Plonk unrolled for Ethereum

2020

Abstract

Short description of a proof system used in zkSync v1.1

1 Main argument unrolled

1.1 Extensions

Compared to the original PLONK paper we use two extensions:

1. State width is four instead of three in the original paper. This requires one extra permutation polynomial at the setup $\sigma_3(x)$ and does not change the copy-permutation argument that is generic over number of state (witness) polynomials.

2. Main gate equation is now $q_a(x)a(x) + q_b(x)b(x) + q_c(x)d(x) + q_d(x)d(x) + q_m(x)a(x)b(x) + q_{const}(x) + q_{d_{next}} d(x\omega) = 0$ where $a(x), .., d(x)$ are state polynomials (witnesses) and ω is a generator of the multiplicative domain D, such that for a circuit size n one has $|D| = n + 1$. Such gate equation allows one to work more efficiently with long linear combination by including up to three terms per gate compared to the original paper where it would be one term per gate ($q_{d_{next}} d(x\omega)$ term allows one to "peek" into the next row for efficient placement of terms).

3. For "outer" circuit (one that does aggregation of the "inner" proofs) there is a custom gate used that verifies that a differences $c(x) - 4d(x)$, $b(x) - 4c(x)$, $a(x) - 4b(x)$, $d(x\omega) - 4a(x)$ are indeed 2 bit wide. Thus if one starts with $d(x) = 0$ at the current trace step it allows an efficient range check of 8 bits per gate. It's done by addition of extra 4 terms of the form $\alpha(c(x) - 4d(x))(c(x) - 4d(x) - 1)(c(x) - 4d(x) - 2)(c(x) - 4d(x) - 3)$ to the main Plonk equation (we first place two highest bits, then two lower ones, etc).

1.2 Transcript implementation

To make verification non-interactive one has to use FS transformation. For this purpose we instantiate a transcript using keccak256 hash function that is a PRF hash function. We keep a two 32 byte variables $state_0$ and $state_1$, single $uint32$ variable $counter$, and create constants (domain separation tags) $DST_0 = uint32(0)$, $DST_1 = uint32(1)$, $DST_{chall} = uint32(2)$ and instantiate the following five functions ($uint32$ variables are encoded as BigEndian when used as byte strings):

1. $Commit(data)$: takes 32 bytes of $data$ and updates a state as
$$tmp := state_0$$
,
$$state_0 := keccak256(DTS_0||tmp||state_1||data)$$
,
$$state_1 := keccak256(DTS_1||tmp||state_1||data)$$
. Initially $state_0 = bytes32(0)$ and $state_1 = bytes32(0)$.

2. Commit $uint256$ scalar s: scalar is $uint256$ type that is in turn encoded as Big Endian 32 bytes. Thus commitment updates a state as
$$Commit(EncodeBE32(s))$$

3. Commit scalar field element Fr: On BN254 curve used in Ethereum scalar field element is encoded as $uint256$ type that is in turn encoded as Big Endian 32 bytes. Thus commitment updates a state as
$$Commit(EncodeBE32(Fr))$$

4. Commit curve point P: On BN254 curve used in Ethereum base field element is encoded as $uint256$ type that is in turn encoded as Big Endian 32 bytes. Commitment of the curve point (in affine form with coordinates x and y) is a sequence of steps
$$Commit(EncodeBE32(P.x))$$
,
$$Commit(EncodeBE32(P.y))$$

5. $GetChallenge$: draws a random challenge to perform aggregation over values of the scalar field Fr. To ensure uniform distribution of the challenge we do not use modular reduction. Bit length of the scalar field modulus is 254 bits on BN254 curve, so we take
$$challenge = uint256(keccak256(DST_{chall}||state_0||state_1||counter))$$
$$counter+ = 1$$
and take lowest 253 bits are of received $challenge$

1.3 Protocol

Here we present a protocol for the standard Plonk circuit without the custom gate, and later on extend it for our recursive case in the corresponding section.

