EIP1962

algorithms and protocols

Alex Vlasov
Matter Labs
email: av@matterlabs.dev

Konstantin Panarin
Matter Labs
email: kp@matterlabs.dev

February 10, 2020

This document is a part of a series of specification documents for EIP 1962 and describes an implementation of arithmetic and guarantees it's consistency.

1 Finite Field Algorithms.

Main challenge for consistent arithmetic is no guarantee for primarity of modulus p. For that reason there is a check (described in ABI document) that $p > 3$, $p = 1 \ mod \ 2$ that ensures that at least Montgomery form is consistent as it requires $gcd(p, R) = 1$ where $R = 2^{64n}$ in this implementation (Montgomery form operates on n limbs of 64 bits). Another document in a series will describe gas metering that takes numbers of 64 bit words in representation of various big integer parameters as input. This does not mean that any 3rd party implementation can not use smaller limbs (e.g. 32 bits), but it still would have to follow the gas metering spec for 64 bit words.

There are few edge cases and conventions that are decisions of the spec writers, those are described in a sections 5 and 6.

Referenced implementation performs all field arithmetic in Montgomery representation with 64 bit limbs for efficiency reasons and under assumption that miners would run a modern x64 hardware.

Some algorithms described in this spec use static single assignment (SSA) form to help implementers.

Algorithm 1

Montgomery Inverse

Input: a, p, n, where p is odd, $p > a > 0$, and n is the number of bits in p. Division by two is Euclidean division (bit shift)

Output: "Not relatively prime" if $gcd(a,p) \neq 1$ or $a^{-1}2^n \mod p$

```
First phase
u ← p, v ← a, r ← 0, s ← 1
k ← 0
while u > 0 do
  if u mod 2 ≡ 0 then
    u ← u/2, s ← 2s
  else if v mod 2 ≡ 0 then
    v ← v/2, r ← 2r
  else if u > v then
    u ← (u−v)/2, r ← r + s, s ← 2s
  else
    v ← (v−u)/2, s ← r + s, r ← 2r
  end if
  k ← k + 1
end while
if u ≠ 1 then
  return "Not relatively prime"
end if
if r ≥ p then
  r ← r − p
end if
Second phase
for i=1 to k-n do
  if r mod 2 ≡ 0 then
    r = r/2
  else
    r = (r + a)/2
  end if
end for
return a − r
```

This algorithm takes $O(logp) = O(n)$ steps.

Note: there is a variant of Montgomery inversion algorithm based on extended Euclidean algorithm. However, this algorithm doesn't terminate if p is not prime, although there is a deterministic estimate for number of cycles for prime p.

Reference implementation deals with extensions of degree 2 and 3 only, generated by polynomials of the form $X^k - a$, where $a \in \mathbb{F}_q$ and k is 2 (3 respectively) and a $a \in \mathbb{F}_q$ in a quadratic (cubic) non-residue in $\mathbb{F}_q$.

We sometimes require extensions of degree 6 and 12: these extensions are generated by composition of extension 2 and 3. "Final" steps in extension towers have fixed structure like $w^2 - v = 0$, where u is a non-residue to construct previous extension using polynomial $v^3 - u = 0$. Such constructions are most common and widely used in popular and efficient curves.

Example: F_p^{12} as extension 2 over F_p^6, that is itself extension 3 over F_p^2. This would require u to be a 6th power non-residue. Let's use F_p^2 element u to construct F_p^6 using polynomial $v^3 - u = 0$. During some operation in F_p^{12} we would need to perform multiplication of F_p^6 by non-residue that would generate F_p^{12} from F_p^6. This is performed as the following: take an element in F_p^6 that is 3 over 2 and multiply by non-residue: $(c_0 + c_1 v + c_2 v^2)v$ with $v^3 - u = 0$ results in $(c_2 u + c_0 v + c_1 v^2)$. Total extension tower would look like: $z^2 - \xi = 0$ where ξ is an element of F_p, $v^3 - u = 0$ where u is an element of F_p^2, $w^2 - v = 0$.

In what follows we represent $\mathbb{F}_{p^2}$ as $\mathbb{F}_p[u]/(u^2 - \beta)$.

