t_1 == c \pmod 2$ then
    $R_y \leftarrow t_1$
else
    $R_y \leftarrow p - t_1$
end if
```

Routine 2.2.5 `ec_is_point_affine` (S): Check whether the affine point S is on the curve. Access to curve parameter b is required.

```
$t \leftarrow S_x^3 - 3S_x + b \pmod p$
if $S_y^2 == t$ then
    return true
else
    return false
end if
```

Routine 2.2.6 `ec_double` (R, S): Set R to $2S$. Both points projective. Assumes that curve parameter $a \equiv -3 \mod p$

```
$t_1 \leftarrow S_x$
$t_2 \leftarrow S_y$
$t_3 \leftarrow S_z$
if $t_3 == 0$ then
    $R \leftarrow (1, 1, 0)$ and return
end if
$t_4 \leftarrow t_3^2$
$t_5 \leftarrow t_1 - t_4$
$t_4 \leftarrow t_1 + t_4$
$t_5 \leftarrow t_4 \cdot t_5$
$t_4 \leftarrow 3t_5$ {see note 1.}
$t_3 \leftarrow t_3 \cdot t_2$
$t_3 \leftarrow 2t_3$
$t_2 \leftarrow t_2^2$
$t_5 \leftarrow t_1 \cdot t_2$
$t_5 \leftarrow 4t_5$ {see note 1.}
$t_1 \leftarrow t_4^2$
$t_1 \leftarrow t_1 - 2t_5$
$t_2 \leftarrow t_2^2$
$t_2 \leftarrow 8t_2$ {see note 1.}
$t_5 \leftarrow t_5 - t_1$
$t_5 \leftarrow t_4 \cdot t_5$
$t_2 \leftarrow t_5 - t_2$
$R_x \leftarrow t_1$
$R_y \leftarrow t_2$
$R_z \leftarrow t_3$
```

Routine 2.2.7 `ec_add` (R, S, T): Set R to $S + T$. All points projective. Set R to $(0, 0, 0)$ if $S == T$

```
$t_1 \leftarrow S_x$; $t_2 \leftarrow S_y$; $t_3 \leftarrow S_z$; $t_4 \leftarrow T_x$; $t_5 \leftarrow T_y$
if $T_z \neq 1$ then
    $t_6 \leftarrow T_z$
    $t_7 \leftarrow t_6^2$
    $t_1 \leftarrow t_1 \cdot t_7$
    $t_7 \leftarrow t_6 \cdot t_7$
    $t_2 \leftarrow t_2 \cdot t_7$
end if
$t_7 \leftarrow t_3^2$
$t_4 \leftarrow t_4 \cdot t_7$
$t_7 \leftarrow t_3 \cdot t_7$
$t_5 \leftarrow t_5 \cdot t_7$
$t_4 \leftarrow t_1 - t_4$
$t_5 \leftarrow t_2 - t_5$
if $t_4 == 0$ then
    if $t_5 == 0$ then
        $R \leftarrow (0, 0, 0)$ and return
    else
        $R \leftarrow (1, 1, 0)$ and return
    end if
end if
$t_1 \leftarrow 2t_1 - t_4$
$t_2 \leftarrow 2t_2 - t_5$
if $T_z \neq 1$ then
    $t_3 \leftarrow t_3 \cdot t_6$
end if
$t_3 \leftarrow t_3 \cdot t_4$
$t_7 \leftarrow t_4^2$
$t_4 \leftarrow t_4 \cdot t_7$
$t_7 \leftarrow t_1 \cdot t_7$
$t_1 \leftarrow t_5^2$
$t_1 \leftarrow t_1 - t_7$
$t_7 \leftarrow t_7 - 2t_1$
$t_5 \leftarrow t_5 \cdot t_7$
$t_4 \leftarrow t_2 \cdot t_4$
$t_2 \leftarrow t_5 - t_4$
$t_2 \leftarrow t_2/2$ {see note 2.}
$R_x \leftarrow t_1$; $R_y \leftarrow t_2$; $R_z \leftarrow t_3$
```

Routine 2.2.8 `ec_full_add` (R, S, T): Set R to $S{+}T$. All points projective.

```
if $S_z == 0$ then
    $R \leftarrow T$
    return
end if
if $T_z == 0$ then
    $R \leftarrow S$
    return
end if
ec_add $(R, S, T)$
if $R == (0, 0, 0)$ then
    ec_double $(R, S)$
end if
```

Routine 2.2.9 `ec_full_sub` (R, S, T): Set R to $S{-}T$. All points projective.

```
$U \leftarrow T$
$U_y \leftarrow p - U_y$
ec_full_add $(R, S, U)$
```

Routine 2.2.10 `ec_mult` (R, d, S): Set R to dS. All points projective, $0 \leq d < p$.

```
if $d == 0$ then
  $R \leftarrow (1, 1, 0)$ and return
end if
if $d == 1$ then
  $R \leftarrow S$ and return
end if
if $S_z == 0$ then
  $R \leftarrow (1, 1, 0)$ and return
end if
if $S_z \neq 1$ then
  ec_affinify $(T, S)$
  ec_projectify $(S, T)$
end if
$R \leftarrow S$ {keep running total in $R$}
Let $h_l h_{l-1} \dots h_1 h_0$ be the binary representation of $3d$, where $h_l$ is the
  most significant bit of $3d$.
Let $k_l k_{l-1} \dots k_1 k_0$ be the binary representation of $d$.
for $i$ from $l - 1$ to 1 do
  ec_double $(R, R)$
  if $h_i == 1$ and $k_i == 0$ then
    ec_full_add $(U, R, S)$
  end if
  if $h_i == 0$ and $k_i == 1$ then
    ec_full_sub $(U, R, S)$
  end if
  if $h_i \neq k_i$ then
    $R \leftarrow U$
  end if
end for
```

Routine 2.2.11 $F(t)$: an auxilliary function for `ec_twin_mult`

```
if $18 \leq t < 22$ then
    return 9
else if $14 \leq t < 18$ then
    return 10
else if $22 \leq t < 24$ then
    return 11
else if $4 \leq t < 12$ then
    return 14
else
    return 12
end if
```

Routine 2.2.12 `ec_twin_mult` (R, d_0, S, d_1, T): Set R to $d_0S + d_1T$. All points projective, $0 \leq d_0, d_1 < p$.

