Review of algorithms for algebraic primitives (finite fields and elliptic curves)

Balázs Kőműves

Faulhorn Labs
July 26, 2023

Contents

1. Introduction

There are a lot of algorithms and trick used when implementing algebraic primitives (primarily: finitie field arithmetic, elliptic curves and polynomials) used in zero-knowledge proofs. In this document I try to collect together and describe some of these.

2. Core operations

Some of the most important operations we will need are listed below.

Finite fields:

- negation, addition, subtraction
- multiplication, squaring
- multiplicative inverse
- division
- general n-th power
- for some more complex algorithms: square root
- uniformly random field element

Elliptic curves:

- checking if a point is actually on the curve
- conversion between different representations (affine, projective, weighted projective, etc)
- point addition and doubling
- point negation and subtraction
- mixed addition (adding points in different repr.: an affine and a projective)
- scalar multiplication
- multi-scalar multiplication (MSM)
- for more complex algorithms: pairings
- uniformly random curve / subgroup element
- subgroup check
- hash-to-curve
- point compression / decompression

Univariate polynomials:

- evaluation at points
- evaluation on a multiplicative subgroup (NTT)
- negation, addition, subtraction
- multiplying by a scalar
- polynomial multiplication
- long division; division by special polynomials
- Lagrange interpolation
- interpolation on a subgroup (inverse NTT)

Multivariate polynomials: TODO

CHAPTER 1

Finite fields

In this context we mostly deal with large prime fields and their their low-degree (usually quadratic or cubic) extensions. Elements of prime fields has at least two common representations: the standard form, where (the unique) number $0 \leq x < p$ represents the element $[x] \in \mathbb{F}_p$; and the Montgomery form, where $0 \leq (Rx \mod p) \leq p$ represents $[x] \in \mathbb{F}_p$, for some fixed $R = 2^k > p \in \mathbb{N}$. Since multiplication is faster in the Montgomery form, it's almost universally used in ZK-oriented libraries. However the standard representation can be still useful in some contexts, or for testing purposes.

Since we are targeting 64-bit microprocessors, it seems natural to represent field elements by ℓ 64-bit words (called "limbs"), where $\ell = \lceil \log_2(p)/64 \rceil$.

1. Operations on finite fields

Some of the useful operations on finite field elements are:

- equality of elements, equality with zero or one
- addition, subtraction, negation (additive inverse)
- multiplication, squaring
- division, (multiplicative) inverse, batch inverse
- raising to the n-th power for arbitrary $n \in \mathbb{Z}$
- square root, checking if an element is a square (quadratic residue)
- conversion between different representation

For extension fields, we also have:

- embedding of the base field
- Frobenius automorphism

2. Prime fields in standard representation

Addition, subtraction and negation is simple: Since we assume canonical representation (though an implementation must take care to actually enforce this!), we can just add / subtract big integers and check for overflow / underflow. Note: in practice subtraction is a tiny bit faster, because checking for underflow does not need any comparision, it's enough to check the final carry flag. Negation of $[x]$ is $[p - x]$.

2.1. Multiplication and squaring. Multiplication is done by multiplying the representant numbers as non-negative integers, and then taking reduction modulo p. The latter is normally done using the Barret reduction algorithm.

Multiplication of integers can be done in a few ways too:

- "Comba multiplication" doesn't change the basic multiplication, but rearranges carry, so that is more efficient on actual computers.
  `https://everything2.com/title/Comba+multiplication`

- it's not clear if using something like Karatsuba multiplication can make the first step faster on modern hardware and small ℓ-s.

2.2. Power. This is normally done using the standard 'fast exponentiation algorithm': Write the exponent in binary: $n = \sum 2^k e_k$; then

$$a^n = \prod_{k\,:\,e_k=1} a^{2^k}$$

and one can compute the sequence $\{1, a, a^2, a^4, a^8, \dots\}$ using repeated squaring.

2.3. Inverse. This can be done (rather inefficiently) using Fermat's litte theorem: $a^{-1} = a^{p-2}$, or (more efficiently) using the extended binary Euclidean algorithm.

