This is a request for help in surveying algorithms for basic crypto operations which can be partitioned into off-chain and on-chain parts. Off-chain parts are precomputed and passed as calldata which is used to finish the on-chain part. We call such an algorithm “economical” with respect to a fully on-chain algorithm if it has lower gas cost.

In this post, we present some precomputation algorithms and show that they are economical. We also state open problems, including for pairings. Please contribute to this survey.

Model of computation

Our model of computation is EVM extended with EVM384 opcodes ADDMOD384

, SUBMOD384

, and MULMODMONT384

. ADDMOD384

and SUBMOD384

apply modular reduction with a conditional subtraction or addition, respectively, to bring the output back into the field. MULMODMONT384

is montgomery multiplication. These three opcodes seem to cover the bottlenecks for lots of cryptography. We are also interested in covering more cryptography with similar extensions EVM256 and EVM768. A relevant gas cost table follows.

	ADDMOD384	SUBMOD384	MULMODMONT384	calldata zero byte	calldata non-zero byte
gas	1-2	1-2	3-6	8	16

Table 1.

Some relevant gas costs. Ranges mean that costs are not yet finalized.

Edit: Each EVM384 opcode may require 0-3 gas to set up its parameters, depending on the final spec. We assume this setup costs 2 gas in estimates below.

A heuristic: until EVM opcodes are repriced, complicated algorithms in EVM are expensive, and on-chain parts should be kept to mostly EVM384 opcodes. An explanation of the situation is here.

General tricks

There may be several functionally equivalent algorithms, and the most economical one may depend on the input. We deploy EVM bytecode for each algorithm. Then for each input, we decide which on-chain algorithm is least expensive, and pass that algorithm’s identifier as calldata.

Non-primitive field operations (e.g. multiplicative inverse, square root)

Multiplicative inverse is used in some signature verification, coordinate transformation, and pairing algorithms. Square root appears in hash-to-curve algorithms.

[Z2019] (“weierstrudel”) propose precomputing inverse operations and passing the result as calldata. Verification of an inverse is checking whether the result times the input equals the multiplicative identity. Similarly for square root, verification is checking whether the square the result equals the input value (but prime fields are not closed under square root, so extra checks may be needed).

Using EVM384 opcodes, the precomputation method to compute inverse modulo a 256-bit prime costs ~550 gas

mostly from 32 bytes of extra calldata

. The author’s best non-precomputation inverse method costs ~2400 gas. So this precomputation method is economical.

Elliptic curve point addition (e.g. ECADD)

Elliptic curve point addition with this formula uses 13 ADDMOD384

or SUBMOD384

and 16 MULMODMONT384

. Point doubling with this formula uses 14 ADDMOD384

or SUBMOD384

and 7 MULMODMONT384

. Some optimizations may be possible with different coordinate systems or special cases with affine coordinates.

Under the conservative 2,2,6 gas costs for EVM384 opcodes, the costs are 180 gas

and 112 gas

for point addition and doubling, respectively. These gas costs are so low that precomputation would have little benefit.

These point addition formulas leave the output point in jacobian coordinates. Transforming back to affine coordinates (note: this transformation may be delayed) includes an expensive multiplicative inverse, but inverses can use the economical trick above.

Scalar multiplication of a group element (e.g. ECMUL, field element exponentiation)

We can precompute an addition chain for the scalar. The addition chain’s structure is passed as calldata and verified alongside the calculation to arrive at the scalar multiplication result (this algorithm sketched here is not yet implemented).

Now a rough estimate of gas costs for a 256-bit scalar multiplication of an elliptic curve point using addition chain calldata. Many optimizations are omitted.

- The addition chain length is ~306, of which ~244 are doublings. These estimates are based on [BC1989] and experimentally reproducible here with inputs num_scalars = 1; bitlength = 256

. The total cost of curve operations is 38,488 gas (at 180 and 112 gas per addition and doubling).

- By observing the experimental data, we notice that addition chain can be encoded concisely because the doublings come in consecutive chunks over the same value. Arbitrarily let the 61 additions be encoded with one byte each, and the doublings encoded as one byte for each of the <60 consecutive doubling chunks, for a total of 120 bytes of extra calldata

costing 2064 gas.

- Arbitrarily let the calldata processing and verification overhead be 30 gas per byte, so 3600 gas.
- Total is 38488+2064+3600=44152 gas

.

This precomputation is economical relative to the double-and-add algorithm which does 256 doubles and ~128 adds costing 51,712 gas. (We don’t compare to weierstrudel because our algorithm is more generic. More on weierstrudel later.)

