CS409

Chalk and Talk

KZG Polynomial Commitment Scheme

Instructor: Prof. Sruthi Sekar

November 2024

Arnav Jain
22B2527

Arjun Jhunjhunwala
23B1245

Abstract

Polynomial commitment schemes are a fundamental cryptographic primitive that allow a prover to commit to a polynomial in such a way that they can later prove certain properties of the polynomial (such as evaluations at specific points) without revealing the entire polynomial. These schemes have applications in zero-knowledge proofs, secure multiparty computation, and blockchain technologies. A commitment scheme provides two essential properties: *hiding* and *binding*.

Hiding ensures that the commitment does not leak any information about the polynomial except for its evaluation at specific points. Binding guarantees that once a commitment is made, the prover cannot change the committed polynomial, preventing cheating or backtracking. A commitment is typically much smaller in size than the polynomial itself, offering efficiency advantages in cryptographic protocols. For example, in the Kate-Zaverucha-Goldberg (KZG) commitment scheme, the polynomial is committed to in a compressed form, typically as a single elliptic curve group element. This allows for fast, secure evaluation proofs, where the prover can demonstrate that a particular evaluation at a point s equals a specified value phi(s), without revealing the polynomial.

The KZG commitment scheme utilizes elliptic curve pairings to efficiently generate commitments, with security relying on the hardness of the discrete logarithm problem in elliptic curve groups. By leveraging a trusted setup phase, KZG provides a secure method of polynomial commitment, ensuring both binding and hiding properties. This makes it highly suitable for modern cryptographic applications like blockchain-based smart contracts, where verifiable yet private computations are required.

Contents

1 Theoretical Foundations

1.1 Algebraic Structures

The scheme uses elliptic curve pairings, which allow for the construction of compact, verifiable commitments to polynomials. Consider a prime field $\mathbb{F}_p$ and two elliptic curve groups G_1 and G_2 with the following properties:

- Bilinear pairing $e : G_1 \times G_2 \to G_T$
- Prime order p
- Generators $G \in G_1$ and $H \in G_2$

1.2 Trusted Setup Phase

The trusted setup involves:

1. Selecting a secret value $s \in \mathbb{F}_p$.
2. Generating public parameters $\{[s^i]_1, [s^i]_2\}_{i=0}^{n-1}$, where:
$$[s^i]_1 = s^i G \quad \text{and} \quad [s^i]_2 = s^i H,$$
with $G \in G_1$ and $H \in G_2$ being the generators.
3. Securely destroying the secret s.

2 Polynomial Commitment Construction

2.1 The Commitment Phase

To commit to a polynomial $\phi(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_t x^t$, where $\phi(x) \in \mathbb{F}_p[x]$, the commitment $C \in G_1$ is computed as:

$$C = [\phi(s)]_1 = \sum_{i=0}^{t} a_i [s^i]_1.$$

The public parameters $\{[s^i]_1\}$ enable the prover to efficiently compute C without knowledge of s. Each coefficient a_i is multiplied with $[s^i]_1$ and summed in G_1.

2.2 The Opening Phase

In this step, the verifier requests the prover to "open" the commitment C at a specific point $z \in \mathbb{F}_p$, where z is a random evaluation point. The prover evaluates $\phi(z)$ and provides the opening triplet:

$$OT = (z, \phi(z), [\psi(z)]_1),$$

where $\psi(x)$ is the proof polynomial constructed as:

$$\psi(x) = \frac{\phi(x) - \phi(z)}{x - z},$$

and $\psi(x)$ has degree $t - 1$.

2.3 The Verification Phase

The verifier checks that the opening triplet $OT = (z, \phi(z), [\psi(z)]_1)$ is valid, ensuring that $\phi(z)$ correctly evaluates $\phi(x)$, which is hidden in the commitment $C = [\phi(s)]_1$.

The verifier has access to:

- Public parameters $PP = \{[s^i]_1, [s^i]_2\}$,
- The commitment $C = [\phi(s)]_1$,
- The secret key s,
- The evaluation $\phi(s)$,
- The proof commitment $[\psi(z)]_1$,

and uses the bilinear pairing $e : G_1 \times G_2 \to G_T$, where G_1 and G_2 are elliptic curve groups, and G_T is the target group.

Without the pairing e, the verifier would need knowledge of s to verify the commitment C, as:

$$C = [\phi(s)]_1 = [\psi(z)]_1 \cdot [z - s]_1 + [\phi(s)]_1,$$

where $[\psi(z)]_1$ is the commitment to the proof polynomial $\psi(x)$. This would require the verifier to compute $[z - s]_1$, which depends on the secret s.

Using the pairing, the verifier checks the following equation:

$$e(C, H) = e([\psi(z)]_1, [z - s]_2) \cdot e([\phi(s)]_1, H),$$

where:

- $e(C, H) = e([\phi(s)]_1, H)$ represents the pairing of the commitment with the generator $H \in G_2$,
- $e([\psi(z)]_1, [z - s]_2)$ verifies the proof polynomial $\psi(x)$,
- $e([\phi(s)]_1, H)$ confirms the evaluation $\phi(s)$ at the point s.