2.4. Division. This can be done either by multiplying with the inverse, or directly using the extended binary Euclidean algorithm.

2.5. Montgomery batch inversion. Computing the inverse is expensive; this trick replaces computing n inverses by computing a single inverse and $3(n-1)$ multiplications, which is much cheaper. The idea is:

(1) first compute the products

$$m_1 := a_1, \quad m_2 := a_1 a_2, \quad m_3 := a_1 a_2 a_3, \quad \dots \quad m_n := a_1 a_2 \cdots a_n$$

(2) calculate the inverse $d_n := m_n^{-1}$
(3) recursively compute the inverses

$$d_{n-1} := m_{n-1}^{-1} = a_n d_n, \quad d_{n-2} := m_{n-2}^{-1} = a_{n-1} d_{n-1}, \quad \dots \quad d_1 := m_1^{-1} = a_2 d_2$$

(4) set $m_0 = 1$, and finally compute $a_k^{-1} = m_{k-1}/m_k = m_{k-1} d_k$ for all $1 \le k \le n$.

2.6. Checking being a square (quadratic resiude). One can compute the Jacobi symbol using a standard algorithm based on quadratic reciprocity.

2.7. Computing square root. There are several algorithms for computing the square root modulo a prime p, for example:

- the Tonelli-Shanks algorithm (can be deterministic, relatively simple)
- Cipolla's algorithm (probabilistic; could be possibly faster for highly 2-adic fields?)
- the Berlekamp-Rabin algorithm (probabilistic, complicated)
- Pocklington's algorithm (1917, uses case separation and large powers, one case is probabilistic)
- Kunerth's algorithm (1878, works for non-prime modulus?)

3. Prime fields in Montgomery representation

The Montgomery representation allows for faster multiplication than the standard representation, and since multiplication is probably the operation where most time is spent, it makes a lot of sense to use this representation. For the Montgomery multiplier normally $R = 2^{64\ell}$ is chosen.

This is the representation used in most ZK libraries and file formats (for example the powers-of-tau files implicitly use this).

Addition, subtraction, negation and scaling by a constant (in standard representation) is the same as with the standard representation.

For multiplication, we can use the Montgomery reduction algorithm. Wikipedia has reasonable explanation of these.

For division and inversion you can reuse the standard representation division and inversion algorithm but need to add a Montgomery reduction step.

For conversion between standard form and Montgomery form, again you can use the Montgomery reduction algorithm.

3.1. Montgomery reduction. This algorithms takes a number T in the the range $T \in [0, Rp)$, and returns a number $S \in [0, p)$ such that $R^{-1}T \equiv S \mod p$. It assumes that: $\gcd(R, p) = 1$ (automatic here), $R = 2^{64k}$ (in the multi-precision version), and that we have access an inverse $q \in [0, 2^{64})$ such that $pq + 1 \equiv 0 \mod 2^{64}$ (this can be precalculated). In practice we also need $R > p$; so a very natural choice is $R = 2^{64\ell}$. More general versions of the algorithm exist, but this is the one which looks relevant here.

4. Field extensions

TODO

CHAPTER 2

Elliptic curves

Elliptic curves, defined over a field $\mathbb{F}$ (to avoid technical difficulties, assume that $\text{char}(\mathbb{F}) \neq 2, 3$) are most commonly presented in the so-called "short Weierstrass form", as the set of solutions $(x, y) \in \mathbb{F}^2$ of the equation

$$y^2 = x^3 + Ax + B$$

where $A, B \in \mathbb{F}$ are parameters, such that $4A^3 + 27B^2 \neq 0$. All elliptic curves can be written in this form.

There is one more point on the curve, the point at infinity $O = [0 : 1 : 0] \in \mathbb{P}^2 := \mathbb{PF}^3$, which can be recovered using projective coordinates:

$$y^2 z = x^3 + Axz^2 + Bz^3$$

1. The group law

The most important property of elliptic curves is that the points of a curve form a group. A possible intuition behind this is that if you intersect a line with a degree 3 curve, then you get 3 intersections P, Q, R, so you can define an operation $+$ for example such that $P + Q + R = 0$. The point at infinity O will play the role of 0. Of course you have to prove that this operation satisfy the required properties of a group.