Algorithm 2
Field multiplication in $\mathbb{F}_{p^2}$

Input: $a = a_0 + a_1 u, b = b_0 + b_1 u \in \mathbb{F}_{p^2}$
Output: $c = a \cdot b \in \mathbb{F}_{p^2}$
  $v_0 \leftarrow a_0 b_0$
  $v_1 \leftarrow a_1 b_1$
  $c_0 \leftarrow v_0 + \beta v_1$
  $c_1 \leftarrow (a_0 + a_1)(b_0 + b_1) - v_0 - v_1$
  return $c = c_0 + c_1 u$

Algorithm 3
Field squaring in $\mathbb{F}_{p^2}$

Input: $a = a_0 + a_1 u \in \mathbb{F}_{p^2}$
Output: $c = a^2 \in \mathbb{F}_{p^2}$
$v_0 \leftarrow a_0 - a_1, v_3 \leftarrow a_0 - \beta a_1$
$v_2 \leftarrow a_0 a_1$
$v_0 \leftarrow v_0 v_3 + v_2$
$c_1 \leftarrow 2v_2$
$c_0 \leftarrow v_0 + \beta v_2$
return $c = c_0 + c_1 u$

Algorithm 4
Inversion in $\mathbb{F}_{p^2}$

Input: $a = a_0 + a_1 u \in \mathbb{F}_{p^2}$
Output: $c = a^{-1} \in \mathbb{F}_{p^2}$
$v_0 \leftarrow a_0^2, v_1 \leftarrow a_1^2$
$v_0 \leftarrow v_0 - \beta v_1$
$v_1 \leftarrow v_0^{-1}$
$c_0 \leftarrow a_0 v_1, c_1 \leftarrow -a_1 v_1$
return $c = c_0 + c_1 u$

Now we are going to consider cubic extensions: $\mathbb{F}_{p^3} = \mathbb{F}_p[w]/(w^3 - \alpha)$.

Algorithm 5
Multiplication in the cubic extension $\mathbb{F}_{p^3}$

Input: $a = (a_0 + a_1 w + a_2 w^2), b = (b_0 + b_1 w + b_2 w^2) \in \mathbb{F}_{p^3}$
Output: $c = a \cdot b \in \mathbb{F}_{p^3}$
$v_0 \leftarrow a_0 b_0, v_1 \leftarrow a_1 b_1, v_2 \leftarrow a_2 b_2$
$c_0 \leftarrow ((a_1 + a_2)(b_1 + b_2) - v_1 - v_2)\alpha + v_0$
$c_1 \leftarrow (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 - \alpha v_2$
$c_2 \leftarrow (a_0 + a_2)(b_0 + b_2) - v_0 - v_2 + v_1$
return $c = (c_0 + c_1 w + c_2 w^2)$

Algorithm 6

Squaring in cubic extension $\mathbb{F}_{p^3}$

Input: $a = (a_0 + a_1 w + a_2 w^2) \in \mathbb{F}_{p^3}$

Output: $c = a^2 \in \mathbb{F}_{p^3}$

$v_0 \leftarrow a_0 b_0$

$v_1 \leftarrow a_1 b_1$

$v_2 \leftarrow a_2 b_2$

$c_0 \leftarrow ((a_1 + a_2)(b_1 + b_2) - v_1 - v_2)\alpha + v_0$

$c_1 \leftarrow (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 - \alpha v_2$

$c_2 \leftarrow (a_0 + a_2)(b_0 + b_2) - v_0 - v_2 + v_1$

return $c = (c_0 + c_1 w + c_2 w^2)$

Algorithm 7

Inversion in cubic extension $\mathbb{F}_{p^3}$

Input: $a = (a_0 + a_1 w + a_2 w^2) \in \mathbb{F}_{p^3}$

Output: $c = a^{-1} \in \mathbb{F}_{p^3}$

$v_0 \leftarrow a_0^2$

$v_1 \leftarrow a_1^2$

$v_2 \leftarrow a_2^2$

$v_3 \leftarrow a_0 a_1$

$v_4 \leftarrow a_0 a_2$

$v_5 \leftarrow a_1 a_2$

$A \leftarrow v_0 - \alpha v_5$

$B \leftarrow \alpha v_2 - v_3$

$C \leftarrow v_1 - v_4$

$v_6 \leftarrow a_0 A + \alpha a_2 B + \alpha a_1 C$

$F \leftarrow 1/v_6$

$c_0 \leftarrow AF$

$c_1 \leftarrow BF$

$c_2 \leftarrow CF$

return $c = (c_0 + c_1 w + c_2 w^2)$

---

2 Elliptic Curve Algorithms.

In what follows we assume that our curve $E(\mathbb{F}_q)$ is given in short Weierstrass form: $Y^2 = X^3 + aX + b$. All input and output points provided by the interface are represented in affine coordinates. However, internally reference implementation operate on points given in Jacobian coordinates. For point of infinity in affine coordinates refer to section 5. Point of infinity in Jacobian coordinates has $z = 0$.

This specification only supports curves with $b \neq 0$ to avoid point $(0, 0)$ being on a curve.

