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

Balázs Kőműves

Faulhorn Labs
June 8, 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
- 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 $0Rx \mod p \leq p$ 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
- 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 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. 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. Checking being a square (quadratic resiude). One can compute the Jacobi symbol using a standard algorithm based on quadratic reciprocity.

2.6. 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 needs 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.