```
ec_full_add $(SpT, S, T)$
ec_full_sub $(SmT, S, T)$
Let $e_{0,m_0-1}e_{0,m_0-2}\ldots e_{0,0}$ be the binary representation of $d_0$ where $e_{0,m_0-1}$
  is the most significant bit of $d_0$.
Let $e_{1,m_1-1}e_{1,m_1-2}\ldots e_{1,0}$ be the binary representation of $d_1$ where $e_{1,m_1-1}$
  is the most significant bit of $d_1$.
Let $m = \max(m_0, m_1)$. Let $c$ be the $2 \times 6$ binary matrix

  $$c = \begin{bmatrix} 0 & 0 & e_{0,m-1} & e_{0,m-2} & e_{0,m-3} & e_{0,m-4} \\ 0 & 0 & e_{1,m-1} & e_{1,m-2} & e_{1,m-3} & e_{1,m-4} \end{bmatrix}$$

  where $e_{ij} = 0$ whenever $j < 0$ or $j \geq m_i$.
$R \leftarrow \mathcal{O}$
for $k$ from $m$ to 0 by $-1$ do
  for $i$ from 0 to 1 do
    $h_i \leftarrow 16c_{i,1} + 8c_{i,2} + 4c_{i,3} + 2c_{i,4} + c_{i,5}$
    if $c_{i,0} == 1$ then
      $h_i \leftarrow 31 - h_i$
    end if
  end for
  for $i$ from 0 to 1 do
    $$u_i \leftarrow \begin{cases} 0 & : \text{ if } h_i < F(h_{1-i}) \text{ (see 2.2.11)} \\ (-1)^{c_{i,0}} & : \text{ otherwise} \end{cases}$$
  end for
  $$c \leftarrow \begin{bmatrix} (|u_0| \texttt{ xor } c_{0,1}) & c_{0,2} & c_{0,3} & c_{0,4} & c_{0,5} & e_{0,k-5} \\ (|u_1| \texttt{ xor } c_{1,1}) & c_{1,2} & c_{1,3} & c_{1,4} & c_{1,5} & e_{1,k-5} \end{bmatrix}$$
  ec_double $(R, R)$
  if $(u_0 == -1)$ and $(u_1 == -1)$ then ec_full_sub $(R, R, SpT)$
  if $(u_0 == -1)$ and $(u_1 == 0)$ then ec_full_sub $(R, R, S)$
  if $(u_0 == -1)$ and $(u_1 == 1)$ then ec_full_sub $(R, R, SmT)$
  if $(u_0 == 0)$ and $(u_1 == -1)$ then ec_full_sub $(R, R, T)$
  if $(u_0 == 0)$ and $(u_1 == 1)$ then ec_full_add $(R, R, T)$
  if $(u_0 == 1)$ and $(u_1 == -1)$ then ec_full_add $(R, R, SmT)$
  if $(u_0 == 1)$ and $(u_1 == 0)$ then ec_full_add $(R, R, S)$
  if $(u_0 == 1)$ and $(u_1 == 1)$ then ec_full_add $(R, R, SpT)$
end for
```

3

Special Routines for NIST Primes

3.1 Notes

For each given bitsize $n = 192, 224, 256, 384, 521$ pseudocode is provided for specialized field arithmetic. This specialization is possible because the NIST primes are of a particular form. [1]

The modular square root routines given here are based on exponentiation. [2] They use the fact that there are long strings of consecutive 1's and 0's in the binary representations of the NIST primes to significantly cut down the number of multiplies in an exponentiation. They are simple to implement and reasonably efficient. Note well that they will compute spurious values if the argument passed does not actually have a modular square root. This necessitates the checking of the computed value, e.g., Routine 2.2.4, lines 5–7.

In the modular reduction routines the following notation is used. Given a $2n$–bit number a, it can be written as a sum of powers of 2^e, with e–bit coefficients:

$$a = \sum_{i=0}^{d-1} a_i \cdot 2^{ei} \quad (0 \leq a_i < 2^e).$$

Table 3.1 gives the values of d and e used in this section. In all cases $\|$ is

[1] They are instances of so–called generalized Mersenne numbers [Sol1].

[2] Except for square roots modulo p_{224}. This is discussed at the end of this section.

Table 3.1: Values of d and e for the NIST bitsizes

n	d	e
192	6	64
224	14	32
256	16	32
384	24	32
521	1042	1

used as a concatenation operator, and one writes

$$a = (a_{d-1} \| a_{d-2} \| \cdots \| a_1 \| a_0).$$

All comments which describe an assignment of a variable to a computed value are modulo the associated prime. For instance, in line 6 of Routine 3.2.10 the comment

$$\{r = c^{2^{64}-2^{32}+1}\}$$

should be implicitly read as $r = c^{2^{64}-2^{32}+1} \pmod{p_{256}}$.

The notation $t \leftarrow r^{2^j}$ is used to denote successive modular squarings, i.e.,

```
$t \leftarrow r$
**for** $i = 1$ to $j$ **do**
  $t \leftarrow t \cdot t$ {multiplication mod $p$}
**end for**
```

In the modular reduction routines 3.2.1 line 6, 3.2.3 line 9, 3.2.9 line 15, and 3.2.11 line 15, several values are added before reduction modulo the associated prime via several subtractions. The descriptions of `mp_add` and `mp_sub` in 2.1 only account for addition or subtraction of *two* values. It is up to the implementer to resolve any possible overflow issues.

Routines 3.2.4, 3.2.5, 3.2.6, and 3.2.7 are all auxilliary to 3.2.8, which is used to compute square roots modulo p_{224}. The method is based on an algorithm of Pocklington described in [Ber]. [3]

[3] While the method is theoretically non-deterministic, the practical probability of failure is negligible.

The routine for reducing modulo p_{256}, 3.2.9, adds two multiples of p_{256} in the case $d_1 > p_{256}$ or $d_2 > p_{256}$. The probability of such happening is about 2^{-32}.

3.2 Routines

Routine 3.2.1 `mp_mod_192` (r, a): Set $r = a \pmod{p_{192}}$

```
{Note: the $a_i$ are 64–bit quantities.}
$t \leftarrow ( a_2 \parallel a_1 \parallel a_0 )$
$s_1 \leftarrow ( 0 \parallel a_3 \parallel a_3 )$
$s_2 \leftarrow ( a_4 \parallel a_4 \parallel 0 )$
$s_3 \leftarrow ( a_5 \parallel a_5 \parallel a_5 )$
$r \leftarrow t + s_1 + s_2 + s_3$
Reduce $r$ mod $p_{192}$ by subtraction of up to *three* multiples of $p_{192}$.
```

Routine 3.2.2 `mp_mod_sqrt_192` (r, c): Set $r = \sqrt{c} \pmod{p_{192}}$