Algorithm 8

Point addition in Jacobian coordinates

Input: two points $F = (X_1, Y_1, Z_1)$ and $G = (X_2, Y_2, Z_2) \in E(\mathbb{F}_q)$ represented in Jacobian coordinates.

Output: $H = F + G \in E(\mathbb{F}_q)$ in Jacobian coordinates.

$U_1 \leftarrow X_1 \cdot Z_2^2$
$U_2 \leftarrow X_2 \cdot Z_1^2$
$S_1 \leftarrow Y_1 \cdot Z_2^3$
$S_2 \leftarrow Y_2 \cdot Z_1^3$
if $U_1 == U_2$ **then**
  if $S_1 \neq S_2$ **then**
    return POINT_AT_INFINITY
  else
    return POINT_DOUBLE(X_1, Y_1, Z_1)
  end if
end if
$H \leftarrow U_2 - U_1$
$R \leftarrow S_2 - S_1$
$X_3 \leftarrow R^2 - H^3 - 2 \cdot U_1 \cdot H^2$
$Y_3 \leftarrow R * (U_1 * H^2 - X_3) - S_1 * H^3$
$Z_3 \leftarrow H * Z_1 * Z_2$
return (X_3, Y_3, Z_3)

Algorithm 9
Point doubling in Jacobian coordinates

Input: point $P = (X, Y, Z) \in E(\mathbb{F}_q)$ given in Jacobian coordinates.
Output: $Q = 2P \in E(\mathbb{F}_q)$ represented in Jacobian coordinates.
  if $Y == 0$ **then**
    return POINT_AT_INFINITY
  end if
  $S \leftarrow 4 \cdot X \cdot Y^2$
  $M \leftarrow 3 * \cdot X^2 + a \cdot Z^4$
  $X' \leftarrow M^2 - 2 \cdot S$
  $Y' \leftarrow M \cdot (S - X') - 8 \cdot Y^4$
  $Z' \leftarrow 2 \cdot Y \cdot Z$
  return (X', Y', Z')

Algorithm 10
Conversion to Jacobain coordinates

Input: point $P = (X, Y)$ given in affine coordinates.
Output: representation (X', Y', Z') of P in Jacobian coordinates.
  $X' \leftarrow X$
  $Y' \leftarrow Y$
  $Z' \leftarrow 1$
  return (X', Y', Z')

Algorithm 11
Conversion from Jacobian coordinates

Input: point $P = (X, Y, Z)$ given in Jacobian coordinates.
Output: representation (X', Y') of P in affine coordinates.
  $X' \leftarrow X/Z$
  $Y' \leftarrow Y/Z^2$
  return (X', Y')

Remark. In the last algorithm we need to invert element Z in field $\mathbb{F}_q$. As parameters of this field are supplied by user it is possible for Z to have no inverse. We treat this case separately and simply return a point at infinity (see section 5).

3 Point Multiexponentiation

Given a vector of points $(X_1, X_2, \ldots, X_n) \in E(\mathbb{F}_q)$ and a vector of corresponding degrees $(p_1, p_2, \ldots, p_n)$ the multiexponentiation task is to compute the product $X_1^{p_1} X_2^{p_2} \dots X_n^{p_n}$. Reference implementation accomplish this task with the divide-and-conquer style Pippenger algorithm.

Algorithm 12
Pippenger algorithm

Input: vector of Points $\mathbb{X} = (X_1, X_2, \dots, X_n) \in E(\mathbb{F}_q)$ and vector of powers $\mathbb{P} = (p_1, p_2, \dots, p_n) \in \mathbb{Z}$, n - the upper bound of all elements of $\mathbb{P}$, c - bit-length of one chunk. For simplicity we assume that n is divisible by c (if this is not the case, we may slightly enlarge n to satisfy this condition).
Output: multiexponentiation: $X_1^{p_1} \dots X_n^{p_n}$

```
sum ← POINT_AT_INFINITY
for i n/c - 1 to 0 do
  make 2^c − 1 buckets and initialize them with POINT_AT_INFINITY
  (that's equivalent of zero). We call the constructed set of buckets 𝔹
  and index them from 1 to 2^c − 1 - there is no bucket for "zero".
  for X, p in ZIP(𝕏, ℙ) do
    idx ← (p >> 2^i c)% 2^c
    if idx ≠ 0 then
      𝔹[idx]+ = X
    end if
  end for
  temp ← POINT_AT_INFINITY
  for j in 2^c − 1 to 1 do
    temp ← temp + 𝔹[j]
  end for
  sum = 2^c ∗ sum + temp
end for
return sum
```

---

4 Pairing of elliptic curves

4.1 overview

We describe Ate's pairing for a generic curve in short Weierstrass form.

