Improved Polynomial Division in Cryptography

Kostas Kryptos Chalkias[1], Charanjit Jutla[2], Jonas Lindstrøm[1], Varun Madathil[3], and Arnab Roy[1]

[1] Mysten Labs, Palo Alto, CA
[2] IBM Research, Yorktown, NY
[3] Yale University, New Haven, CT

Abstract. Several cryptographic primitives, especially succinct proofs of various forms, transform the satisfaction of high-level properties to the existence of a polynomial quotient between a polynomial that interpolates a set of values with a cleverly arranged divisor. Some examples are SNARKs, like Groth16, and polynomial commitments, such as KZG. Such a polynomial division naively takes $O(n \log n)$ time with Fast Fourier Transforms, and is usually the asymptotic bottleneck for these computations.
Several works have targeted specific constructions to optimize these computations and trade-off one-time setup costs with faster online computation times. In this paper, we present a unified approach to polynomial division related computations for a diverse set of schemes. We show how our approach provides a common abstract lens which recasts and improves existing approaches. Additionally, we present benchmarks for the Groth16 and the KZG systems, illustrating the significant practical benefits of our approach in terms of speed, memory, and parallelizability. We get a speedup of $2\times$ over the state-of-the-art in computing all openings for KZG commitments and a speed-up of about $2-3\%$ for Groth16 proofs when compared against the Rust Arkworks implementation. Although our Groth16 speedup is modest, our approach supports twice the number of gates as Arkworks and SnarkJS as it avoids computations at higher roots of unity. Conversely this reduces the need for employing larger groups for bigger circuits. For example, our approach can support 2^{28} gates with BN254, as compared to 2^{27} for coset-based approaches, without sacrificing computational advantages.
Our core technical contributions are novel conjugate representations and compositions of the derivative operator and point-wise division under the Discrete Fourier Transform. These allow us to leverage l'Hôpital's rule to efficiently compute polynomial division, where in the evaluation basis such divisions maybe of the form 0/0. Our techniques are generic with potential applicability to many existing protocols.

1 Introduction

Polynomial divisions play a very important role in various cryptographic applications, especially in the domain of zero-knowledge proofs. With the advent of succinct non-interactive arguments of knowledge (SNARKs), zero-knowledge proofs have gained immense popularity due to their efficiency and scalability, enabling practical applications in blockchain technologies and privacy-preserving computations.

A typical recipe for constructing a SNARK involves combining a polynomial commitment scheme with an Interactive Oracle Proof (IOP). In this framework, the prover and verifier engage in an interactive protocol where the prover sends messages that the verifier can query at arbitrary positions, effectively treating them as oracles. The IOP allows for checks on certain properties of the computation by querying these oracles, enhancing the efficiency and scalability of the proof system.

Polynomial commitment schemes are essential in this setting because they enable the prover to commit to polynomials used in the computation and later prove properties about them without revealing the polynomials themselves. The Kate, Zaverucha, and Goldberg (KZG) commitment scheme [KZG10] is widely used for this purpose due to its succinctness and efficiency. The KZG scheme leverages polynomial division to efficiently verify polynomial evaluations, making it a critical component in SNARK protocols.

In these SNARK constructions, polynomial division plays a pivotal role. For example, when a prover needs to prove that a committed polynomial $f(x)$ evaluates to a certain value at a specific point, they often compute a quotient polynomial $q(x) = \frac{f(x)-f(z)}{x-z}$. This operation inherently involves polynomial division and is essential for generating the proof that the verifier can efficiently check.

Protocols such as Sonic [MBKM19], Marlin [CHM+20], and Plonk [GWC19] follow this paradigm by combining polynomial commitment schemes with IOPs to achieve efficient and scalable zero-knowledge proofs. The reliance on polynomial divisions in these protocols underscores the importance of optimizing polynomial division operations to improve the overall efficiency of SNARK systems.

Other Cryptographic Applications of Polynomial Division. Beyond zero-knowledge proofs, polynomial division plays a significant role in other cryptographic domains. For example, in error-correcting codes, such as Reed-Solomon and Bose–Chaudhuri–Hocquenghem (BCH) codes, polynomial division is fundamental to encoding and decoding processes. These codes use polynomial division to detect and correct errors in data transmission and storage. Notable works by Forney [For65] and the Berlekamp-Welch algorithm [WB83] have refined these techniques, influencing subsequent research in theoretical and applied cryptography.

