Formal Verification of the c-kzg library: Establishing the Basis

Audit Report

Grant ID FY24-1544 — Ethereum Foundation

Brett Decker decker@galois.com
Ryan McCleeary mccleeary@galois.com
Roberto Saltini roberto.saltini@consensys.net
Thanh-Hai Tran thanh-hai.tran@consensys.net

Galois, Inc.
421 SW 6th Ave., Ste. 300
Portland, OR 97204

Consensys Software, Inc.

Contents

1. Audit Report

The goal of this audit report is to determine the adherence of the structure of the c-kzg implementation to the structure of its stated reference, the Polynomial Commitments section of the Python Deneb consensus specification (Python KZG Specification, hereafter) which encodes the KZG commitment scheme.

Identifying similarities and possible discrepancies between the two structures is key to the next two phases of this project. This comparison will guide us in structuring the Cryptol version of the Python KZG Specification in a way that enables subsequent verification against both the Python KZG specification itself and the c-kzg implementation.

It is important to note that determining the correctness of the c-kzg implementation with respect to the Python KZG Specification is beyond the scope of this document.

Moreover, determining the accuracy of the Python KZG Specification against the pseudocodes in the KZG paper is outside the scope of this project.

1.1. Audit of the Python KZG Specification against the algorithm in the KZG paper

The Python KZG Specification follows the ideas in the polynomial commitment scheme $\mathsf{PolyCommit}_{\mathsf{DL}}$ introduced in the KZG paper to commit data blobs. However, the Python KZG Specification has modifications for optimization, and the differences include:

1. Instead of using coefficient vectors, polynomials in the Python KZG Specification are represented in evaluation form, i.e., a vector of evaluations of the polynomial at the various `FIELD_ELEMENTS_PER_BLOB`(4096)-th roots of unity. Blobs are of Python Blob type and can be transformed to polynomials in evaluation form by calling the Python function `blob_to_polynomial`.
2. Polynomial evaluation and operations are also specified for the evaluation form, e.g., by using the Barycentric evaluation.
3. $\mathsf{PolyCommit}_{\mathsf{DL}}$ consists of six algorithms: Setup, Commit, Open, VerifyPoly, CreateWitness, and VerifyEval.
   (a) Setup is not formalized in the Python KZG Specification. The trusted setup is part of the `preset` and the setup information is stored in constants: `KZG_SETUP_G1`, `KZG_SETUP_G2_LENGTH`, `KZG_SETUP_G2`, and `KZG_SETUP_LAGRANGE`.
   (b) Commit is to output a commitment of a polynomial and potentially decommitment information to be used by the Open algorithm. In the Python KZG Specification, the commitment of a blob is computed by the Python function blob_to_kzg_commitment. No decommitment information is output by the Python KZG Specification.
   (c) Open is not included in the Python KZG Specification.
   (d) VerifyPoly is not also included in the Python KZG Specification.

(e) CreateWitness is to output the KZG proof for the evaluation of a polynomial at a specific point. It takes as input an evaluation point and a polynomial. It outputs a triple comprising the evaluation point, the evaluation of the polynomial at this point and the so called witness. The direct equivalent of CreateWitness in the Python KZG Specification is the function compute_kzg_proof_impl. A minor difference is that the proof output by compute_kzg_proof_impl, compared to the one output by CreateWitness, does not include the evaluation point provided as input. Also, in terms of terminology, the Python KZG Specification uses `proof` to refer to the witness part of the triple.

The Python KZG Specification also provides the function `compute_kzg_proof` which takes as input a blob rather than a polynomial.

Additionally, the Python KZG Specification provides `compute_blob_kzg_proof`. This function applies the Fiat-Shamir transformation and the hash function to randomly compute a point called `evaluation_challenge`, and then computes the KZG proof at this point. This proof only includes the witness.

Note that all of these functions deal with polynomials in evaluation form.

(f) VerifyEval is the verification algorithm. It takes as input a commitment $\mathcal{C}$, an evaluation point z, the evaluation $\phi(z)$ of a polynomial ϕ and a witness. It outputs whether the polynomial with commitment $\mathcal{C}$ evaluates to $\phi(z)$ on input z.

The direct equivalent of CreateWitness in the Python KZG Specification is the function verify_kzg_proof_impl.

The Python KZG Specification also provides `verify_blob_kzg_proof` where, in the input parameters of the function, the evaluation point and the evaluation are replaced by a blob data type.

In addition, the Python KZG Specification also provides the functions `verify_kzg_proof_batch` and `verify_blob_kzg_proof_batch` to verify multiple KZG proofs.

4. In the KZG paper, the authors use symmetric pairings; however, the BLS12-381 asymmetric pairing is applied in the Python KZG Specification.