Remark In preceding algorithm we assumed that embedding degree k is even. This is done for simplification of algorithm (odd k will require additional computations in Miller loop.) The reference implementation deals only with even embedding degree curves.

Algorithm 13
Tate's pairing

Input: $r \in \mathbb{N}$ - odd prime, $\mathbb{G}_1$ - subgroup of order r in $E(\mathbb{F}_q)$. $k > 1$ - is even embedding degree of $E(\mathbb{F}_q)$ corresponding to r, $\mathbb{G}_2$ - subgroup of order r in $E[r] \cap Ker(\pi_q - qId)(\mathbb{G}_2 \subset E(\mathbb{F}_{q^k}$, n - number of points in $E(\mathbb{F}_q)$, $T = q - n$, Point $P \in \mathbb{G}_1$ and point $Q \in \mathbb{G}_2$.
Output: reduced Ate pairing of Q and $P \in \mathbb{F}_r^*$

Miller loop
compute binary decomposition of $T : T = \sum_{i=0}^{I} b_i 2^i$
$T \leftarrow Q$
$f \leftarrow 1$
for i in I-1 to 0 **do**
  $\alpha \leftarrow (3x_T^2 + a)/(2yT)$
  $x_{2T} \leftarrow \alpha^2 - 2X_T$
  $y_{2T} \leftarrow -y_T - \alpha(x_{2T} - x_T)$ **compute line function**
  $f \leftarrow f^2(y_P - y_T - \alpha(X_P - X_T))$
  $T \leftarrow 2T$
  if $b_i = 1$ **then**
    $\alpha \leftarrow \frac{y_T - y_Q}{x_T - x_Q}$
    $x_{T+Q} \leftarrow \alpha^2 - X_T - X_Q$
    $y_{T+Q} = -y_T - \alpha(x_{T+Q} - x_T)$ **compute line function**
    $f \leftarrow f(y_P - y_T - \alpha(x_P - x_T))$
    $T \leftarrow T + Q$
  end if
end for
$f = f(x_P - x_T)$
Final exponentiation
return $f^{\frac{q^k-1}{r}}$

4.2 Final exponentiation

. Usually we have: $log_2(r) \approx log_2(p)$, so that power of final exponentiation is $(k-1)log_2(p)$-bits long. This means that final exponentiation is time exhaustive operation, so different methods are used to reduce its' computational cost. The main observation is that computing Frobenius morphism (p-th power in $\mathbb{F}_{p^k}$) is a rather cheap operation, so the trick is to find a suitable p-adic representation of exponent. Another useful fact is that $p^k - 1$ is divisible by $\Phi_k(p)$ and $\Phi_k(p)$ is divisible by r where $\Phi_k(x)$ - is k-th cyclotomic polynomial. Hence:

$$\frac{p^k-1}{r} = \frac{p^k-1}{\Phi_k(p)} \cdot \frac{\Phi_k(p)}{r}.$$

The first multiplier is called the easy part and the second multiplier is called the hard part of the final exponentiation.

MNT-4 curves Easy part: $\frac{p^4-1}{\Phi_k(p)} = \frac{p^4-1}{p^2+1} = p^2-1$ Hard part: $\frac{p^2+1}{r} = w_1p + w_0$, where w_0 is chosen so that $|w_o| < p/2$.

MNT-6 curves Easy part: $\frac{p^6-1}{p^2-p+1} = (p^3-1)(p+1)$ Hard part: $\frac{p^2-p+1}{r} = w_1p + w_0$, where w_0 is chosen so that $w_o < p/2$.

BN-curves Easy part: $\frac{p^12-1}{p^4-p^2+1} = (p^6-1)(p^2+1)$

Algorithm 14
Final exponentiation for BN-curves

Input: $f \in \mathbb{F}_{p^12}, x \in \mathbb{Z}$ - parameter used to generate BN-curve
Output: $z = f^{\frac{\Phi_12(p)}{r}}$

$a = f^x$
$b = a^2$
$a = b \cdot f^2$
$a = a^2$
$a = a \cdot b$
$a = a \cdot f$
$a = a^{-1}$
$b = a^p$
$b = a \cdot b$
$a = a \cdot b$
$t_0 = f^p$
$t_1 = t_0 \cdot f$
$t_1 = t_1^9$
$a = t_1 \cdot a$
$t_1 = f^4$
$a = a \cdot t_1$
$t_0 = t_0^2$
$b = b \cdot t_0$
$t_0 = b^x$
$t_1 = t_0^2$
$t_0 = t_1^2$
$t_0 = t_0 \cdot t_1$
$t_0 = t_0^x$
$t_0 = t_0 \cdot b$
$a = t_0 \cdot a$
$t_0 = f^{p^3}$
$z = t_0 \cdot a$
return z