Before the verification there is a setup phase when selector and permutations polynomials are defined and the corresponding commitments are output for the polynomials $q_a(x)$, $q_b(x)$, $q_c(x)$, $q_d(x)$, $q_m(x)$, $q_{const}(x)$, $q_{d_{next}}(x)$, $\sigma_0(x)$, $\sigma_1(x)$, $\sigma_2(x)$, $\sigma_3(x)$, as well as quadratic non-residues $k_{1,2,3}$ are agreed on.

One can refer to the separate document for expanded equations that make the main argument. Here we only unroll the sequence of steps to perform the verification. We use a bar symbol like $\bar{a}$ to define a commitment value to the polynomial $a(x)$ using Kate commitment.

Terms "transcript is updated" and "prover sends ... to verifier" are intrinsically synonims as this is unrolled non-interactive protocol. Explicit instantiation of the transcript described above is just a particular choice. In non-interactive version prover would send the same values to the verifier and verifier would send truly random challenges to the prover.

1. Prover outputs commitments to witness polynomials $a(x)$, $b(x)$, $c(x)$, $d(x)$ using Kate commitment
2. Transcript is updated by committing curve points $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{d}$
3. Challenges β, γ are drawn from the transcript using $GetChallenge$
4. Prover constructs permutation argument polynomial $z(x)$ using β and γ and commits to it
5. Transcript is updated by committing curve point $\bar{z}$
6. Challenge α is drawn from the transcript using $GetChallenge$ to construct $T(x)$ polynomial
7. Prover constructs $T(x)$ polynomial of degree $4n$ and splits it into four polynomials of degree n as $t_{0,1,2,3}(x)$ and commits to them
8. Transcript is updated by committing curve points $\bar{t_0}$, $\bar{t_1}$, $\bar{t_2}$, $\bar{t_3}$
9. Random point z is drawn from the transcript using $GetChallenge$ to check for relationships between polynomials
10. Prover constructs linearization polynomial $r(x)$ and claims values $a(z)$, $b(z)$, $c(z)$, $d(z)$, $d(z\omega)$, $\sigma_0(z)$, $\sigma_1(z)$, $\sigma_2(z)$, $r(z)$ and $T(z)$
11. Transcript is updated by committing claimed values (scalar field elements) in some fixed agreed order
12. Challenge v is drawn from the transcript using $GetChallenge$

13. Prover performs aggregated multiopening of the polynomial by constructing

$$\begin{aligned} W(x) = \frac{1}{x - z}[& \\ & t_3(x)z^3n + t_2(x)z^2n + t_1(x)z^n + t_0(x) - t(z) \\ & v(r(x) - r(z)) \\ & v^2(a(x) - a(z)) \\ & v^3(b(x) - b(z)) \\ & v^4(c(x) - c(z)) \\ & v^5(d(x) - d(z)) \\ & v^6(\sigma_0(x) - \sigma_0(z)) \\ & v^7(\sigma_1(x) - \sigma_1(z)) \\ & v^8(\sigma_2(x) - \sigma_2(z)) \\ &] \end{aligned} \tag{1}$$

and

$$\begin{aligned} W'(x) = \frac{1}{x - z\omega}[& \\ & v^9(z(x) - z(z\omega)) \\ & v^{10}(d(x) - d(z\omega)) \\ &] \end{aligned}$$

that are quotient polynomials for opening of Kate commitment (effectively proofs of opening) and output commitments to them $\bar{W}$ and $\bar{W}'$.

14. Transcript is updated by committing curve points $\bar{W}$ and $\bar{W}'$.

15. Challenge u is drawn from the transcript using $GetChallenge$ (only verifier actually performs this step as described below).

16. All the values that were committed to the transcript are part of the proof, so verifier trivially reconstructs the transcript and pulls all the values from there (e.g. α, β, ...) to perform the checks below.

17. Verifier checks an equation

$$\begin{aligned} T(z) = \frac{1}{Z_H(z)}[& r(z) + PI(z) \\ & - \alpha z(z\omega)(a(z) + \beta\sigma_0(z) + \gamma)(b(z) + \beta\sigma_1(z) + \gamma) \\ & (c(z) + \beta\sigma_2(z) + \gamma)(d(z) + \gamma) \\ & - \alpha^2 L_0(z)] \end{aligned} \tag{2}$$

where $PI(z)$ is a value of the public inputs polynomial as described in the original paper. If this check fails verifier returns "false".