```
$t_1 \leftarrow c^2; t_1 \leftarrow t_1 \cdot c$            $\{t_1 = c^{2^2-1}\}$
$t_2 \leftarrow t_1^{2^2}; t_2 \leftarrow t_2 \cdot t_1$          $\{t_2 = c^{2^4-1}\}$
$t_3 \leftarrow t_2^{2^4}; t_3 \leftarrow t_3 \cdot t_2$          $\{t_3 = c^{2^8-1}\}$
$t_4 \leftarrow t_3^{2^8}; t_4 \leftarrow t_4 \cdot t_3$          $\{t_4 = c^{2^{16}-1}\}$
$t_5 \leftarrow t_4^{2^{16}}; t_5 \leftarrow t_5 \cdot t_4$         $\{t_5 = c^{2^{32}-1}\}$
$t_6 \leftarrow t_5^{2^{32}}; t_6 \leftarrow t_6 \cdot t_5$         $\{t_6 = c^{2^{64}-1}\}$
$r \leftarrow t_6^{2^{64}}; r \leftarrow r \cdot t_6$           $\{r = c^{2^{128}-1}\}$
$r \leftarrow r^{2^{62}}$        $\{r = c^{2^{190}-2^{62}} = \sqrt{c} \pmod{p_{192}}\}$
```

Routine 3.2.3 `mp_mod_224` (r, a): Set $r = a \pmod{p_{224}}$

```
{Note: the $a_i$ are 32–bit quantities.}
$t \leftarrow ( a_6 \| a_5 \| a_4 \| a_3 \| a_2 \| a_1 \| a_0 )$
$s_1 \leftarrow ( a_{10} \| a_9 \| a_8 \| a_7 \| 0 \| 0 \| 0 )$
$s_2 \leftarrow ( 0 \| a_{13} \| a_{12} \| a_{11} \| 0 \| 0 \| 0 )$
$d_1 \leftarrow ( a_{13} \| a_{12} \| a_{11} \| a_{10} \| a_9 \| a_8 \| a_7 )$
$d_2 \leftarrow ( 0 \| 0 \| 0 \| 0 \| a_{13} \| a_{12} \| a_{11} )$
$d_1 \leftarrow p_{224} - d_1$
$r \leftarrow t + s_1 + s_2 + d_1 - d_2$
Reduce $r$ mod $p_{224}$ by subtraction of up to *three* multiples of $p_{224}$.
```

Routine 3.2.4 `RS` $(d_1, e_1, f_1, d_0, e_0, f_0)$: Set d_1, e_1, f_1 where $d_1 + e_1 x = (d_0 + e_0 x)^2 \pmod{x^2 + c}$ and $f_1 = -e_1^2 c \pmod{p_{224}}$

```
Require: $f_0 = -e_0^2 c \pmod{p_{224}}$
$t \leftarrow d_0^2$
$e_1 \leftarrow d_0 \cdot e_0$
$d_1 \leftarrow t + f_0$
$e_1 \leftarrow e_1 + e_1$
$f_1 \leftarrow t \cdot f_0$
$f_1 \leftarrow f_1 + f_1$
$f_1 \leftarrow f_1 + f_1$
```

Routine 3.2.5 `RSS` $(d_1, e_1, f_1, d_0, e_0, f_0, j)$: Set d_1, e_1, f_1 where $d_1 + e_1 x = (d_0 + e_0 x)^{2^j} \pmod{x^2 + c}$ and $f_1 = -e_1^2 c \pmod{p_{224}}$

```
$d_1 \leftarrow d_0, e_1 \leftarrow e_0, f_1 \leftarrow f_0$
for $i$ in $[1..j]$ do
    RS $(d_1, e_1, f_1, d_1, e_1, f_1)$
end for
```

Routine 3.2.6 RM $(d_2, e_2, f_2, c, d_0, e_0, d_1, e_1)$: Set d_2, e_2, f_2 where $d_2 + e_2x = (d_0 + e_0x)(d_1 + e_1x) \pmod{x^2 + c}$ and $f_2 = -e_2^2c \pmod{p_{224}}$

```
$t_1 \leftarrow e_0 \cdot e_1$
$t_1 \leftarrow t_1 \cdot c$
$t_1 \leftarrow p_{224} - t_1$
$t_2 \leftarrow d_0 \cdot d_1$
$t_2 \leftarrow t_2 + t_1$
$t_1 \leftarrow d_0 \cdot e_1$
$e_2 \leftarrow d_1 \cdot e_0$
$e_2 \leftarrow e_2 + t_1$
$f_2 \leftarrow e_2^2$
$f_2 \leftarrow f_2 \cdot c$
$f_2 \leftarrow p_{224} - f_2$
$d_2 \leftarrow t_2$
```

Routine 3.2.7 RP (d_1, e_1, f_1, c, r): Set d_1, e_1, f_1 where $d_1 + e_1x = (r + x)^{2^{128}-1} \pmod{x^2 + c}$ and $f_1 = -e_1^2c \pmod{p_{224}}$

```
$d_0 \leftarrow r$, $e_0 \leftarrow 1$, $f_0 \leftarrow p_{224} - c$
for $i$ in [0..6] do
    RSS $(d_1, e_1, f_1, d_0, e_0, f_0, 2^i)$
    RM $(d_1, e_1, f_1, c, d_1, e_1, d_0, e_0)$
    $d_0 \leftarrow d_1$, $e_0 \leftarrow e_1$, $f_0 \leftarrow f_1$
end for
```

Routine 3.2.8 `mp_mod_sqrt_224` (r, c): Set $r = \sqrt{c} \pmod{p_{224}}$

```
Let $s$ be a random number modulo $p_{224}$.

RP $(d_0, e_0, f_0, c, s)$          $\{d_0 + e_0 x = (s + x)^{2^{128}-1} \pmod{x^2 + c}$
                                    $f_0 = -e_0^2 c \pmod{p_{224}}\}$

RS $(d_1, e_1, f_1, d_0, e_0, f_0)$  $\{d_1 + e_1 x = (d_0 + e_0 x)^2 \pmod{x^2 + c}$
                                    $f_1 = -e_1^2 c \pmod{p_{224}}\}$
for $i$ in [1..95] do
  $d_0 \leftarrow d_1, e_0 \leftarrow e_1, f_0 \leftarrow f_1$
  RS $(d_1, e_1, f_1, d_0, e_0, f_0)$
  if $d_1$ == 0 then
    break
  end if
end for
$r \leftarrow d_0 / e_0$                      {Use mp_mod_inv to invert $e_0$.}
```

Routine 3.2.9 `mp_mod_256` (r, a): Set $r = a \pmod{p_{256}}$