Note that the case $U = P + P = 2P$ is special: The natural thing is to use the tangent line at that point. This is called "doubling". In affine Weierstrass coordinates, one can derive the following formulas for point addition and doubling:

addition			doubling		
$R = P + Q$			$U = 2P$		
s	$=$	$\frac{y_Q - y_P}{x_Q - x_P}$	t	$=$	$\frac{3x_P^2 + A}{2y_P}$
x_R	$=$	$s^2 - x_P - x_Q$	x_U	$=$	$t^2 - 2x_P$
y_R	$=$	$-y_P - s(x_R - x_P)$	y_U	$=$	$-y_P - s(x_U - x_P)$

Here s and t are the slopes of the secant resp. tangent lines through P, Q (resp. at P). Note that the special point O must be handled separately! Finally, negating a point is very simple: Just flip the y coordinate, so

$$-(x, y) = (x, -y)$$

2. Different representations

Elliptic curves can have many different representations. We've already seen affine and projective coordinates, but there are many more.

The web database `https://hyperelliptic.org/EFD/g1p/auto-shortw.html` contains many explicit addition and doubling formulas for different representations.

2.1. Mixed addition. Sometimes we have to add two points in (possibly weighted) projective coordinates, but one of them is normalized to have $z = 1$. This happens for example when doing MSM with group elements stored in affine form (for example when doing KZG commitments). In this case we can have a faster algorithm.

2.2. Projective coordinates. Projective coordinates have the big advantage that you can implement addition and doubling without division, which is good because division in finite fields is very slow to compute compared to multiplication.

For explicit formulas, see:
`https://hyperelliptic.org/EFD/g1p/auto-shortw-projective.html`
Summary: 12M for addition; 5M+6S=∼11M for doubling, 11M for mixed addition (M=multiplication, S=squaring).

2.3. Weighted projective coordinates. A next interesting one is weighted projective coordinates with weights $(2, 3, 1)$. That is, $[x : y : z] = [\lambda^2 x : \lambda^3 y : \lambda z]$ for $\lambda \neq 0$. This is often called "Jacobian coordinates" (not to be confused with Jacobian curves!). In this form doubling becomes cheaper but addition more expensive.

$A = 0$: `https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html`
$A \neq 0$: `https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian.html`

Summary: For $A = 0$, 11M+5S=∼16M for addition, 2M+5S=∼7M for doubling, 7M+4S=∼11M for mixed addition. For the case of general A it's the same but doubling is 9M instead.

2.4. Montgomery form. Montgomory curves have the most efficient scalar multiplication formula (?), but not all curve can be written in Montgomery form. Wikipedia claims that an elliptic curve in Weierstrass form can be written in Montgomery form if and only if

- the equation $z^3 + Az + B = 0$ has at least one root $\alpha \in F$;
- and for that α the field element $3\alpha^2 + A$ is a quadratic residue in $\mathbb{F}$.

3. Subgroup check

Normally we are interested in a prime-order subgroup $\mathbb{G}_1 \subset E(\mathbb{F}_p)$ of the curve, with $|\mathbb{G}_1| = r$. For some curves, like the BN128, we actually have $\mathbb{G}_1 = E(\mathbb{F}_p)$, but for other curves, like the BLS12-381, we actually have $|G_1| = r \ll |E(\mathbb{F}_p)$. When we got an elliptic curve point from an untrusted source (for example in the verifier), it's important to check both that it's a point on the curve and that it is a point on the required subgroup.

A simple algorithm for this is to compute $r * (x, y)$ and check whether it's the point at the infinity. However, this is relatively slow.

More efficient algorithms for the curve BLS12-381 are described in the paper "Faster Subgroup Checks for BLS12-381" by Sean Bowe.

4. Hash-to-curve and random curve points

We may want to convert a number to a curve point or subgroup point, for several possible reasons: Either to randomly generate a point, or to map some other data (typically a message hash) to a point.

The simplest way to take a number mod r, and multiply the generator by it. This will give an element of $\mathbb{G}_1$. However, this is relatively slow.