Possible improvements include (i) using addition-subtraction chains, (ii) using better chain search algorithms than fast greedy heuristics in our experiment above, (iii) using hand-tuned algorithms for point tripling, quadrupling, etc. possibly with double base tricks, (iv) in special cases (e.g. BN128, BLS12-381, and secp256k1 curves), major optimizations like the GLV method, and (v) in special cases when the base group element is fixed we can precompute convenient scalar multiples of it and hard-code them at deploy-time.

Multi-scalar multiplication (“multi-exponentiation”)

Consider the precomputation method reported in [R1995] to precompute an (slightly generalized) addition chain which, when followed, starts with the input group elements and ends in the sum of the scalar multiplications. This addition chain’s structure can be passed as calldata and verified alongside the calculation. (This algorithm sketch is not yet implemented.)

Now a rough estimate of gas costs for elliptic curve points. First we give addition chain info, namely the number of additions and number of doublings.

	2 scalars	18 scalars
128-bit scalars	~110, ~87	~629, ~0
256-bit scalars	~219, ~180	~1252, ~0

Table 2.

Lengths of addition chains split into the number of additions and number of doublings, respectively. All values are experimental from a greedy algorithms and random inputs, reproducible here with appropriate bitlength

and num_scalars

.

We arbitrarily assume that calldata to encode the addition chain has bytelength equal to the length of the addition chain, and it costs 30 gas to process each byte of this calldata.

- For 2 128-bit scalars, we have 110 additions and 87 doublings for a cost of 29,544 gas. Plus 197 bytes of extra calldata

for 3,152 gas and processing for 5,910 gas. Total is 38,606 gas

.

- For 18 128-bit scalars, we have 629 additions for 113,220 gas. Plus 629 bytes of extra calldata

for 10,064 gas and processing for 18,870 gas. Total is 142,154 gas

.

- For 2 256-bit scalars we have 219 additions and 180 doublings for 59,580 gas. Plus 399 bytes of extra calldata

for 6,384 gas and processing for 11,970 gas. Total is 77,934 gas

.

- For 18 256-bit scalars we have 1,252 additions for 225,360 gas. Plus 1,252 bytes of extra calldata

for 20,032 gas and processing for 37,560 gas. Total is 282,952 gas

.

This precomputation of 2 128-bit scalars is economical with respect to weierstrudel benchmarks (which uses the GLV technique to split the 256-bit scalar to two 128-bit scalars). Actually, weierstrudel is an elaborate greedy addition chain algorithm which we can reproduce as an off-chain precomputation, and possibly improve upon. They use ~127 doublings and ~50 adds per point, which would save thousands of gas relative to our rough estimate above.

Possible improvements include (i) using addition-subtraction chains, (ii) using better chain search algorithms than fast greedy heuristics we use above (for example using tricks from weierstrudel like Strauss-Shamir and windowing but omitting unused table values), (iii) using hand-tuned algorithms for point tripling, quadrupling, etc. possibly with double base tricks, (iv) in special cases (e.g. BN128, BLS12-381, and secp256k1 curves), major optimizations like the GLV method, and (v) adding extra scalars which are convenient for producing a short addition chain, and subtracting them out later.

Pairings (e.g. ECPAIRING)

There are many well-known deploy-time hard-coded precomputation methods for pairings. For example, frobenius coefficients can be hard-coded. Another example for the special case of a fixed-input (e.g. BLS signature verification and PLONK), [BLKS2002] precompute all values dependent only on the fixed input.

An example of input-dependent precomputation is using the multiplicative inverse trick above. But inverse is only a small part of pairing algorithms.

Open problem: Find a pairing algorithm which allows input-specific off-chain precomputation that can be passed as calldata which is verifiable, economical, and gives significant (>10%) savings relative to doing it fully on-chain.

Request for help

Besides the open problem for pairings, there are similar open problems for other operations such as subgroup checks and hash-to-curve. The operations mentioned above can also be improved upon. And there is the more general open problem of designing whole cryptosystems to be more economical.

References

[BC1989] J. Bos and M. Coster, “Addition chain heuristics”, Advances in Cryptology - Proceedings of Crypto’89 (G. Brassard, ed.), Lecture Notes in Computer Science, vol. 435, Springer-Verlag, 1990, pp. 400-407.

[BKLS2002] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairingbased cryptosystems. In CRYPTO ’02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology, pages 354–368, London, UK, 2002. Springer-Verlag.

[R1995] Peter de Rooij, Efficient exponentiation using precomputation and vector addition chains, in Eurocrypt ’94 [28] (1995), 389–399.

[Z2019] Zachary Williamson et. al., Weierstrudel, (2019), Git repository, https://github.com/AztecProtocol/weierstrudel

.