1.2. Audit of the c-kzg Implementation against the Python KZG Specification

The Python KZG Specification entails 28 functions from four categories: Bit-reversal permutation, BLS12-381 helpers, Polynomials, and KZG. Table **??** identifies all of these functions, according to their category, with the GitHub references to Python KZG Specification and the c-kzg implementation.

In most cases the names of the functions in the Python KZG Specification match the c-kzg implementation. The differences include:

1. The Python function `multi_exp` is represented in the c-kzg implementaion by splitting multiplication into two functions, `g1_mul` and `g2_mul`, to have separate functions handle multiplication for the two groups, G1 and G2, separately, followed by a call to `blst_pX_add_or_double`;

2. A naming convention difference between the Python function `bls_field_to_bytes` and the c-kzg implementation function `bytes_from_bls_field`;

3. Two instances where the c-kzg implementation provides two implementing functions (for speed reasons) with `g1_lincomb_naive` and `g1_lincomb_fast` for Python function `g1_lincomb`, and similarly in the case of functions `blst_fr_inverse` and `blst_fr_eucl_inverse` for Python function `blst_modular_inverse` (though there appears currently to be no difference between these two BLST imported functions as defined in the BLST repository).

4. One instance where the c-kzg implementaion has the prefix `fr_`, for `fr_div`.

In addition, there are a couple of other slight differences. The computation specified in the Python function `compute_quotient_eval_within_domain` is handled inside the c-kzg implementation of the function `compute_kzg_proof_impl` instead of as a subroutine.

Another difference is in `blst_modular_inverse`, which uses the builtin Python function `pow`, whereas the corresponding BLST functions, as implented in `reciprocal_fr`, use BLST utility functions `ct_inverse_mod_256`, `redc_mont_256`, and `mul_mont_sparse_256`.

Finally, while both the Python spec and the c-kzg implementation have a function named `reverse_bits`, the c-kzg function does not match the spec. The spec for `reverse_bits` defines the computation as reversing the bits of a value for some specified bit length. The c-kzg function defines the computation as reversing all the bits of the value as defined by the size of the memory representation (in this case a 64-bit integer). The c-kzg implementation has a separate function, `reverse_bits_limited` that does match the Python spec. It should be noted that this spec equivalent function is not used in the implementation of `bit_reversal_permutation`. Instead `bit_reversal_permutation` calls the non-equivalent `reverse_bits` function, but it handles the result in a way that conforms to the Python spec, so that the overall computation is equivalent.

Table 1: Comparing c-kzg vs. Python KZG Specification

Category	Python KZG Specification	c-kzg implementation
Bit-reversal permutation	is_power_of_two reverse_bits bit_reversal_permutation	is_power_of_two reverse_bits bit_reversal_permutation
BLS12-381 helpers	multi_exp hash_to_bls_field bytes_to_bls_field bls_field_to_bytes validate_kzg_g1 bytes_to_kzg_commitment bytes_to_kzg_proof blob_to_polynomial compute_challenge bls_modular_inverse div g1_lincomb compute_powers compute_roots_of_unity	g1_mul , g2_mul hash_to_bls_field bytes_to_bls_field bytes_from_bls_field validate_kzg_g1 bytes_to_kzg_commitment bytes_to_kzg_proof blob_to_polynomial compute_challenge blst_fr_inverse , blst_fr_eucl_inverse fr_div g1_lincomb_naive , g1_lincomb_fast compute_powers compute_roots_of_unity
Polynomials	evaluate_polynomial _in_evaluation_form	evaluate_polynomial _in_evaluation_form
KZG	blob_to_kzg_commitment verify_kzg_proof verify_kzg_proof_impl verify_kzg_proof_batch compute_kzg_proof compute_quotient _eval_within_domain compute_kzg_proof_impl compute_blob_kzg_proof verify_blob_kzg_proof verify_blob_kzg_proof_batch	blob_to_kzg_commitment verify_kzg_proof verify_kzg_proof_impl verify_kzg_proof_batch compute_kzg_proof compute_kzg_proof_impl compute_kzg_proof_impl compute_blob_kzg_proof verify_blob_kzg_proof verify_blob_kzg_proof_batch

2. Next Steps

The next phase of this grant's work is to create a Cryptol specification that matches the Python KZG Specification.

After this second milestone, we will create a test bench that will verify the Cryptol specification against the Python KZG Specification, using the Python as the oracle for property based testing of the Cryptol. We will also verify properties for KZG correct construction, e.g. calling `compute_kzg_proof` with valid inputs for `blob` and `z_bytes` always results in successful verification when the output proof is passed to `verify_kzg_proof`.

After this third milestone, we will create a script using the Software Analysis Workbench (SAW) to attempt to formally verify and prove one or more of the Python KZG Specification functions as implemented in c-kzg is functionally equivalent to the corresponding Cryptol specification function, as well as memory safe.

After this fourth milestone, we will deliver a final report with our results and conclusions.