The following method is faster: Map the number to a coordinate $x \in \mathbb{F}_p$. Check whether $x^3 + Ax + B$ is a quadratic residue or not. If yes, we got (usually two) curve points, select one. If you want to end in the subgroup G_1, multiply by the cofactor h.

An even better approach is using the so-called SWU map.

5. Scalar multiplication

For an integer $k \in \mathbb{Z}$ and a group element $g \in \mathbb{G}$ we can define

$$k * g := \begin{cases} g + g + g + \cdots + g & (k \text{ times} \quad \text{if} \quad k > 0 \\ -g - g - g - \cdots - g & (|k| \text{ times} \quad \text{if} \quad k < 0 \\ O & \text{if} \quad k = 0 \end{cases}$$

In applications k is typically very big, say on the order of 2^{256}.

The basic algorithm is essentially the same as the "fast exponentiation" algorithm: Compute the sequence $g, 2g, 4g, 8g, \ldots$ using repeated doubling, then if the binary decomposition of k is $k = \sum b_i 2^i$, then compute

$$k * g = \sum_{i\,:\,b_i=1} (2^i g)$$

A review of some of the ideas can be found in Daniel J. Bernstein's paper:
`https://cr.yp.to/papers/pippenger-20020118-retypeset20220327.pdf`

5.1. Non-adjacent form (NAF). . In the simplest form, the idea is to write n in *signed binary* form instead, that, using digits $1, 0, -1$. With three digits, it's always possible to ensure that there are no consecutive ± 1 digits; this ensures that there will be at most $n/2$ terms in the (signed analogue of the) above sum, where $n = \log_2(k)$ is the number of digits.

5.2. Simple windowed form. The "windowed method", in the simplest form (apparently invented by Brauer around 1939) writes

$$k = \sum_{j=0}^{d-1} 2^{cj} a_j$$

where $0 \le a_j < 2^c$, then precalculates the table $h_i := i * g$ for $1 \le i < 2^c$, finally expresses

$$k * g = \sum 2^{cj} h_{a_j} = h_{a_0} + 2^c\Big(h_{a_1} + 2^c\Big(h_{a_2} + \cdots + 2^c\Big(h_{a_{d-2}} + 2^c\Big(h_{a_{d-1}} + 2^c h_{a_d}\Big)\Big)\cdots\Big)\Big)$$

One has to balance the size of the precalculation and the reduction in number of additions (we still have the same amount of doublings). For example for 256 bits one may choose $c = 4$. Note that the table can be calculated using special "addition chains", for example:

j	1	2	3	4	5	6	7
h_j	h_1	h_2	h_3	h_4	h_5	h_6	h_7
h_j	g	$2g$	$g + h_2$	$2h_2$	$g + h_4$	$2h_3$	$g + h_6$

Observation (Thurber): one can leave out the even entries of the table h_{2i}, by changing $\cdots + h_{2i} + 2^c(\cdots$ to $\cdots + 2(h_i + 2^{c-1}(\cdots$.

5.3. Sliding window. This is kind-of similar to, but an improvement compared to the previous. Precompute $d*g$ for $d = 2^{c-1} \dots 2^c - 1$, that is, where the most significant digit is 1. Start from the topmost digit of k, going down. If the given digit 0, just double the running sum; if it's 1, extract $(c-1)$ more digits (or less at the very end), multiply by 2^c by doubling, add the corresponding precomputed value, and continue.

`https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Sliding-window_method`

5.4. GLV optimization. This is a trick described by Gallant-Lambert-Vanstone. It used to be patented (which is seriously WTF), but the patent expired in 2020. The idea is that in some curves we can find parameters $\beta \in \mathbb{F}_p$ and $\lambda \in \mathbb{F}_r$ such that

- λ is a third root of unity: $\lambda^3 = 1 \in \mathbb{F}_r$
- β is a third root of unity: $\beta^3 = 1 \in \mathbb{F}_p$
- $\lambda * (x, y) = (\beta x, y)$
- λ has few bits, say about $\log_2(r)/2$

In this case we have $k \in \mathbb{F}_r$ (r being the size of the subgroup of the curve we are interested in), and we can write $k = k_0 + \lambda k_1$ with k_0, k_1 approximately half the size as k was. Then we have

$$k * (x, y) = (k_0 + \lambda k_1) * (x, y) = k_0 * (x, y) + (k_1 \lambda) * (x, y) = k_0 * (x, y) + k_1 * (\beta x, y)$$

The final expression on the right can be then computed using the "parallel windowed scalar exponentiation" below.