In secure multiparty computation (MPC), polynomial division is essential for secret sharing schemes. Shamir's secret sharing [Sha79] divides a secret into shares using polynomials, and reconstructing the secret involves polynomial interpolation and division. Subsequent MPC protocols [MGW87, BGW88] have built upon these principles to enable secure computation among multiple parties, ensuring privacy and correctness even in the presence of malicious actors.

Computation Complexity of Polynomial Division in Cryptography. In cryptographic applications, polynomial divisions are often performed over finite fields and involve polynomials of high degrees. Traditional algorithms for polynomial division have a computational complexity of $\mathcal{O}(n^2)$ where n is the degree of the polynomial. To improve efficiency, algorithms leveraging the Fast Fourier Transform (FFT) have been adopted, reducing the complexity to $\mathcal{O}(n \log n)$. These FFT-based methods enable faster polynomial multiplication and division, which are crucial for high-performance cryptographic protocols.

Despite these optimizations, polynomial division remains a computational bottleneck, particularly in resource-constrained environments or when dealing with very high-degree polynomials common in modern SNARK systems.

1.1 Our Contributions

Polynomial multiplications are efficiently performed by evaluating the multiplicand polynomials at roots of unity by using FFT, point-wise multiplying the evaluations, and then reverting back to the coefficient form by an inverse FFT. A similar recipe works for polynomial division as well. However, this approach fails if the numerator and denominator polynomials are both 0 at some or all of the evaluation points.

1. We provide a novel formal linear algebraic framework for doing polynomial division efficiently. We comprehensively cover cases where the evaluation basis may have a $0/0$ form, by leveraging l'Hôpital's rule. On the way to achieve this, we derive novel conjugate representations of the derivative operator under the discrete Fourier transform.
2. We provide novel algorithmic approaches for two widely used cryptographic constructions: KZG vector commitments and Groth16 zkSNARKs. For both constructions, we achieve more elegant representations than similar other works in the literature. [4] We compare our algorithms against the best optimizations in the literature that we know of and achieve competitive efficiency in all cases. We also achieve qualitative advancement and substantial practical benefits in some cases, including better amenability to parallelization.
3. These algorithmic advances are also applicable to several other proof systems as well, such as STARK [BBHR18], Plonk [GWC19], Aurora [BCR+19], Marlin [CHM+20], Spartan [Set20], and so on, which use polynomial divisions extensively. We describe how to approach inner product arguments (IPA) based on univariate sumchecks in our framework. We also briefly go over how our framework can be utilized for STARK and PLONK.

1.2 Comparison with Previous Work

There have been many recent works that have shown efficient polynomial division in the above mentioned cryptographic applications. The most salient of these that have performance comparable to our contribution are detailed below. However, we emphasize that while the benchmarks we obtain offer practical benefits, our main focus is on developing a comprehensive linear-algebraic, and

[4] Our benchmarks are open-sourced here: `https://github.com/MystenLabs/polydiv`.

more precisely a linear-operator based theory for obtaining fast algorithms. We now briefly describe two competing algorithms (in their respective cryptographic applications):

KZG Commitments. Feist and Khovratovich [FK23] present a construction to compute n KZG proofs in $O(n \log n)$ time. This is achieved by employing a few well-known techniques in a clever and judicious manner: (a) the bi-variate polynomial $\frac{f(X)-f(Y)}{X-Y}$ has a representation such that the coefficients (arranged in a matrix) is a Toeplitz matrix T formed from coefficients of f, (b) The Toeplitz form is easily extended to be a circulant matrix, which then allows multiplication of T into given powers of a secret $X = s$ (hidden in the exponent of a hard group) to be just a convolution, which can be computed in time $\mathrm{O}(n \log n)$, (c) the evaluations on different values of Y can be computed using known algorithms for computing a polynomial at multiple points. More details can be found in Section 4.4. While this is an innovative use of known techniques, our approach allows for the possibility of further practical optimization as we obtain closed form representations for evaluating all proofs simultaneously.

Groth16 SNARK. Popular implementations of the Groth16 SNARK, such as SnarkJS [SNA] and Arkworks compute $f(X)/t(X)$, where $f(X)$ is a multiple of $t(X)$, and $t(X)$ has roots at roots of unity, using a coset FFT [Ber07]. For more details, see Section 5.3. We show that this can instead be computed using the derivative operator, the main theme of this work. Polynomial division via coset FFTs is performed using the $2n$-th roots of unity to avoid encountering issues with $\frac{0}{0}$ form. The use of $2n$-th roots of unity implies that the coset approach can only support half the number of gates as our approach when instantiated with the same bilinear group.