```
{Note: the $a_i$ are 32–bit quantities.}
 $t$   ← ( $a_7$    || $a_6$    || $a_5$    || $a_4$    || $a_3$    || $a_2$    || $a_1$    || $a_0$    )
 $s_1$ ← ( $a_{15}$ || $a_{14}$ || $a_{13}$ || $a_{12}$ || $a_{11}$ || 0        || 0        || 0        )
 $s_2$ ← ( 0        || $a_{15}$ || $a_{14}$ || $a_{13}$ || $a_{12}$ || 0        || 0        || 0        )
 $s_3$ ← ( $a_{15}$ || $a_{14}$ || 0        || 0        || 0        || $a_{10}$ || $a_9$    || $a_8$    )
 $s_4$ ← ( $a_8$    || $a_{13}$ || $a_{15}$ || $a_{14}$ || $a_{13}$ || $a_{11}$ || $a_{10}$ || $a_9$    )
 $d_1$ ← ( $a_{10}$ || $a_8$    || 0        || 0        || 0        || $a_{13}$ || $a_{12}$ || $a_{11}$ )
 $d_2$ ← ( $a_{11}$ || $a_9$    || 0        || 0        || $a_{15}$ || $a_{14}$ || $a_{13}$ || $a_{12}$ )
 $d_3$ ← ( $a_{12}$ || 0        || $a_{10}$ || $a_9$    || $a_8$    || $a_{15}$ || $a_{14}$ || $a_{13}$ )
 $d_4$ ← ( $a_{13}$ || 0        || $a_{11}$ || $a_{10}$ || $a_9$    || 0        || $a_{15}$ || $a_{14}$ )
$d_1 \leftarrow 2p_{256} - d_1$
$d_2 \leftarrow 2p_{256} - d_2$
$d_3 \leftarrow p_{256} - d_3$
$d_4 \leftarrow p_{256} - d_4$
$r \leftarrow t + 2\,s_1 + 2\,s_2 + s_3 + s_4 + d_1 + d_2 + d_3 + d_4$
Reduce $r$ mod $p_{256}$ by subtraction of up to *ten* multiples of $p_{256}$.
```

Routine 3.2.10 `mp_mod_sqrt_256` (r, c): Set $r = \sqrt{c} \pmod{p_{256}}$

```
$t_1 \leftarrow c^2; t_1 \leftarrow t_1 \cdot c$          $\{t_1 = c^{2^2-1}\}$
$t_2 \leftarrow t_1^{2^2}; t_2 \leftarrow t_2 \cdot t_1$          $\{t_2 = c^{2^4-1}\}$
$t_3 \leftarrow t_2^{2^4}; t_3 \leftarrow t_3 \cdot t_2$          $\{t_3 = c^{2^8-1}\}$
$t_4 \leftarrow t_3^{2^8}; t_4 \leftarrow t_4 \cdot t_3$          $\{t_4 = c^{2^{16}-1}\}$
$r \leftarrow t_4^{2^{16}}; r \leftarrow r \cdot t_4$          $\{r = c^{2^{32}-1}\}$
$r \leftarrow r^{2^{32}}; r \leftarrow r \cdot c$          $\{r = c^{2^{64}-2^{32}+1}\}$
$r \leftarrow r^{2^{96}}; r \leftarrow r \cdot c$          $\{r = c^{2^{160}-2^{128}+2^{96}+1}\}$
$r \leftarrow r^{2^{94}}$          $\{r = c^{2^{254}-2^{222}+2^{190}+2^{94}} = \sqrt{c} \pmod{p_{256}}\}$
```

Routine 3.2.11 `mp_mod_384` (r, a): Set $r = a \pmod{p_{384}}$

```
{Note: the $a_i$ are 32–bit quantities.}
$t \leftarrow (\, a_{11} \| a_{10} \| a_9 \| a_8 \| a_7 \| a_6 \| a_5 \| a_4 \| a_3 \| a_2 \| a_1 \| a_0 \,)$
$s_1 \leftarrow (\, 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| 0 \| 0 \| 0 \| 0 \,)$
$s_2 \leftarrow (\, a_{23} \| a_{22} \| a_{21} \| a_{20} \| a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \,)$
$s_3 \leftarrow (\, a_{20} \| a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \| a_{23} \| a_{22} \| a_{21} \,)$
$s_4 \leftarrow (\, a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \| a_{20} \| 0 \| a_{23} \| 0 \,)$
$s_5 \leftarrow (\, 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| a_{20} \| 0 \| 0 \| 0 \| 0 \,)$
$s_6 \leftarrow (\, 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| 0 \| 0 \| a_{20} \,)$
$d_1 \leftarrow (\, a_{22} \| a_{21} \| a_{20} \| a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \| a_{23} \,)$
$d_2 \leftarrow (\, 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| a_{20} \| 0 \,)$
$d_3 \leftarrow (\, 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{23} \| 0 \| 0 \| 0 \,)$
$d_1 \leftarrow p_{384} - d_1$
$r \leftarrow t + 2\, s_1 + s_2 + s_3 + s_4 + s_5 + s_6 + d_1 - d_2 - d_3$
Reduce $r$ mod $p_{384}$ by subtraction of up to *four* multiples of $p_{384}$.
```

Routine 3.2.12 `mp_mod_sqrt_384` (r, c): Set $r = \sqrt{c} \pmod{p_{384}}$

```
$t_1 \leftarrow c^2; t_1 \leftarrow t_1 \cdot c$    $\{t_1 = c^{2^2-1}\}$
$t_2 \leftarrow t_1^{2^2}; t_2 \leftarrow t_2 \cdot t_1$    $\{t_2 = c^{2^4-1}\}$
$t_2 \leftarrow t_2^2; t_2 \leftarrow t_2 \cdot c$    $\{t_2 = c^{2^5-1}\}$
$t_3 \leftarrow t_2^{2^5}; t_3 \leftarrow t_3 \cdot t_2$    $\{t_3 = c^{2^{10}-1}\}$
$t_4 \leftarrow t_3^{2^5}; t_4 \leftarrow t_4 \cdot t_2$    $\{t_4 = c^{2^{15}-1}\}$
$t_2 \leftarrow t_4^{2^{15}}; t_2 \leftarrow t_2 \cdot t_4$    $\{t_2 = c^{2^{30}-1}\}$
$t_3 \leftarrow t_2^{2^2}$    $\{t_3 = c^{2^{32}-4}\}$
$t_1 \leftarrow t_3 \cdot t_1$    $\{t_1 = c^{2^{32}-1}\}$
$t_3 \leftarrow t_3^{2^{28}}; t_2 \leftarrow t_2 \cdot t_3$    $\{t_3 = c^{2^{60}-2^{30}}, t_2 = c^{2^{60}-1}\}$
$t_3 \leftarrow t_2^{2^{60}}; t_3 \leftarrow t_3 \cdot t_2$    $\{t_3 = c^{2^{120}-1}\}$
$r \leftarrow t_3^{2^{120}}; r \leftarrow r \cdot t_3$    $\{r = c^{2^{240}-1}\}$
$r \leftarrow r^{2^{15}}; r \leftarrow r \cdot t_4$    $\{r = c^{2^{255}-1}\}$
$r \leftarrow r^{2^{33}}; r \leftarrow r \cdot t_1$    $\{r = c^{2^{288}-2^{32}-1}\}$
$r \leftarrow r^{2^{64}}; r \leftarrow r \cdot c$    $\{r = c^{2^{352}-2^{96}-2^{64}+1}\}$
$r \leftarrow r^{2^{30}}$    $\{r = c^{2^{382}-2^{126}-2^{94}+2^{30}} = \sqrt{c} \pmod{p_{384}}\}$
```

Routine 3.2.13 `mp_mod_521` (r, a): Set $r = a \pmod{p_{521}}$