For example, the curve BLS12-381 curve's 256-bit *subgroup* allows to do this with $\lambda \sim 2^{128}$.

But maybe the size of λ is not that important, because we are in $\mathbb{F}_r$ and not in $\mathbb{Z}$? See the paper "Analysis of the Gallant-Lambert-Vanstone Method based on Efficient Endomorphisms" by Sica, Ciet and Quisquater.

5.5. Parallel windowed form. Apparently this is by Straus (sometimes called Shamir-Straus). The idea is to compute $k * P + l * Q$ by doing the same decomposition

$$k = \sum_{i=0}^{d} 2^{ci} k_i \qquad \text{and} \qquad l = \sum_{i=0}^{d} 2^{ci} l_i$$

precaculate 2 tables (one for P and one for Q), and do the add-then-double loop at the same time for the two (so all the doublings are shared).

Of course one can generalize for more points.

5.6. Multiple exponents at the same time. For the situations when we want to compute $g^{k_1}, \dots, g^{k_2}$ at the same time, we can do better. See Pippenger's algorithm in Bernstein's review article.

6. Multi-scalar multiplication (MSM)

Here the task is to compute the "linear combination"

$$\sum_{j=1}^{N} (k_j * g_j) := k_1 * g_1 + k_2 * g_2 + \dots + k_N * g_N$$

where $k_j \in \mathbb{Z}$ are numbers and $g_j \in \mathbb{G}$ are group elements. Often N is very big, say $10^5 - -10^6$ or even bigger. Somewhat surprisingly, this can be done faster than just naively doing N scalar multiplications and summing the results!

Optimizing this operation is very important because it's the bottleneck in many constructions of SNARKs (for example Groth16 or KZG-based ones).

Another review paper is "Algorithms for Multi-exponentiation" by Bodo Möller: `https://www.bmoeller.de/pdf/multiexp-sac2001.pdf`

6.1. The bucketing method. This is usually attributed to Pippenger. However Pippenger's full algorithm is much more complex. The idea is that as above, write coefficients k_j as

$$k_j = \sum 2^{ci} a_i^{(j)}$$

with $0 \leq a^{(j)} < 2^c$. Then, for a given window i, collect together all the g_j-s with the same $a_i^{(j)}$:

$$\sum_{j=1}^{N} a_i^{(j)} * g_j = \sum_{b=1}^{2^c-1} b * \underbrace{\left[\sum_{j\,:\,a_i^{(j)}=b} g_j \right]}_{S_b^{(i)}} = \sum_{b=1}^{2^c-1} b * S_b^{(i)}$$

Then the sum on the RHS can be rewritten as

$$\sum_{b=1}^{2^c-1} b * S_b = 1 * S_1 + 2 * S_2 + 3 * S_3 + \cdots + (2^c - 1) * S_{2^c-1} = T_1 + T_2 + \cdots + T_{2^c-1}$$

where

$$\begin{aligned}
T_{2^c-1} &= S_{2^c-1} \\
T_{2^c-2} &= S_{2^c-2} + T_{2^c-1} \\
T_{2^c-3} &= S_{2^c-3} + T_{2^c-2} \\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
T_2 &= S_2 + T_3 \\
T_1 &= S_1 + T_2
\end{aligned}$$

Finally we can handle the sum over the windows as usual.

Note that if g_j are given in affine coordinates (which is the case for trusted setups like KZG), then when computing $S_b^{(i)}$ we can always use mixed additions! Since for large N that dominates, this is important.

Let's count the costs. For a window size of 2^c, there are $m = \lceil \mathsf{nbits}/c \rceil$ windows.

