In a previous post, I introduced [libkzg

](https://github.com/weijiekoh/libkzg), an implementation of the KZG polynomial commitment scheme. This library now supports multi-point proof generation, as well as on-chain verification of up to 128 points of evaluation. In this post, I will describe the new functions pertaining to multi-point proof generation and verification, discuss its implementation, and describe tradeoffs made.

Implementation details

Typescript functions

genMultiProof(coefficients: Coefficient[], indices: number[] | bigint[]): MultiProof

Generates a KZG proof to the evaluations at indices

for the polynomial with the given coefficients.

A MultiProof

is a single G2 point.

verifyMulti(commitment: Commitment, proof: MultiProof, indices: number[] | bigint[], values: bigint[])

Verifies a MultiProof

given the polynomial commitment, as well as the associated indices and evaluations.

Solidity functions

verifyMulti(Pairing.G1Point memory _commitment, Pairing.G2Point memory _proof, uint256[] memory _indices, uint256[] memory _values, uint256[] memory _iCoeffs, uint256[] memory _zCoeffs)) public view returns (bool)

The Solidity verifyMulti

function does the same as the Typescript verifyMulti

function, but requires two additional parameters: _iCoeffs

and _zCoeffs

.

_iCoeffs

should be the coefficients to a polynomial which interpolates the points denoted by _indices

and _values

.

_zCoeffs

is a polynomial which intersects y=0

for each x-value in _indices

.

These polynomials are required to verify the proof. See the appendix for a more detailed explanation.

The contract will ensure that _iCoeffs

and _zCoeffs

are valid by evaluating them at each point in _index

and checking whether the result is a corresponding value in _values

or 0 respectively. An alternative method is to perform polynomial interpolation in Solidity, but I have not explored that yet.

Next, it will compute the KZG commitments for _iCoeffs

and _zCoeffs

respectively. It is able to do this as the verifier contract has 129 G1 values from the trusted setup, stored during deployment.

Finally, it performs a single pairing check described in the appendix.

Limitations

The contract supports verification of up to 128 points per verifyMulti

function call.

Gas costs

Deployment

The contract costs about 7.3 million gas to deploy. The bulk of this comes from storing 129 G1 points (a total of 258 uint256

values) from the trusted setup on-chain.

Proof verification

Multi-point proof verification is significantly cheaper per point than single-point proof verification as long as the number of points is greater than 1.

Points

Cost

% Savings

1

193084

-8

2

221505

38

3

250283

53

4

280075

61

5

310152

65

6

341279

68

7

372763

70

8

405237

72

9

438008

73

50

2368081

73

75

4098829

69

100

6245670

65

128

9145327

60

Open questions

More efficient polynomial identity verification

The Solidity function verifies two polynomials by evaluating them at each index and checking the result. Is there a way to speed this up?

Practical use cases

What are some use cases that would immediately benefit from libkzg

? One downside of KZG polynomial commitments is that updates are less efficient than those for Merkle trees, which prevents them from substituting Merkle trees in applications which require them.

Trusted setup

What are some ways to more efficiently establish social trust in the structured reference string (SRS) values which this library uses? These values come from the Perpetual Powers of Tau ceremony. Teams which use it should independently verify the ceremony transcript and check whether the SRS values stored in the verifier contract match those from the chosen challenge file. Yet, the transcript is extremely bulky and takes a long time to verify.

Verkle trees

Some authors have suggested using KZG commitments as the accumulator for Merkle trees. This deserves a longer post, but for this construction to be gas-competitive, it must beat the efficiency of the Keccack and SHA256 hash functions, which consume 349 and 1586 gas respectively to hash two 32-byte values each. In contract, the KZG commitment function is much more expensive on-chain: it consumes 47496 gas to commit to two coefficients.

One may wonder if KZG commitments can be competitive accumulators for zero-knowledge use cases. This is an open question, as it is difficult to generate and verify them in ZK. Nonetheless, it is useful to compare them with the zk-SNARK friendly Poseidon function, which consumes 49858 gas to hash 2 values, and 112055 gas to hash 5 values. In contrast, the KZG commitment function consumes 85177 gas to hash 5 values.

Appendix I: the math behind multi-point proof verification

The KZG10 paper describes how to evaluate a multi-point proof as such:

[

1017×513 138 KB

](https://ethresear.ch/uploads/default/original/2X/b/bbeddaf21bdf67a30e39bb4122ad7c6d1c4650b7.png)

In summary:

1. Given a polynomial \phi(x)

whose KZG commitment is C

, and a set of indices B

to prove exist on \phi(x)

, i.e. (i_0, y_0) ... (i_{k-1}, y_{k-1})

, compute an interpolation polynomial r(x)

such that it goes through each point.

1. Compute a zero polynomial \prod_{i \in B}(x - i)

.

1. Compute a quotient polynomial \psi_B(x) = \frac{\phi(x) - r(x)}{ \prod_{i \in B}(x - i)}

.

1. The multi-point proof is the KZG commitment to the quotient polynomial: g^{\psi_B(\alpha)}

.

1. To verify the proof, perform this pairing check:

e(g^{\prod_{i \in B}(\alpha - i)}, g^{\psi_B(\alpha)}) \cdot e(g, g^{r(\alpha}})) = e(C, g)

Multiply each side by e(g, g^{r(\alpha}}))^{-1}

and apply [a]:

e(g^{\prod_{i \in B}(\alpha - i)}, g^{\psi_B(\alpha)}) = e(C, g) \cdot e(g, -1 \cdot g^{r(\alpha}}))

Recall some fundamental properties of elliptic curve pairings:

[a]

: e(P, Q)^{x} = e(x \cdot P, Q) = e(P, x\cdot Q)

[b]

: e(x \cdot P, y \cdot Q) = e(y \cdot P, x \cdot Q)

[c]

: e(P + S, Q) = e(P, Q) \cdot e(S, Q)

[d]

: e(P, Q + R) = e(P, Q) \cdot e(P, R)

Apply [a] and [c]:

e(g^{\prod_{i \in B}(\alpha - i)}, g^{\psi_B(\alpha)}) = e(C - g^{r(\alpha)}, g)

Make it EIP-197 friendly:

e(g^{\prod_{i \in B}(\alpha - i)}, g^{\psi_B(\alpha)}) \cdot e(-C - g^{r(\alpha)}, g) = 1

As such, the verifier contract function requires the following to perform a pairing check. It needs the following values:

- g^{\prod_{i \in B}(\alpha - i)}

as a G1 point.

- g^{\psi_B(\alpha)}

as a G2 point.

- C

as a G1 point.

- g^{r(x)}

as a G1 point.

- The G2 generator g

.

To compute g^{\prod_{i \in B}(\alpha - i)}

and g^{r(x)}

on-chain requires:

- Verification of r(x)

and \prod_{i \in B}(\alpha - i)

if they are provided as parameters; otherwise, polynomial interpolation to produce them.

- Two elliptic curve multiplication operations per point.
- Two elliptic curve addition operations per point.

To produce -C - g^{r(\alpha)}

, the verifier must also perform an elliptic curve addition operation in G1.

Credits

Many thanks to @barryWhiteHat, @kobigurk, @liangcc, @yingtong, and @lsankar4033 for their feedback, and @dankrad for his extremely helpful blog post on the topic.