```
{Note: the $a_i$ are 1–bit quantities.}
$t \leftarrow (a_{520} \| a_{519} \| a_{518} \| \ldots \| a_2 \| a_1 \| a_0)$
$s \leftarrow (a_{1041} \| a_{1040} \| a_{1039} \| \ldots \| a_{523} \| a_{522} \| a_{521})$
$r \leftarrow t + s$
Reduce $r$ by subtraction of up to *one* multiple of $p_{521}$.
```

Routine 3.2.14 `mp_mod_sqrt_521` (r, c): Set $r = \sqrt{c} \pmod{p_{521}}$

```
$r \leftarrow c$    $\{r = c\}$
for $i = 1$ to 519 do
    $r \leftarrow r^2$
end for    $\{r = c^{2^{519}} = \sqrt{c} \pmod{p_{521}}\}$
```

4

Curves and Example Data

4.1 Curve P–192

4.1.1 Parameters

The curve P–192 is given by the following parameters (see also [FIPS186-2]). The prime $p_{192} = 2^{192} - 2^{64} - 1$:

$p_{192} =$ `6277101735386680763835789423207666416083\`
`908700390324961279`

in hexadecimal form:

$p_{192} =$ `ffffffff ffffffff ffffffff fffffffe ffffffff ffffffff`

The parameter $a = p_{192} - 3$:

$a =$ `6277101735386680763835789423207666416083\`
`908700390324961276`

in hexadecimal form:

$a =$ `ffffffff ffffffff ffffffff fffffffe ffffffff fffffffc`

the parameter b:

$b =$ `2455155546008943817740293915197451784769\`
`108058161191238065`

in hexadecimal form:

$b =$ `64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1`

Base point G:

$x_G =$ `6020462823756886567582134805875261119166\`
`98976636884684818`

$y_G =$ `1740503322936220314048575522802194103640\`
`23488927386650641`

in hexadecimal form:

$x_G =$ `188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012`

$y_G =$ `07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811`

in hexadecimal form with X9.63 compression (lead byte `02` if y_G is even, `03` if y_G is odd):

$\overline{G} =$ `00000003 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd`
`82ff1012`

Order q of the point G (and of the elliptic curve group E):

$q =$ `6277101735386680763835789423176059013767\`
`194773182842284081`

hexadecimal form:

$q =$ `ffffffff ffffffff ffffffff 99def836 146bc9b1 b4d22831`

4.1.2 Example calculations

We give the results of basic calculations on a pair of points S and T using affine coordinates. S has coordinates

$x_S =$ `d458e7d1 27ae671b 0c330266 d2467693 53a01207 3e97acf8`

$y_S =$ `32593050 0d851f33 6bddc050 cf7fb11b 5673a164 5086df3b`

and T has coordinates:

$x_T =$ `f22c4395 213e9ebe 67ddecdd 87fdbd01 be16fb05 9b9753a4`

$y_T =$ 26442409 6af2b359 7796db48 f8dfb41f a9cecc97 691a9c79

Full add $R = S + T$:

$x_R =$ 48e1e409 6b9b8e5c a9d0f1f0 77b8abf5 8e843894 de4d0290

$y_R =$ 408fa77c 797cd7db fb16aa48 a3648d3d 63c94117 d7b6aa4b

Full subtract $R = S - T$:

$x_R =$ fc9683cc 5abfb4fe 0cc8cc3b c9f61eab c4688f11 e9f64a2e

$y_R =$ 093e31d0 0fb78269 732b1bd2 a73c23cd d31745d0 523d816b

Double $R = 2S$:

$x_R =$ 30c5bc6b 8c7da253 54b373dc 14dd8a0e ba42d25a 3f6e6962

$y_R =$ 0dde14bc 4249a721 c407aedb f011e2dd bbcb2968 c9d889cf

Scalar multiply $R = dS$:

$d \ =$ a78a236d 60baec0c 5dd41b33 a542463a 8255391a f64c74ee

$x_R =$ 1faee420 5a4f669d 2d0a8f25 e3bcec9a 62a69529 65bf6d31

$y_R =$ 5ff2cdfa 508a2581 89236708 7c696f17 9e7a4d7e 8260fb06

Joint scalar multiply $R = dS + eT$ (d as above):

$e \ =$ c4be3d53 ec3089e7 1e4de8ce ab7cce88 9bc393cd 85b972bc

$x_R =$ 019f64ee d8fa9b72 b7dfea82 c17c9bfa 60ecb9e1 778b5bde

$y_R =$ 16590c5f cd8655fa 4ced33fb 800e2a7e 3c61f35d 83503644

4.2 Curve P–224

4.2.1 Parameters

The curve P–224 is given by the following parameters (see also [FIPS186-2]). The prime $p_{224} = 2^{224} - 2^{96} + 1$:

$p_{224} =$ `26959946667150639794667015087019630673557\`
`916260026308143510066298881`

in hexadecimal form:

$p_{224} =$ `ffffffff ffffffff ffffffff ffffffff 00000000 00000000`
`00000001`

The parameter $a = p_{224} - 3$:

$a =$ `26959946667150639794667015087019630673557\`
`916260026308143510066298878`

in hexadecimal form:

$a =$ `ffffffff ffffffff ffffffff fffffffe ffffffff ffffffff`
`fffffffe`

the parameter b:

$b =$ `18958286285566608000408668544493926415504\`
`680968679321075787234672564`

in hexadecimal form:

$b =$ `b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943`
`2355ffb4`

Base point G:

$x_G =$ `19277929113566293071110308034699488026831\`
`934219452440156649784352033`

$y_G =$ `19926808758034470970197974370888749184205\`
`991990603949537637343198772`

in hexadecimal form:

$x_G =$ `b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6`
`115c1d21`

$y_G =$ `bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199`
`85007e34`

in hexadecimal form with X9.63 compression (lead byte `02` if y_G is even, `03` if y_G is odd):

$\overline{G} =$ `00000002 b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122`
`343280d6 115c1d21`

Order q of the point G (and of the elliptic curve group E):

$q =$ `26959946667150639794667015087019625940457\`