1.3 Paper Organization

We start with preliminaries in Section 2 to explain all the notations and background concepts. Then we give a technical overview and explain linear algebraic tools and techniques in Section 3. Then we describe our approach and algorithms, compare with existing works and provide evaluation and benchmarks for two cryptographic constructions: KZG vector commitments in Section 4, and Groth16 SNARKs in Section 5. Finally, we describe our approach for univariate sumchecks in Section 6. We also briefly describe a couple of more applications of our technique in Appendix E.

2 Preliminaries

Notations. In the subsequent sections λ is our security parameter. $\mathbb{G}_1$ and $\mathbb{G}_2$ are a group of prime order p, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear pairing [MVO91, Jou00]. In this work, we present all group operations using additive notation i.e., $[a]_k$ represents a group element in $\mathbb{G}_k$ [EHK$^+$13].

The primitive n-th root of unity in (some finite extension field of) $\mathbb{Z}_p^*$ is represented by ω. Typically, p and n are chosen so that this root of unity is in $\mathbb{Z}_p^*$ itself. We denote DFT as the Vandermonde matrix with rows induced by powers of ω. We will follow the convention that rows and columns start with the index 0. The i-th entry of a vector $\boldsymbol{v}$ is denoted as $(\boldsymbol{v})_i$, and the (i,j)-th entry of a matrix M is denoted as $(\mathsf{M})_{i,j}$. The transpose of a matrix M is denoted $\mathsf{M}^\top$. In particular $(\mathsf{DFT})_{i,j} = \omega^{ij}$. The Hadamard product, or entry-wise product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is denoted $\boldsymbol{a} \circ \boldsymbol{b}$. The notation $\textbf{pow}(x)$ denotes the vector of powers of x: $[1\ x\ x^2\ \cdots x^{n-1}]^\top$. The notation $\mathbf{1}$ denotes a vector of all entries equal to 1, that is, $[1\ 1\ 1\ \cdots 1]^\top$.

Fourier Transforms. The Discrete Fourier Transform (DFT) matrix is a structured $n \times n$ matrix that facilitates the transformation of vectors from the time (or spatial) domain to the frequency domain. In the context of polynomials, the DFT matrix can be used to evaluate polynomials at the roots of unity. Given a polynomial $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ with coefficients $\{a_0, a_1, \ldots, a_{n-1}\}$ multiplying by the DFT matrix effectively evaluates this polynomial at the roots of unity ω^i. This process converts the polynomial from its coefficient representation to its point form, making subsequent operations like multiplication more efficient. If one has the evaluations of a polynomial at the roots of unity, then using the inverse DFT matrix (DFT^{-1}), one can compute the corresponding polynomial coefficients.

The operation $\widehat{\mathsf{A}} = \ (\mathsf{DFT} \cdot \mathsf{A} \cdot \mathsf{DFT}^{-1})$ is an example of a *similarity transform.* We will call this resulting matrix $\widehat{\mathsf{A}}$ as the *conjugate* of the matrix A. If A represents a linear transformation acting on polynomial coefficients, then $\widehat{\mathsf{A}}$ corresponds to how this transformation behaves when the polynomial is expressed in its point-value form at these roots of unity. This change of basis is particularly useful because certain operations, such as polynomial multiplication, become much simpler (often element-wise) in the transformed domain. Therefore, the conjugate matrix $\widehat{\mathsf{A}}$ can be seen as the 'frequency domain' representation of A capturing how A interacts with polynomials evaluated at these special points.

A square matrix will be called *sparse* if it has only $O(n)$ non-zero entries. We will leverage the fact that sparse matrices with closed form entries can be multiplied with a vector in $O(n)$ time. A sparse matrix will be called *star-shaped* if its only non-zero entries are the diagonal, k-th row and k-th column, for some k. We will also use the fact that, when n is a power of 2, multiplication of a vector by DFT and DFT^{-1} matrices can be performed in $O(n \log n)$ time by the Fast Fourier Transform (FFT) algorithm [CT65]. More precisely, the Cooley-Tukey FFT algorithm is an in-place butterfly algorithm requiring $\log n$ rounds, with each round requiring $n/2$ butterfly-steps. A butterfly step takes two inputs a, b and outputs $a + \tau \cdot b$ and $a - \tau \cdot b$, for some scalar τ. Note that a, b can be in an elliptic-curve group of order p. Then τ is typically in the multiplicative group of scalars $\mathbb{Z}_p^*$. As can be seen, the total number of (elliptic-curve) scalar-multiplications is then $\log n * (n/2)$ (in addition to $n \log n$ group additions/subtractions). The inverse FFT can also be computed in a similar way, by just using ω^{-1} in place of ω. It's worth noting that the Cooley-Tukey in-place algorithm produces the output in an index bit-reversed fashion. So, if the same algorithm (i.e. using the butterfly-step mentioned above) is to be used to compute the inverse, one must permute the input and output array when computing the inverse FFT.