- on average, $\frac{2^c-1}{2^c}N$, but at most N mixed additions per window (each g_j can belong to only 1 bucket)
- $2(2^c - 2)$ additions per window for computing first T_i-s and then $\sum T_i$
- nbits doublings and m additions for the final sum

In total, about

$$\mathsf{nbits} + \lceil \mathsf{nbits}/c \rceil \cdot \left\{ 1 + 2(2^c - 2) + \frac{2^c - 1}{2^c} N \right\}$$

operations (mostly additions).

For $\mathsf{nbits} = 256$, we can summarize choosing the best c as

N	10	100	1,000	10,000	100,000	1,000,000
c	1	4	7	10	13	16
ops	189	81	46	31	23	18

where ops means the number of operation divided by N. Note that we can do this whole thing in constant memory!

6.2. Precalculation + Straus. We can also do the "parallel windowed" thing for arbitrary N. For a window size of 2^c, we need about $(2^c - 2)N$ operations (and memory!) to precalculate the first $(2^c - 1)$ multiples of each g_j; then for each window we have at most N additions. In total, about

$$\mathsf{nbits} + (2^c - 2)N + \lceil \mathsf{nbits}/c \rceil \cdot \left\{ 1 + \frac{2^c - 1}{2^c} N \right\}$$

operations. This can be summarized as

N	10	100	1,000	10,000	100,000	1,000,000
c	4	4	4	4	4	4
ops	106	77	74	74	74	74

So this can be faster than the bucket method for $N < 100$, but for larger N-s it will be significantly slower.

CHAPTER 3

Appendix

1. BN128 curve

Also called: alt-bn128, BN256, BN254. See `https://hackmd.io/@jpw/bn254` for some more details.

Note: this has reduced security of around 100 bits because of the number field sieve algorithm. However Ethereum currently only supports this curve, via precompiles.

The curve is defined by the equation $y^2 = x^3 + 3$ on $\mathbb{F}_p$.

$A = 0$
$B = 3$
$p = 21888242871839275222246405745257275088696311157297823662689037894645226208583$
$r = 21888242871839275222246405745257275088548364400416034343698204186575808495617$
$h = 1$

The 2-adicity is 28, that is $2^{28}|(r-1)$. The canonical generator is

$$G_1 = (1, 2) \in \mathbb{G}_1$$

We can kind-of use the GLV trick $\lambda * (x, y) = (\beta x, y)$, but λ is too big (or maybe that's not such a big deal because the exponent is in $\mathbb{F}_r$???):

$\beta = 2203960485148121921418603742825762020974279258880205651966$
$\lambda = 4407920970296243842393367215006156084916469457145843978461$
$\log_2(\lambda) \approx 192$

2. BLS12-381 curve

See `https://hackmd.io/@benjaminion/bls12-381#About-curve-BLS12-381` for some more details.

The curve is defined by the equation $y^2 = x^3 + 4$ on $\mathbb{F}_p$.

$A = 0$

$B = 4$

$p = 4002409555221667393417789825735904156556882819939007885332 0\ldots$

$\ldots 5813612403165049083786444268762912901566403789427255978 7$

$r = 52435875175126190479447740508185965837690552500527637822603658699938581184513$

$h = 76329603384216526031706109802092473003$

The 2-adicity is 32, that is $2^{32}|(r-1)$.

The canonical generator in $\mathbb{G}_1$ (see the above link on how this was determined) is:

$G_1 = ($ $3685416753713387016781088315183077757961620795782546409894578378\ldots$

$\ldots 6886075923783763188360549476763458215481041854645 07$

$,$ $1339506544944476473020471379941921221584933875938349620426543736\ldots$

$\ldots 4165114239563335064727246553533665349923917564415 69$ $) \in \mathbb{G}_1$

We can use the GLV trick (in the r-sized subgroup, that is): $\lambda * (x, y) = (\beta x, y)$:

$\beta = 4002409555221667392624310435006688643935503118305586438271 1\ldots$

$\ldots 7139584297115748038137701540598005353935841713554093943 6$

$\lambda = 228988810152649578064853576960394133503$

$\log_2(\lambda) \approx 128$