`807714424391721682722368061`

hexadecimal form:

$q =$ `ffffffff ffffffff ffffffff ffff16a2 e0b8f03e 13dd2945`
`5c5c2a3d`

4.2.2 Example calculations

We give the results of basic calculations on a pair of points S and T using affine coordinates. S has coordinates

$x_S =$ `6eca814b a59a9308 43dc814e dd6c97da 95518df3 c6fdf16e`
`9a10bb5b`

$y_S =$ `ef4b497f 0963bc8b 6aec0ca0 f259b89c d8099414 7e05dc6b`
`64d7bf22`

and T has coordinates:

$x_T =$ `b72b25ae a5cb03fb 88d7e842 00296964 8e6ef23c 5d39ac90`
`3826bd6d`

$y_T =$ `c42a8a4d 34984f0b 71b5b409 1af7dceb 33ea729c 1a2dc8b4`
`34f10c34`

Full add $R = S + T$:

$x_R =$ `236f26d9 e84c2f7d 776b107b d478ee0a 6d2bcfca a2162afa`
`e8d2fd15`

$y_R =$ `e53cc0a7 904ce6c3 746f6a97 471297a0 b7d5cdf8 d536ae25`
`bb0fda70`

Full subtract $R = S - T$:

$x_R =$ `db4112bc c8f34d4f 0b36047b ca1054f3 61541385 2a793133 5210b332`

$y_R =$ `90c6e830 4da48138 78c1540b 2396f411 facf787a 520a0ffb 55a8d961`

Double $R = 2S$:

$x_R =$ `a9c96f21 17dee0f2 7ca56850 ebb46efa d8ee2685 2f165e29 cb5cdfc7`

$y_R =$ `adf18c84 cf77ced4 d76d4930 417d9579 207840bf 49bfbf58 37dfdd7d`

Scalar multiply $R = dS$:

$d =$ `a78ccc30 eaca0fcc 8e36b2dd 6fbb03df 06d37f52 711e6363 aaf1d73b`

$x_R =$ `96a7625e 92a8d72b ff1113ab db95777e 736a14c6 fdaacc39 2702bca4`

$y_R =$ `0f8e5702 942a3c5e 13cd2fd5 80191525 8b43dfad c70d15db ada3ed10`

Joint scalar multiply $R = dS + eT$ (d as above):

$e =$ `54d549ff c08c9659 2519d73e 71e8e070 3fc8177f a88aa77a 6ed35736`

$x_R =$ `dbfe2958 c7b2cda1 302a67ea 3ffd94c9 18c5b350 ab838d52 e288c83e`

$y_R =$ `2f521b83 ac3b0549 ff4895ab cc7f0c5a 861aacb8 7acbc5b8 147bb18b`

4.3 Curve P–256

4.3.1 Parameters

The curve P–256 is given by the following parameters (see also [FIPS186-2]). The prime $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$:

$p_{256} =$ `115792089210356248762697446949407573530086\`
`143415290314195533631308867097853951`

in hexadecimal form:

$p_{256} =$ `ffffffff 00000001 00000000 00000000 00000000 ffffffff`
`ffffffff ffffffff`

The parameter $a = p_{256} - 3$:

$a =$ `115792089210356248762697446949407573530086\`
`143415290314195533631308867097853948`

in hexadecimal form:

$a =$ `ffffffff 00000001 00000000 00000000 00000000 ffffffff`
`ffffffff fffffffc`

the parameter b:

$b =$ `410583637251521421293261297800472684091\`
`14441015993725554835256314039467401291`

in hexadecimal form:

$b =$ `5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6`
`3bce3c3e 27d2604b`

Base point G:

$x_G =$ `484395612939064517590525852527979142027\`
`62949526041747995844080717082404635286`

$y_G =$ `361342509567497957985851279195878819566\`
`11106672985015071877198253568414405109`

in hexadecimal form:

$x_G =$ `6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0`
`f4a13945 d898c296`

$y_G =$ 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece
cbb64068 37bf51f5

in hexadecimal form with X9.63 compression (lead byte 02 if y_G is even, 03 if y_G is odd):

$\overline{G} =$ 00000003 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81
2deb33a0 f4a13945 d898c296

Order q of the point G (and of the elliptic curve group E):

$q =$ 115792089210356248762697446949407573529\
96955224135760342422259061068512044369

hexadecimal form:

$q =$ ffffffff 00000000 ffffffff ffffffff bce6faad a7179e84
f3b9cac2 fc632551

4.3.2 Example calculations

We give the results of basic calculations on a pair of points S and T using affine coordinates. S has coordinates

$x_S =$ de2444be bc8d36e6 82edd27e 0f271508 617519b3 221a8fa0
b77cab39 89da97c9

$y_S =$ c093ae7f f36e5380 fc01a5aa d1e66659 702de80f 53cec576
b6350b24 3042a256

and T has coordinates:

$x_T =$ 55a8b00f 8da1d44e 62f6b3b2 5316212e 39540dc8 61c89575
bb8cf92e 35e0986b

$y_T =$ 5421c320 9c2d6c70 4835d82a c4c3dd90 f61a8a52 598b9e7a
b656e9d8 c8b24316

Full add $R = S + T$:

$x_R =$ 72b13dd4 354b6b81 745195e9 8cc5ba69 70349191 ac476bd4
553cf35a 545a067e

$y_R =$ 8d585cbb 2e1327d7 5241a8a1 22d7620d c33b1331 5aa5c9d4
6d013011 744ac264

Full subtract $R = S - T$:

$x_R =$ c09ce680 b251bb1d 2aad1dbf 6129deab 837419f8 f1c73ea1
3e7dc64a d6be6021

$y_R =$ 1a815bf7 00bd8833 6b2f9bad 4edab172 3414a022 fdf6c3f4
ce30675f b1975ef3

Double $R = 2S$:

$x_R =$ 7669e690 1606ee3b a1a8eef1 e0024c33 df6c22f3 b17481b8
2a860ffc db6127b0

$y_R =$ fa878162 187a54f6 c39f6ee0 072f33de 389ef3ee cd03023d
e10ca2c1 db61d0c7

Scalar multiply $R = dS$:

$d \ =$ c51e4753 afdec1e6 b6c6a5b9 92f43f8d d0c7a893 3072708b
6522468b 2ffb06fd

$x_R =$ 51d08d5f 2d427888 2946d88d 83c97d11 e62becc3 cfc18bed
acc89ba3 4eeca03f

$y_R =$ 75ee68eb 8bf626aa 5b673ab5 1f6e744e 06f8fcf8 a6c0cf30
35beca95 6a7b41d5

Joint scalar multiply $R = dS + eT$ (d as above):

$e \ =$ d37f628e ce72a462 f0145cbe fe3f0b35 5ee8332d 37acdd83
a358016a ea029db7

