WG Working Group
Internet-Draft
Intended status: Informational
Expires: 24 November 2025

Y. T. Lai

23 May 2025

Polynomial Commitment Schemes
draft-zkproof-polycommit-00

Abstract

This document describes the high-level interface of a polynomial commitment scheme (PCS), a cryptographic primitive used in constructing generic zk-SNARKs. A PCS allows a prover to commit to a polynomial, and later attest to its correct evaluation at a given point. Test vectors and reference implementations for popular instantiations are provided in Appendix A.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://therealyingtong.github.io/draft-zkproof-polycommit/draft-zkproof-polycommit.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-zkproof-polycommit/.

Source for this draft and an issue tracker can be found at https://github.com/therealyingtong/draft-zkproof-polycommit.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 24 November 2025.

Table of Contents

1. Introduction

A polynomial commitment scheme (PCS) is a common building block in the construction of modern generic zk-SNARKs (Fig 1). It allows a prover to commit to a polynomial, and later prove its correct evaluation at a given point. This is used to instantiate oracle access to the prover's polynomial-encoded witness, by introducing a cryptographic hardness assumption (e.g. discrete logarithm hardness).

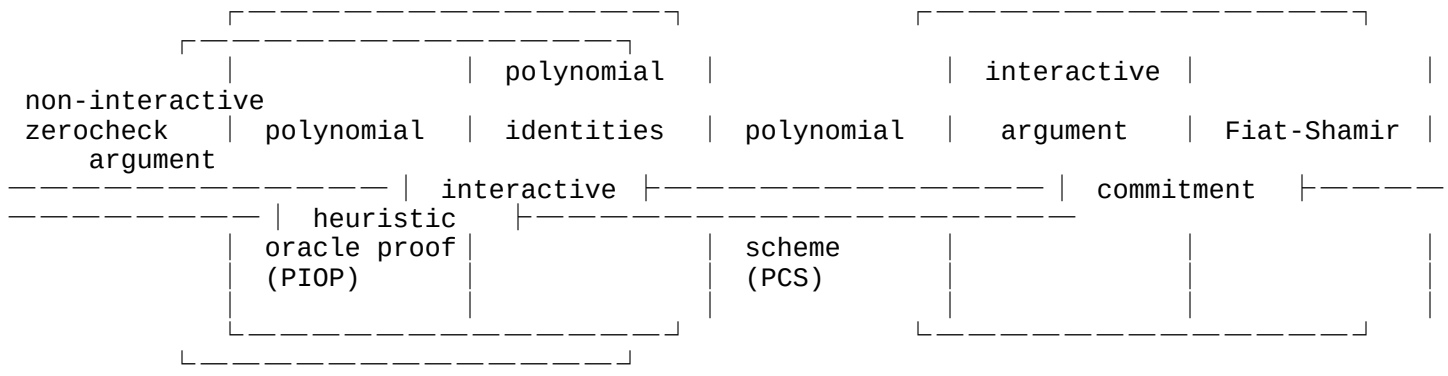

Fig 1: The components of a modern zk-SNARK.

A polynomial commitment scheme is parameterised over a finite field WitnessField for its representation, and a (potentially identical) finite field ChallengeField for its evaluation points. It is also parameterised over an underlying cryptographic hardness assumption, such as a collision-resistant hash function or the elliptic curve discrete logarithm problem.

NON-NORMATIVE NOTE: In small-field PCSes, challenges are usually drawn from an extension field ChallengeField.

1.1. Generic interface

A polynomial commitment scheme provides an interface with the following functions:

1.1.1. setup

On input the parameters security_bits, num_vars, degree_bounds, and an OPTIONAL randomness generator rng, setup samples a public CommitterKey and VerifierKey for the polynomial commitment scheme. NON-NORMATIVE NOTE: This generalises over both trusted setups (SRS) and transparent setups.

```
fn setup(
    security_bits: usize,
    num_vars: usize,
    degree_bounds: Vec<usize>,
    rng: Option<Rng>,
) -> (CommitterKey, VerifierKey)
```

Input:

* security_bits: the desired number of bits of security

* num_vars: the number of variables in the polynomial NON-NORMATIVE NOTE: For univariate polynomials, num_vars = 1

* degree_bounds: the upper bounds on the degree of each variable in the polynomial; degree_bounds.len() MUST equal num_vars NON-NORMATIVE NOTE: For multilinear polynomials, degree_bounds = 1 for each variable

* (OPTIONAL) rng: a randomness generator

Output:

* ck: a committer key used in computing commitments and opening proofs; this contains also the description of finite fields for the witness WitnessField, as well as for the challenge ChallengeField.