Expanding this step-by-step:

$$\begin{aligned} e(C, H) &= e([\phi(s)]_1, H) \\ &= e([\psi(z)]_1 \cdot [z - s]_1 + [\phi(s)]_1, H) \\ &= e([\psi(z)]_1, [z - s]_2) \cdot e([\phi(s)]_1, H) \end{aligned}$$

by the bilinear property.

Since the verifier knows $[z - s]_2$ (via $[z]_2 - [s]_2$) and all other terms from PP, the validity of the opening triplet OT can be efficiently checked.

3 Binding and Hiding Properties

3.1 Binding Property

The binding property ensures that a prover cannot find two distinct polynomials $\phi(x)$ and $\psi(x)$ that produce the same commitment C. Suppose $\phi(x) \neq \psi(x)$ but $[\phi(s)]_1 = [\psi(s)]_1$. Then:

$$[\phi(s) - \psi(s)]_1 = [0]_1.$$

Since $[\phi(s) - \psi(s)]_1$ is a group element, the equation implies $\phi(s) - \psi(s) = 0$. However, $\phi(x) - \psi(x)$ is a polynomial of degree t, and the probability of s being one of its roots is negligible due to the field size. Thus, the scheme is binding.

Theorem 1 (Binding Security)**.** *The KZG scheme is binding with probability* $1 - \frac{t}{p}$*, where* t *is the degree of the polynomial and* p *is the size of the field.*

3.2 Hiding Property

The hiding property ensures that the commitment C reveals no information about $\phi(x)$ beyond its degree. Since $C = [\phi(s)]_1$, it is computed using scalar multiplication in G_1, which is computationally hard to reverse without knowledge of s.

Theorem 2 (Hiding Security)**.** *Under the discrete logarithm assumption in* G_1*, the commitment* C *does not reveal the coefficients of* $\phi(x)$ *or any evaluations other than those explicitly revealed by the prover.*

The security of the hiding property relies on the infeasibility of recovering s or $\phi(x)$ from $[\phi(s)]_1$, even with access to public parameters $\{[s^i]_1\}$.

4 Code Implementation

```
from sage.all import *

def safe_random_prime(k):
    """Generate a safe random prime."""
    while True:
        p = random_prime(2^k - 1, False, 2^(k - 1))  # Generating a
            random prime of bit length k
        if ZZ((p - 1) / 2).is_prime():  # Check if (p-1)/2 is prime for
             safety
             return p

k = 16  # Define bit length for the prime
p = safe_random_prime(k)
print(f"Our safe random prime is {p}")

# Define the prime field F_p
F_p = GF(p)
print(f"Finite Field F_p: {F_p}")

# Set the maximum degree for polynomials (t)
t = 0
while t == 0:
    t = F_p.random_element()  # Random degree from the finite field F_p
print(f"The maximum degree is {t}")

# Define the generator of the field
g = F_p.multiplicative_generator()
print(f"The generator is {g}")

# Define the pairing function
def e(g1, g2):
    return F_p.prod((g1, g2))

# Define s as the secret key in F_p (used in public parameters)
s = F_p.random_element()

# Compute public parameters (PP)
def compute_public_parameters(F_p, s, t):
    PP = []
    accumulated = 1
    for i in range(t + 1):
        PP.append(F_p.prod((accumulated, g)))  # Multiply accumulated
            value by g
        accumulated = F_p.prod((accumulated, s))  # Update accumulated
            for the next iteration
    return PP

PP = compute_public_parameters(F_p, s, t)
print("The first elements of the public parameters are: {}".format(PP
    [:100]))

# Function to encode a message using Reed-Solomon encoding
from reedsolo import RSCodec
def reed_solomon_encode_string(input_string):
    # Initialize Reed-Solomon encoder with error correction level
    rs = RSCodec(10)
    # Encode the string into bytes and convert to a list of integers
    encoded_bytes = rs.encode(input_string.encode())
    return list(encoded_bytes)

# Convert the message into coordinates
def message_as_coords(input_string):
    y = reed_solomon_encode_string(input_string)
    x = range(1, len(y) + 1)
    return list(zip(x, y))

msg = message_as_coords("Hello Zepp!")
print(f"Message as coordinates: {msg}")

# Define the polynomial ring over F_p
F_p_X.<x> = PolynomialRing(F_p)

# Create the Lagrange polynomial for the message
msg_poly = F_p_X.lagrange_polynomial(msg)
print(f"Polynomial for the message: {msg_poly}")

# Function to evaluate the polynomial on the exponents
from functools import reduce
def exponent_evaluation(PP, poly, g):
    # Recover the base field F_p from the polynomial
```

```python
    F_p = poly.base_ring()
    # Get polynomial coefficients in ascending order
    poly_coefs = poly.coefficients()
    # Use only the necessary public parameters
    pp_used = PP[:len(poly_coefs)]

    return reduce(lambda a, b: a + (b[0] * b[1]), zip(poly_coefs,
       pp_used), 0)

# Compute the commitment to the polynomial
commitment = exponent_evaluation(PP, msg_poly, g)
print(f"Commitment: {commitment}")

# Choose a random point for evaluation (z is the evaluation point, not
    the secret s)
z = F_p.random_element()
print(f"The point of evaluation ‘z‘ is {z}")

# Evaluate the polynomial at z
poly_eval = msg_poly(z)
print(f"The evaluation at z is {poly_eval}")

# Construct the proof polynomial
proof_poly = ((msg_poly - poly_eval) / (x - z)).numerator()
print(f"Proof polynomial: {proof_poly}")

# Compute the opening commitment
opening_commit = exponent_evaluation(PP, proof_poly, g)
print(f"Opening commitment: {opening_commit}")

# The opening triplet
opening_triplet = (z, poly_eval, opening_commit)
print(f"Opening triplet: {opening_triplet}")

# Final verification equation (check using pairings)
assert e(commitment, g) == (e(opening_commit, PP[1] - z * g) + (
    poly_eval * e(g, g))), "Verification failed!"
```