18. Verifier keeps a track of claimed values and reconstructs $\bar{r}$ himself

$$\begin{gathered}
\bar{r} = \bar{q_a}a(z) + \bar{q_b}b(z) + \bar{q_c}c(z) + \bar{q_d}d(z) \\
+ \bar{q_m}a(z)b(z) + \bar{q_{const}} + \bar{q_{d_{next}}}d(z\omega) \\
+ \alpha[\bar{z}(a(z) + \beta z + \gamma)(b(z) + \beta k_1 z + \gamma)(c(z) + \beta k_2 z + \gamma)(d(z) + \beta k_3 z + \gamma) \\
- z(z\omega)(a(z) + \beta\sigma_0(z) + \gamma)(b(z) + \beta\sigma_1(z) + \gamma)(c(z) + \beta\sigma_2(z) + \gamma)\beta\bar{\sigma_3}] \\
+ \alpha^2[\bar{z}L_0(z)]
\end{gathered} \tag{3}$$

19. Verifier aggregates pairing checks for opening equations into one using u as a random challenge by reconstructing a right hand side of equations for $W(x)$ and $W'(x)$. Reconstruction happens "in the exponent", so we use a symbol [1] to define multiplication by the generator of G1 group of BN254

$$\begin{gathered}
\bar{E} = [\\
\bar{t_3}z^3n + \bar{t_2}z^2n + \bar{t_1}z^n + \bar{t_0} - t(z)[1] \\
v(\bar{r} - r(z)[1]) \\
v^2(\bar{a} - a(z)[1]) \\
v^3(\bar{b} - b(z)[1]) \\
v^4(\bar{c} - c(z)[1]) \\
v^5(\bar{d} - d(z)[1]) \\
v^6(\bar{\sigma_0} - \sigma_0(z)[1]) \\
v^7(\bar{\sigma_1} - \sigma_1(z)[1]) \\
v^8(\bar{\sigma_2} - \sigma_2(z)[1]) \\
]
\end{gathered} \tag{4}$$

$$\begin{gathered}
\bar{F} = [\\
v^9(\bar{z} - z(z\omega)[1]) \\
v^{10}(\bar{d} - d(z\omega)[1]) \\
]
\end{gathered}$$

20. Verifier performs actual aggregated pairing check. We use [2] and $[s]$ to define generator of G2 and corresponding value $s[2]$ from the trusted setup.

$$e(E + uF + z\bar{W} + uz\omega\bar{W}', [2])e(-\bar{W} - u\bar{W}', [s]) == 1 \quad (5)$$

If this check succeeds verifier returns "true" and "false" otherwise.

We should note that aggregation of all the parts in a form $y[1]$ can be done by first aggregating scalars and then performing a single multiplication by the generator of G1.

2 Recursion

2.1 Aggregation

To perform a recursive aggregation of the proofs we assume the following:

1. Canonical encoding of F_q element (where BN254 curve point coordinates reside) as a sequence of F_r elements (scalar field of the circuit)
2. We use such canonical encoding to encode verification keys vk_i and proofs π_i of Plonk circuits (in the example below we talk about a single proof aggregation first, and then about extension to more than one proof being aggregated per circuit)
3. We create a circuit that implements a Plonk verification procedure described in the previous section all the way up to the check 5. Instead we accumulate terms $A = E + uF + z\bar{W} + uz\omega\bar{W}'$ and $B = -\bar{W} - u\bar{W}'$ (respectively the terms that have to be paired with [2] and [s]) and propagate A and B through the inputs to the cirucit itself. We do not return whether verification has returned "true" or "false", but instead we make a circuit unsatisfiable if verification fails

2.2 Verification equaltion for aggregation circuit

To implement a mentioned circuit we use the same Plonk proof system with an extension of the custom range-check gate described in the introduction. This changes an equaltion 2 to the following form

$$\begin{aligned}T(z) = \frac{1}{Z_H(z)}[r(z) + PI(z)\\ - \alpha^5 z(z\omega)(a(z) + \beta\sigma_0(z) + \gamma)(b(z) + \beta\sigma_1(z) + \gamma)\\ (c(z) + \beta\sigma_2(z) + \gamma)(d(z) + \gamma)\\ - \alpha^6 L_0(z)] \quad (6)\end{aligned}$$