Polynomial and Vector Commitment Schemes. In a polynomial commitment scheme [KZG10] the prover commits to a polynomial f and later opens it to $f(x_i)$ for some x_i. A polynomial commitment scheme consists of the following algorithms: $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$. A polynomial commitment scheme can be thought of as a vector commitment scheme where the vector committed to are the evaluations of the polynomial. In this context there are two more algorithms - $\mathsf{UpdateCom}$ and $\mathsf{UpdateOpen}$. We present the syntax for vector commitments below, since that is the focus of our work:

– $\mathsf{Setup}(\lambda) \rightarrow pp$: generates public parameters for the commitment scheme.
– $\mathsf{Commit}(pp, \boldsymbol{v}) \rightarrow C$: This algorithm takes as input the vector $\boldsymbol{v}$ and outputs a commit C.
– $\mathsf{Open}(pp, \boldsymbol{v}, i) \rightarrow \pi_i$: This algorithm takes as input the vector $\boldsymbol{v}$ and an index i and outputs a proof π_i that proves that the value at index i is $(\boldsymbol{v})_i$.
– $\mathsf{Verify}(pp, C, \pi_i, (\boldsymbol{v})_i, i) \rightarrow b$: This algorithm takes as input the commitment C, the value at position i and verifies if the proof of opening is valid. This algorithm outputs a bit 1 if it verifies.
– $\mathsf{UpdateCom}(pp, C, i, v_i', v_i) \rightarrow C'$: This algorithm takes as input the commitment C, the original value at index i and the new value at index i and outputs a new commitment C' with the value at position i updated to v_i'.
– $\mathsf{UpdateOpen}(pp, \pi_j, j, i, v_i', v_i) \rightarrow \pi_j'$: This algorithm takes as input the proof π_j, the original value at index i - v_i and the new value v_i' and outputs a new proof π_j'. The algorithm to update the proof of opening in the case $i = j$ and $i \neq j$ may be different.

Succinct Non-Interactive Arguments of Knowledge - SNARKs. SNARKs are non-interactive systems with short proofs that enable verifying NP computations with substantially lower complexity than that required for classical NP verification. A SNARK is typically described by three algorithms:

– $\mathsf{Setup}(\lambda) \rightarrow \mathsf{crs}$ is a setup algorithm that is typically run by a trusted party. This algorithm outputs a common random string crs.
– $\mathsf{Prove}(\mathsf{crs}, x, w) \rightarrow \pi$ is run by the prover and takes as input a statement x, a witness w and outputs a succinct proof π.
– $\mathsf{Verify}(\mathsf{crs}, x, \pi) \rightarrow b$ is run by the verifier and takes as input the crs, the statement x and a proof π and outputs 1 if the proof is valid.

Most constructions and implementations of SNARKs [PHGR16, Lip13, DFGK14, Gro16, GMNO18] make use of quadratic programs (introduced in [GGPR13]). This framework allows to build SNARKs for statements that can be represented as an arithmetic or boolean circuit. In this work we focus on the Groth16 [Gro16] construction. We will present more details on the same in Section 5.1.

Linear Operators. A linear operator $\Phi : V \rightarrow V$ on a vector space V over a field $\mathbb{F}$ satisfies the following two properties: (i) $\Phi(\boldsymbol{v}_1 + \boldsymbol{v}_2) = \Phi(\boldsymbol{v}_1) + \Phi(\boldsymbol{v}_2)$, and (ii) for all $c \in \mathbb{F}$, $\Phi(c \cdot \boldsymbol{v}) = c \cdot \Phi(\boldsymbol{v})$. In this work we will be interested in linear operators on a vector space of fixed degree (say, $n-1$) polynomials over a field $\mathbb{F}$. Thus, any such linear operator can be represented by a $n \times n$ matrix. One interesting operator we analyze is CDiv_a, which transforms a polynomial f to $\frac{f(x)-f(a)}{X-a}$. Let's first check that this is indeed a linear operator by noting that $\mathsf{CDiv}_a(f_1 + f_2) = \frac{(f_1+f_2)(x)-(f_1+f_2)(a)}{X-a} = \mathsf{CDiv}_a(f_1) + \mathsf{CDiv}_a(f_2)$, and also $\mathsf{CDiv}_a(c \cdot f) = c \cdot \mathsf{CDiv}_a(f)$.