$x_R =$ d867b467 92210092 34939221 b8046245 efcf5841 3daacbef
f857b858 8341f6b8

$y_R =$ f2504055 c03cede1 2d22720d ad69c745 106b6607 ec7e50dd
35d54bd8 0f615275

4.4 Curve P–384

4.4.1 Parameters

The curve P–384 is given by the following parameters (see also [FIPS186-2]). The prime $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$:

$p_{384} =$ 394020061963944792122790401001436138050797392704654\
4466679482934042457217714968703290472660882589380\
01861606973112319

in hexadecimal form:

$p_{384} =$ ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff fffffffe ffffffff 00000000 00000000 ffffffff

The parameter $a = p_{384} - 3$:

$a =$ 394020061963944792122790401001436138050797392704654\
4466679482934042457217714968703290472660882589380\
01861606973112316

in hexadecimal form:

$a =$ ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff fffffffe ffffffff 00000000 00000000 fffffffc

the parameter b:

$b =$ 275801935599597058778490118403890480930569058563615\
6852142870730198868924130986086513626076488374510\
7765439761230575

in hexadecimal form:

$b =$ b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112
0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

Base point G:

$x_G =$ 262470350957996892686231567445669818918529234911092\
1338781561590092551885473805008902238805397571978\
6650872476732087

$y_G =$ 832571096148902998554675128952010817928785304886131\
5594709205902480503199884419224438643760392947333\
078086511627871

in hexadecimal form:

$x_G =$ `aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98`
`59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760aB7`

$y_G =$ `3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c`
`e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5F`

in hexadecimal form with X9.63 compression (lead byte `02` if y_G is even, `03` if y_G is odd):

$\overline{G} =$ `00000003 aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62`
`8ba79b98 59f741e0 82542a38 5502f25d bf55296c 3a545e38`
`72760ab7`

Order q of the point G (and of the elliptic curve group E):

$q =$ `39402006196394479212279040100143613805079739270465\`
`44666794690527962765939911326356939895630815229491\`
`3554433653942643`

hexadecimal form:

$q =$ `ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff`
`c7634d81 f4372ddf 581a0db2 48b0a77a ecec196a ccc52973`

4.4.2 Example calculations

We give the results of basic calculations on a pair of points S and T using affine coordinates. S has coordinates

$x_S =$ `fba203b8 1bbd23f2 b3be971c c23997e1 ae4d89e6 9cb6f923`
`85dda827 68ada415 ebab4167 459da98e 62b1332d 1e73cb0e`

$y_S =$ `5ffedbae fdeba603 e7923e06 cdb5d0c6 5b223014 29293376`
`d5c6944e 3fa6259f 162b4788 de6987fd 59aed5e4 b5285e45`

and T has coordinates:

$x_T =$ `aacc0520 2e7fda6f c73d82f0 a6622052 7da8117e e8f8330e`
`ad7d20ee 6f255f58 2d8bd38c 5a7f2b40 bcdb68ba 13d81051`

$y_T =$ `84009a26 3fefba7c 2c57cffa 5db3634d 286131af c0fca8d2`
`5afa22a7 b5dce0d9 470da892 33cee178 592f49b6 fecb5092`

Full add $R = S + T$:

$x_R =$ `12dc5ce7 acdfc584 4d939f40 b4df012e 68f865b8 9c3213ba`
`97090a24 7a2fc009 075cf471 cd2e85c4 89979b65 ee0b5eed`

$y_R =$ `167312e5 8fe0c0af a248f285 4e3cddcb 557f983b 3189b67f`
`21eee013 41e7e9fe 67f6ee81 b36988ef a406945c 8804a4b0`

Full subtract $R = S - T$:

$x_R =$ `6afdaf8d a8b11c98 4cf177e5 51cee542 cda4ac2f 25cd522d`
`0cd710f8 8059c656 5aef78f6 b5ed6cc0 5a6666de f2a2fb59`

$y_R =$ `7bed0e15 8ae8cc70 e847a603 47ca1548 c348decc 6309f48b`
`59bd5afc 9a9b804e 7f787617 8cb5a7eb 4f6940a9 c73e8e5e`

Double $R = 2S$:

$x_R =$ `2a2111b1 e0aa8b2f c5a19755 16bc4d58 017ff96b 25e1bdff`
`3c229d5f ac3bacc3 19dcbec2 9f9478f4 2dee597b 4641504c`

$y_R =$ `fa2e3d9d c84db895 4ce8085e f28d7184 fddfd134 4b4d4797`
`343af9b5 f9d83752 0b450f72 6443e411 4bd4e5bd b2f65ddd`

Scalar multiply $R = dS$:

$d \ =$ `a4ebcae5 a6659834 93ab3e62 6085a24c 104311a7 61b5a8fd`
`ac052ed1 f111a5c4 4f76f456 59d2d111 a61b5fdd 97583480`

$x_R =$ `e4f77e7f feb7f095 8910e3a6 80d677a4 77191df1 66160ff7`
`ef6bb526 1f791aa7 b45e3e65 3d151b95 dad3d93c a0290ef2`

$y_R =$ `ac7dee41 d8c5f4a7 d5836960 a773cfc1 376289d3 373f8cf7`
`417b0c62 07ac32e9 13856612 fc9ff2e3 57eb2ee0 5cf9667f`

Joint scalar multiply $R = dS + eT$ (d as above):

$e \ =$ `afcf8811 9a3a76c8 7acbd600 8e1349b2 9f4ba9aa 0e12ce89`
`bcfcae21 80b38d81 ab8cf150 95301a18 2afbc689 3e75385d`

$x_R =$ `917ea28b cd641741 ae5d18c2 f1bd917b a68d34f0 f0577387`
`dc812604 62aea60e 2417b8bd c5d954fc 729d211d b23a02dc`

$y_R =$ `1a29f7ce 6d074654 d77b4088 8c73e925 46c8f16a 5ff6bcbd`
`307f758d 4aee684b eff26f67 42f597e2 585c86da 908f7186`

4.5 Curve P–521

4.5.1 Parameters

The curve P–521 is given by the following parameters (see also [FIPS186-2]). The prime $p_{521} = 2^{521} - 1$:

$p_{521} =$