* vk: a verifier key used in verifying opening proofs; this contains also the description of finite fields for the witness WitnessField, as well as for the challenge ChallengeField.

1.1.2. commit

On input the committer key ck, a polynomial poly, and an OPTIONAL random blinding factor r, commit outputs a binding and optionally hiding commitment com.

```
fn commit(
    ck: CommitterKey,
    poly: Polynomial<WitnessField>,
    r: Option<WitnessField>
) -> (Commitment, CommitmentState)
```

Input:

* ck: the committer key

* poly: a polynomial with degree at most deg(X_i) = d_i D_i in each variable

* (OPTIONAL) r: a random element from the WitnessField; this can be set to None if the commitment is non-hiding NON-NORMATIVE NOTE: Zero-knowledge protocols often apply non-hiding polynomial commitment schemes to a "masked" polynomial, instead of the actual witness polynomial. The caller is responsible for masking the polynomial before providing it as input tocommit.

Output:

* com: a binding and optionally hiding polynomial commitment to poly

* com_state: auxiliary state of the commitment, containing information that can be re-used by the committer during the opening phase, such as the commitment randomness.

1.1.3. open

On input the committer key ck, a polynomial poly, a commitment com to the polynomial, the challenge point challenge, and the OPTIONAL random blinding factor r, open outputs the evaluation eval = poly(challenge), and an opening proof proof. The opening proof attests to the claim "com commits to a polynomial that evaluates to eval at challenge".

```
fn open(
    ck: CommitterKey,
    poly: Polynomial<WitnessField>,
    com: Commitment,
    com_state: CommitmentState,
    challenge: Challenge,
    r: Option<WitnessField>
) -> Proof
```

Input:

* ck: the committer key

* poly: a polynomial with degree at most deg(X_i) = d_i D_i in each variable

* com: a commitment to poly

* com_state: auxiliary state of the commitment

* challenge: the evaluation point at which com will be opened; this consists of num_vars elements from the ChallengeField NON-NORMATIVE NOTE: In the non-interactive setting, the challenge is derived from the commitment using the Fiat-Shamir transform [fiat-shamir].

* (OPTIONAL) r: a random element from the WitnessField; this MUST equal the r previously used in commit

Output:

* proof: an opening proof attesting to the correctness of the opening

1.1.4. verify

On input the verifier key vk, a polynomial commitment com, the evaluation point challenge, the purported opening eval, and the opening proof proof, verify checks the opening proof, and either accepts or rejects it.

```
fn verify(
    vk: VerifierKey,
    com: Commitment,
    challenge: Challenge,
    eval: ChallengeField,
    proof: Proof,
) -> bool
```

Input:

* vk: the verifier key

* com: a polynomial commitment

* challenge: the evaluation point at which com is opened

* eval: the purported evaluation of the committed polynomial at challenge

* proof: the opening proof the claim "com commits to a polynomial that evaluates to eval at challenge"

Output:

* verify outputs true if the opening proof is valid, and false otherwise

1.2. Batched setting

This section is NON-NORMATIVE.

Polynomial commitment schemes MAY support opening in a batched setting. In this setting, a single proof attests to the opening of multiple polynomials at multiple challenges (possibly different sets of challenges for each polynomial).

Common special cases of the batched setting include: - opening of a single polynomial at multiple challenges; and - opening of multiple polynomials at a single challenge

1.2.1. batch_open

On input the committer key ck, a vector of polynomials polys, a vector of their commitments coms, a vector of challenge sets challenges, and a vector of OPTIONAL random blinding factors rs, batch_open outputs the evaluations at each challenge set Vec<Vec<ChallengeField>> and a single opening proof BatchProof.

The opening proof attests to the claim that "com[i] commits to a polynomial poly[i] that opens to evals[i][j] at challenges[i][j]", for each index i in the batch of polynomials, and each index j in its corresponding challenge set.

```
fn batch_open(
    ck: CommitterKey,
    polys: Vec<Polynomial<WitnessField>>,
    coms: Vec<Commitment>,
    challenges: Vec<Vec<Challenge>>,
    rs: Vec<Option<ChallengeField>>
) -> (Vec<Vec<ChallengeField>>, BatchProof)
```

Input:

* ck: the committer key

* polys: the batch of polynomials to open

* coms: the commitments corresponding to polys; coms.len() MUST equal polys.len()

* challenges: the sets of challenge points at which to evaluate each polynomial; challenges.len() MUST equal polys.len()

* rs: the OPTIONAL random blinding factors used in each commitment; rs.len() MUST equal polys.len()

Output:

* evals: the evaluations of each polynomial at each challenge set; evals.len() MUST equal polys.len(), and each evals[j].len() MUST equal the corresponding challenges[j].len()