Listing 1: Python implementation of the KZG Polynomial Commitment Scheme

5 What if s is Leaked?

In the event that the secret key s is leaked, the security of the KZG Polynomial Commitment Scheme is compromised. The commitment can be broken by revealing the value $\phi(s)$, and an adversary can then generate a fake polynomial that passes through $(s, \phi(s))$ and other evaluation points. This section demonstrates the potential impact of such a leak and how an adversary can deceive the system.

```python
# Assuming s has been leaked
print(f"Oh no, ‘s‘ leaked: {s}")

# The original commitment, which was computed securely before the leak
print(f"Original commitment: {commitment}")

# Since the commitment is [phi(s)]_1 and is publicly known, we can now
    compute phi(s)
```

```python
phi_s = commitment / g  # Compute phi(s) from the commitment
print(f"Now we know the value of phi(s): {phi_s}")

# Now, let's construct a fake polynomial that passes through (s, phi(s))
# and also through some point (z, phi(z)) where z is a random element from F_p
def make_faux_poly(F_p_X, s, phi_s, z, phi_z):
    xs = [s, z]  # x-values (the points where we have evaluations)
    ys = [phi_s, phi_z]  # y-values (the corresponding evaluations)
    F_p = F_p_X.base_ring()

    # Add a third point to make the polynomial a quadratic (or higher degree)
    while len(xs) != 3:
        x = F_p.random_element()
        if x not in xs:
            xs.append(x)
            ys.append(F_p.random_element())  # Random y-value for the new point

    # Return the Lagrange polynomial for the points
    return F_p_X.lagrange_polynomial(zip(xs, ys))

# Generate a faux polynomial
faux_poly = make_faux_poly(F_p_X, s, phi_s, z, poly_eval)
print(f"Fake polynomial created: {faux_poly}")

# Evaluate the faux polynomial at z
faux_poly_eval = faux_poly(z)
print(f"The evaluation of z on the faux polynomial is {faux_poly_eval}")

# Construct the faux proof polynomial
faux_proof_poly = ((faux_poly - faux_poly_eval) / (x - z)).numerator()
print(f"The faux proof polynomial: {faux_proof_poly}")

# Compute the faux opening commitment for the faux polynomial
faux_opening_commit = exponent_evaluation(PP, faux_proof_poly, g)

# Form the faux opening triplet
faux_opening_triplet = (z, faux_poly_eval, faux_opening_commit)
print(f"The faux opening triplet is {faux_opening_triplet}")

# Final verification: Does the fake polynomial generate the same commitment?
faux_verification = e(commitment, g) == (e(faux_opening_commit, PP[1] - z * g) + (faux_poly_eval * e(g, g)))
```

Listing 2: Python code to demonstrate the effect of a leaked secret key s

We deceive the system and verify the faux polynomial as if it were the true polynomial.

6 Conclusion

The KZG Polynomial Commitment Scheme is a robust cryptographic primitive that provides efficient and secure polynomial commitments. Its binding and hiding properties ensure both integrity and privacy, making it a cornerstone in cryptographic protocols.

7 References

1. Feist, D. (2020, June 16). *Kate Polynomial Commitments.* Retrieved from `https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html`
2. Juneer, K. (2021, February 6). *Explaining KZG Commitment with Code Walkthrough.* `https://kaijuneer.medium.com/`
3. Tomescu, A. (2020, May 6). *KZG Polynomial Commitments.* Retrieved from `https://alinush.github.io/2020/05/06/kzg-polynomial-commitments.html`
4. Saha, N. (2024). *CS741 Project Report, Advanced Network Security and Cryptography.*

Libraries Used in the Code

The following libraries were used in the Python code for implementing the KZG Polynomial Commitment Scheme:

- **SageMath (sage.all)**: A powerful open-source mathematics software system that provides functionality for finite fields, polynomial rings, elliptic curve cryptography, and more. It was used for finite field arithmetic, polynomial operations, and elliptic curve pairings.
- **ReedSolomon (reedsolo)**: A Python library for Reed-Solomon encoding and decoding, used for error correction in the message encoding process.