(only accounts for changed powers of α) and the correpoding linearization equation 3 to the form

$$\begin{aligned}\bar{r} = sel_{main}(z)[\bar{q_a}a(z) + \bar{q_b}b(z) + \bar{q_c}c(z) + \bar{q_d}d(z) \\ + \bar{q_m}a(z)b(z) + \bar{q_{const}} + \bar{q_{d_{next}}}d(z\omega)] \\ + \bar{sel_{range}}[\\ \alpha(c(z) - 4d(z))(c(z) - 4d(z) - 1)(c(z) - 4d(z) - 2)(c(z) - 4d(z) - 3) \\ + \alpha^2(b(z) - 4c(z))(b(z) - 4c(z) - 1)(b(z) - 4c(z) - 2)(b(z) - 4c(z) - 3) \\ + \alpha^3(a(z) - 4b(z))(a(z) - 4b(z) - 1)(a(z) - 4b(z) - 2)(a(z) - 4b(z) - 3) \\ + \alpha^4(d(z\omega) - 4a(z))(d(z\omega) - 4a(z) - 1)(d(z\omega) - 4a(z) - 2)(d(z\omega) - 4a(z) - 3) \\ ] \\ + \alpha^5[\bar{z}(a(z) + \beta z + \gamma)(b(z) + \beta k_1 z + \gamma)(c(z) + \beta k_2 z + \gamma)(d(z) + \beta k_3 z + \gamma) \\ - z(z\omega)(a(z) + \beta\sigma_0(z) + \gamma)(b(z) + \beta\sigma_1(z) + \gamma)(c(z) + \beta\sigma_2(z) + \gamma)\beta\bar{\sigma_3}] \\ + \alpha^6[\bar{z}L_0(z)] \quad (7)\end{aligned}$$

that indicates that we have added custom gate and created additional selectors $sel_{gate}(x)$ to indicate what type of the gate is applied on a particular trace step. Commitments to such selectors become part of the setup for the circuit. We use a value $sel_{main}(z)$, so we extend terms 1 and 4 to include the opening of the commitment to the polynomial $sel_{main}(x)$ at the point z to the value $sel_{main}(z)$.

2.3 Final check

Assume that we've obtained values A and B from the circuit that does an aggregation. In a similar way we call C and D values obtained by out-of-circuit verifier at the step 5 in a same approach $C = E + uF + z\bar{W} + uz\omega\bar{W}'$ and $D = -\bar{W} - u\bar{W}'$. More precisely, our verifications are always performed all the way up to the moment when we have a pair of points in G1 group on BN254 curve that are later used to perform a pairing check. Now we use the same transcript primitive as described above to accumulate A, B, C, D into the fresh transcript, draw a challenge β and perform a batched pairing check as

$$e(A + \beta C, [2])e(B + \beta D, [s]) == 1$$

Validity of such check concludes that our "inner" proof was valid (one that we have aggregated using the described circuit), and simultaneously "outer" proof of validity of such aggregation is valid.

3 Extension to many proofs per aggregate and concrete implementation

To increase gas efficiency of aggregation we further extend the protocol:

1. We allow more than one proof π_i to be aggregated in a circuit. For this purpose we first construct set of pairs (A_i, B_i) as described above, then derive a challenge γ using Fiat-Shamir heuristics over canonical encodings of π_i in the order of aggregation. Namely now $A = \sum_{i=1}^{N} \gamma^i A_i$ and same for B. Alternatively we could also derive γ based on encodings of (A_i, B_i) itself as those are deterministic from the π_i

2. We allow to setup a verification key for each proof at "runtime". For this purpose we allow a root hash of the tree of canonical encodings of vk_i, as well as indexes in this tree to be included as public inputs to the circuit (technically we use only a single input to the circuit that is itself a SHA256 commitment to some encoding of all the the information required)

3. In a similar way we include all inputs to individual proofs that are aggregated to be propagated through the public input of the aggregation circuit

We do not discuss particular implementation details here, e.g. an encoding of indexes of vk_i as bytes to serve as the input for SHA256 to derive a single circuit input.