* batch_proof: an opening proof for the batch opening claim

1.2.2. batch_verify

On input the verifier key vk, a vector of commitments coms, a vector of challenge sets challenges, a vector of their purported corresponding evaluations evals, and an opening proof BatchProof, batch_verify checks the opening proof, and either accepts or rejects it.

```
fn batch_verify(
    vk: VerifierKey,
    coms: Vec<Commitment>,
    challenges: Vec<Vec<ChallengeField>>,
    evals: Vec<Vec<ChallengeField>>,
    proof: BatchProof,
) -> bool
```

Input:

* vk: the verifier key

* coms: a vector of polynomial commitments

* challenges: the sets of challenge points at which each commitment was opened; challenges.len() MUST equal coms.len()

* evals: the purported openings of each commitment at each challenge set; evals.len() MUST equal coms.len(), and each evals[j].len() MUST equal the corresponding challenges[j].len()

* batch_proof: an opening proof for the batch opening claim

Output:

* batch_verify outputs true if the opening proof is valid, and false otherwise

2. Concrete polynomial commitment schemes (WIP)

2.1. Ligero [AHIV17]

The Ligero [AHIV17] proof system can be used to instantiate a polynomial commitment scheme. It is parameterised over a collision-resistant hash function CRHScheme. The following interface is adapted from the arkworks library [arkworks].

Both the LigeroCommitterKey and LigeroVerifierKey are the same type LigeroParams:

```
struct LigeroParams<CRHScheme> {
    security_bits: usize,
    /// The rate of the Reed-Solomon code.
    code_rate: usize,
}
```

2.1.1. setup

```
fn setup<CRHScheme>(
    security_bits: usize,
    _num_vars: usize,
    _degree_bounds: Vec<usize>,
    _rng: Option<Rng>,
) -> (CommitterKey<CRHScheme>, VerifierKey<CRHScheme>) {
    let ck = LigeroParams<CRHScheme> {
        security_bits,
        code_rate = 4
    };
    let vk = LigeroParams<CRHScheme> {
        security_bits,
        code_rate = 4
    };
    (ck, vk)
}
```

2.1.2. commit

```
fn commit(
    ck: CommitterKey,
    poly: Polynomial<WitnessField>,
    r: Option<WitnessField>
) -> (Commitment, CommitmentState) {
    // 1. Arrange the coefficients of the polynomial into a matrix,
    // and apply encoding to get ‘ext_mat‘.

    // 2. Create the Merkle tree from the hashes of each column.

    // 3. Obtain the MT root

    (commitment, leaves)
}
```

2.1.3. open

```
  fn open(
      ck: CommitterKey,
      poly: Polynomial<WitnessField>,
      com: Commitment,
      com_state: CommitmentState,
      challenge: Challenge,
      r: Option<WitnessField>
  ) -> Proof {
      // 1. Create the Merkle tree from the hashes of each column.

      // 2. Generate vector ‘b‘ to left-multiply the matrix.

      // 3. left-multiply the matrix by ‘b‘.

      // 4. Generate t column indices to test the linear combination on.

      // 5. Compute Merkle tree paths for the requested columns.
  }
```

2.1.4. verify

```
fn verify(
    vk: VerifierKey,
    com: Commitment,
    challenge: Challenge,
    eval: ChallengeField,
    proof: Proof,
) -> bool {
    // 1. Ask random oracle for the ‘t‘ indices where the checks happen.

    // 2. Hash the received columns into leaf hashes.

    // 3. Verify the paths for each of the leaf hashes - this is only run once,

    // 4. Compute the encoding w = E(v).

    // 5. Compute ‘a‘, ‘b‘ to right- and left- multiply with the matrix ‘M‘.

    // 6. Probabilistic checks that whatever the prover sent,
    // matches with what the verifier computed for himself.
}
```

3. Security Considerations (WIP)

4. IANA Considerations

This document has no IANA actions.

Acknowledgments

The authors thank Mary Maller, Pierre Daix-Moreux, Oskar Thorn, Alex Kuzmin, and Manu Sporny, for reviewing a previous edition of this specification.

Informative References

[AHIV17] Ames, S., Hazay, C., Ishai, Y., and M. Venkitasubramaniam, "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup", 2017, <https://dl.acm.org/doi/10.1145/3133956.3134104>.

[arkworks] "arkworks zkSNARK ecosystem", <https://arkworks.rs>.

[fiat-shamir] "draft-orru-zkproofs-fiat-shamir", <https://mmaker.github.io/draft-zkproof-sigma-protocols/draft-orru-zkproof-fiat-shamir.html>.

Author’s Address

Ying Tong Lai
Email: yingtong.lai@gmail.com