BLS curves Easy part: $\frac{p^{1}2-1}{p^4-p^2+1} = (p^6-1)(p^2+1)$

Algorithm 15

Final exponentiation for BLS-curves

Input: $f \in \mathbb{F}_{p^{1}2}, x \in \mathbb{Z}$ - parameter used to generate BLS-curve

Output: $z = f^{\frac{\Phi_{1}2(p)}{r}}$

$t_0 = f^2$
$t_0 = t_0^{-1}$
$a = f^x$
$b = a^2$
$c = a \cdot t_0$
$d = c^{-1}$
$d = d \cdot f$
$t_0 = c^x$
$e = t_0 \cdot f$
$e = e^{p^3}$
$d = d \cdot e$
$t_0 = t_0^x$
$e = t_0 \cdot a$
$e = e^{p^2}$
$d = d \cdot e$
$t_0 = t_0^x$
$t_0 = t_0 \cdot b$
$e = f^{-1}$
$e = e \cdot t_0$
$e = e^p$
$d = d \cdot e$
$e = t_0^x$
$z = d \cdot e$
return z

Remark: The output of the easy part of final exponentiation will always be *unitary*, i.e. of the form $u^n + a_{n-1}u^{n-1} + a_{n-2}u^{n-2} + \cdots + a_0$. This also provides additional speedup for finite field arithmetic.

4.3 Twists

Another sort of optimizations comes from the use of twisted curves. $\mathbb{G}_2$ group is isomorphic to particular subgroup of points on the curve $E'(\mathbb{F}_{p^{\frac{k}{d}}})$, where d is the degree of twist. Let $\phi : E'(\mathbb{F}_{p^{\frac{k}{d}}}) \mapsto E(\mathbb{F}_{p^k})$ be this isomorphism. The addition law on E' remains the same, but the number of $\mathbb{F}_p$ elements required to represent x and y coordinates is smaller. In "twisted" version of Ate pairing all double-and-add steps of Miller loop are carried over the point $\phi^{-1}(T)$ on the twisted curve and we return back to E (via ϕ) only for line function computations.

Here are the formulas for twisted curves and morphisms ϕ for all families of curves used in reference implementation.

MNT-4 and MNT-6 curves: Here we have quadratic twist (d = 2): If $E : y^2 = x^3 + ax + b$ then $E' : y^2 = x^3 + a\nu^2 x + b\nu^3$, where $\nu \in \mathbb{F}_{p^{\frac{k}{d}}}$ such that $\sqrt{\nu} \notin \mathbb{F}_{p^{\frac{k}{d}}}$ and $\sqrt{\nu} \in \mathbb{F}_{p^k}$

$\phi : E' \mapsto E \quad (x, y) \to (\frac{x}{\nu}, \frac{y}{\nu\sqrt{\nu}})$

BN-curves and BLS-curves: Here we have setic twist (d = 6). Possible twists are classified in two types, which are called D-type and M-type respectively. Let $E : y^2 = x^3 + b$ (a is always zero for BN and BLS curves), and let $s \in \mathbb{F}_{p^2}$ be such that $X^6 - s$ is irreducible over $\mathbb{F}_{p^2}[X]$ and let $z \in \mathbb{F}_{p^{12}}$ be its' root (so that $z^6 = s$)

D-type twist: $E' : y^2 = x^3 + b/s$ and $\phi : (x, y) \to (x \cdot z^2, y \cdot z^3)$

M-type twist: $E' : y^2 = x^3 + bs$ and $\phi : (x, y) \to (xs^{-1}z^4, ys^{-1}z^3)$

5 Conventions

The only convention applied in this specification is that "point at infinity" is encoded in affine coordinates as $x = 0$, $y = 0$. This is legitimate due to the fact that only elliptic curves in a Weierstrass form with $b \neq 0$ are supported. Such encoding is applied when both parsing the input and encoding the output of any operation.

6 Error propagation

This section describes edge cases. There are only two of them.

6.1 No inverse element in F_p or its' extensions everywhere but when performing Jacobian into affine coordinates conversion

This can happen in Miller loop evaluation in pairings. In this case absence of inversion is propagated to the level of API call and API call SHOULD return error.

6.2 No inverse element in F_p or extensions when performing Jacobian into affine coordinates conversion

In this case "point of infinity" is returned following section 5.