```
6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151
```

in hexadecimal form:

$p_{521} =$

```
000001ff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff
```

The parameter $a = p_{521} - 3$:

$a =$

```
6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057148
```

in hexadecimal form:

$a =$

```
000001ff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff fffffffc
```

the parameter b:

$b =$

```
1093849038073734274511112390766805569936207598951683748994586394495953116150735016013708737573759623248592132296706313309438452531591012912142327488478985984
```

in hexadecimal form:

$b =$

```
00000051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b
99b315f3 b8b48991 8ef109e1 56193951 ec7e937b 1652c0bd
3bb1bf07 3573df88 3d2c34f1 ef451fd4 6b503f00
```

Base point G:

$x_G =$ 26617408020502170632287687167233609607298591687569
73147706671368418802944996427808491545080627771902
35209424122506555866215711354557091681416163731589
5999846

$y_G =$ 37571800257700204635455072244911836035944551347697
62486694567779615544477440556316691234405012945539
56214444453728942852258566672919658081012434427757
8376784

in hexadecimal form:

$x_G =$ 000000c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139
053fb521 f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127
a2ffa8de 3348b3c1 856a429b f97e7e31 c2e5bd66

$y_G =$ 00000118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449
579b4468 17afbd17 273e662c 97ee7299 5ef42640 c550b901
3fad0761 353c7086 a272c240 88be9476 9fd16650

in hexadecimal form with X9.63 compression (lead byte 02 if y_G is even, 03 if y_G is odd):

$G =$ 02c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139 053fb521
f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127 a2ffa8de
3348b3c1 856a429b f97e7e31 c2e5bd66

Order q of the point G (and of the elliptic curve group E):

$q =$ 68647976601306097149819007990813932172694353001433
05409394463459185543183397655394245057746333217197
53296399637136332111386476861244038034037280889270
7005449

hexadecimal form:

$q =$ 000001ff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff fffffffa 51868783 bf2f966b 7fcc0148
f709a5d0 3bb5c9b8 899c47ae bb6fb71e 91386409

4.5.2 Example calculations

We give the results of basic calculations on a pair of points S and T using affine coordinates. S has coordinates

$x_S =$
```
000001d5 c693f66c 08ed03ad 0f031f93 7443458f 601fd098
d3d0227b 4bf62873 af50740b 0bb84aa1 57fc847b cf8dc16a
8b2b8bfd 8e2d0a7d 39af04b0 89930ef6 dad5c1b4
```

$y_S =$
```
00000144 b7770963 c63a3924 8865ff36 b074151e ac33549b
224af5c8 664c5401 2b818ed0 37b2b7c1 a63ac89e baa11e07
db89fcee 5b556e49 764ee3fa 66ea7ae6 1ac01823
```

and T has coordinates:

$x_T =$
```
000000f4 11f2ac2e b971a267 b80297ba 67c322db a4bb21ce
c8b70073 bf88fc1c a5fde3ba 09e5df6d 39acb2c0 762c03d7
bc224a3e 197feaf7 60d63240 06fe3be9 a548c7d5
```

$y_T =$
```
000001fd f842769c 707c93c6 30df6d02 eff399a0 6f1b36fb
9684f0b3 73ed0648 89629abb 92b1ae32 8fdb4553 42683849
43f0e922 2afe0325 9b32274d 35d1b958 4c65e305
```

Full add $R = S + T$:

$x_R =$
```
00000126 4ae115ba 9cbc2ee5 6e6f0059 e24b52c8 04632160
2c59a339 cfb757c8 9a59c358 a9a8e1f8 6d384b3f 3b255ea3
f73670c6 dc9f45d4 6b6a196d c37bbe0f 6b2dd9e9
```

$y_R =$
```
00000062 a9c72b8f 9f88a271 690bfa01 7a6466c3 1b9cadc2
fc544744 aeb81707 2349cfdd c5ad0e81 b03f1897 bd9c8c6e
fbdf6823 7dc3bb00 445979fb 373b20c9 a967ac55
```

Full subtract $R = S - T$:

$x_R =$
```
00000129 2cb58b17 95ba4770 63fef7cd 22e42c20 f57ae94c
eaad86e0 d21ff229 18b0dd3b 076d63be 253de24b c20c6da2
90fa54d8 3771a225 deecf914 9f79a8e6 14c3c4cd
```

$y_R =$
```
00000169 5e3821e7 2c7cacaa dcf62909 cd83463a 21c6d033
93c527c6 43b36239 c46af117 ab7c7ad1 9a4c8cf0 ae95ed51
72988546 1aa2ce27 00a6365b ca3733d2 920b2267
```

Double $R = 2S$:

$x_R =$ 00000128 79442f24 50c119e7 119a5f73 8be1f1eb a9e9d7c6
cf41b325 d9ce6d64 3106e9d6 1124a91a 96bcf201 305a9dee
55fa7913 6dc70083 1e54c3ca 4ff2646b d3c36bc6

$y_R =$ 00000198 64a8b885 5c2479cb efe375ae 553e2393 271ed36f
adfc4494 fc0583f6 bd035988 96f39854 abeae5f9 a6515a02
1e2c0eef 139e71de 610143f5 3382f410 4dccb543

Scalar multiply $R = dS$:

$d =$ 000001eb 7f81785c 9629f136 a7e8f8c6 74957109 73555411
1a2a866f a5a16669 9419bfa9 936c78b6 2653964d f0d6da94
0a695c72 94d41b2d 6600de6d fcf0edcf c89fdcb1

$x_R =$ 00000091 b15d09d0 ca0353f8 f96b93cd b13497b0 a4bb582a
e9ebefa3 5eee61bf 7b7d041b 8ec34c6c 00c0c067 1c4ae063
318fb75b e87af4fe 859608c9 5f0ab477 4f8c95bb

$y_R =$ 00000130 f8f8b5e1 abb4dd94 f6baaf65 4a2d5810 411e77b7
423965e0 c7fd79ec 1ae563c2 07bd255e e9828eb7 a03fed56
5240d2cc 80ddd2ce cbb2eb50 f0951f75 ad87977f

Joint scalar multiply $R = dS + eT$ (d as above):

$e =$ 00000137 e6b73d38 f153c3a7 57561581 2608f2ba b3229c92
e21c0d1c 83cfad92 61dbb17b b77a6368 2000031b 9122c2f0
cdab2af7 2314be95 254de429 1a8f85f7 c70412e3

$x_R =$ 0000009d 3802642b 3bea152b eb9e05fb a247790f 7fc16807
2d363340 133402f2 585588dc 1385d40e bcb8552f 8db02b23
d687cae4 6185b275 28adb1bf 9729716e 4eba653d

$y_R =$ 0000000f e44344e7 9da6f49d 87c10637 44e5957d 9ac0a505
bafa8281 c9ce9ff2 5ad53f8d a084a2de b0923e46 501de579
7850c61b 229023dd 9cf7fc7f 04cd35eb b026d89d

Bibliography

[ANSI-X9.63] American National Standard for Financial Services ANSI X9.63–2001, *Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, November 2001.

[G] D.M. Gordon, "A Survey of Fast Exponentiation Methods", Journal of Algorithms, 27, 1998, pp 129–146

[Sol1] Solinas, J.A, "Generalized Mersenne Numbers", Tech. Report Centre for Applied Cryptographic Research, 1999, `http://www.cacr.math.uwaterloo.ca/techreports/1999/corr99-39.ps`

[Sol2] Solinas, J.A, "Low-Weight Binary Representations for Pairs of Integers", Tech. Report Centre for Applied Cryptographic Research, 2001, `http://www.cacr.math.uwaterloo.ca/techreports/2001/corr-41.ps`

[IEEE-P1363] IEEE STD 1363-2000, Standard Specifications for Public–Key Cryptrography (Annex A), Institute of Electrical and Electronics Engineers, Inc., August 2000.

[FIPS186-2] Digital Signature Standard (DSS). Federal Information Standards Processing Publication 186-2, National Institute of Standards and Technology, 2000

[HMV] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag, New York, 2004.

[Ber] D. J. Bernstein, "Faster Square Roots in Annoying Finite Fields", Draft aimed at the Mathematics of Computation (2001), `http://cr.yp.to/papers/sqroot.pdf`

[Sil] Silverman, Joseph, H., *The Arithmetic of Elliptic Curves*, Springer-Verlag, New York, 1986.