The particular matrix representation of this linear operator depends on the basis we choose for degree $n-1$ polynomials, e.g. the power basis consisting of $1, x, x^2, ...$, or the FFT or evaluation basis consisting of the power basis transformed by the vandermonde matrix $\mathbf{V}$ of n-th roots of unity (in some finite extension field of $\mathbb{F}$). We denote these roots of unity by ω^k ($k \in [0..n-1]$).

Of particular interest are the linear operators CDiv_{ω^k}, which by abuse of notation we will just denote by CDiv_k. In the evaluation basis, this operator is then just taking $f(\omega^j)$ to $\frac{f(\omega^j)-f(\omega^k)}{\omega^j-\omega^k}$. For the special case of $j = k$ the above expression is $0/0$, but by l'Hôpital's Rule for polynomials over arbitrary fields (see Theorem 2), this is same as $f'(\omega^j)$.

While in the power basis the linear operator's matrix representation will be called CDiv_k itself, in the evaluation basis the matrix representation will be called EDiv_k. Thus, $\mathsf{EDiv}_k = \widehat{\mathsf{CDiv}_k} = \mathsf{DFT} \cdot \mathsf{CDiv}_k \cdot \mathsf{DFT}^{-1}$. A little calculation shows that EDiv_k is a sparse star-shaped matrix, and moreover it is intimately related to the derivative linear operator – see Theorem 1 for details.

3 Technical Overview

All polynomial operations, such as evaluation, addition, subtraction, multiplication, and division can be represented as linear algebraic operations on both the coefficient space, that is, the vector of coefficients, and the evaluation space, that is, the vector of evaluations on a predefined vector of points.

Simple addition, subtraction, and scaling of polynomials have direct correspondence between the coefficient space and the evaluation space. The standard high-school method of multiplying two polynomials given in coefficient representation is $O(n^2)$. However, it is much more straightforward in the evaluation space, where the corresponding operation is just point-wise multiplication. This observation is leveraged in the $O(n \log n)$ Fast Fourier Transform (FFT) algorithm for multiplying two polynomials.

3.1 Division in the Evaluation Space

The point-wise multiplication method can be extended to division as well, with a couple of remarks. Firstly, the point-wise division would correspond to polynomial division only in the case the denominator polynomial exactly divides the numerator polynomial. Secondly, the point-wise division fails to work if both the numerator and denominator evaluations are 0 at least at one evaluation point.

Under the assumption that the first condition holds, we extend the FFT-based method of dividing polynomials using the l'Hôpital's rule. While l'Hôpital's rule is well-known for functions over complex numbers, it also holds for polynomials in arbitrary fields. Although this is also known, we give a proof in Appendix A for completeness.

A high level template for division in this framework is as follows. First observe that the derivative operation is a linear shift and scale operation in the coefficient space, based on $\frac{d}{dx}a_i x^i = i a_i x^{i-1}$. Let D stand for the derivative operator, as formally described in Table 1. Let the operation required be $f(X)/g(X)$:

1. Compute $\boldsymbol{f}' = \mathsf{D}\boldsymbol{f}$ and $\boldsymbol{g}' = \mathsf{D}\boldsymbol{g}$ in $O(n)$ time.
2. Compute $\boldsymbol{v} = \mathsf{DFT}\boldsymbol{f}$, $\boldsymbol{w} = \mathsf{DFT}\boldsymbol{g}$, $\boldsymbol{v}' = \mathsf{DFT}\boldsymbol{f}'$, $\boldsymbol{w}' = \mathsf{DFT}\boldsymbol{g}'$, in $O(n \log n)$ time.
3. Collect point-wise divisions of $\boldsymbol{v}$ with $\boldsymbol{w}$. For points of $0/0$ form collect the corresponding point-wise division from the derivative evaluations $\boldsymbol{v}', \boldsymbol{w}'$.
4. Apply DFT^{-1} to this synthesized vector to compute the quotient in coefficient space.

Note that the above approach fails if $(\boldsymbol{w})_i = (\boldsymbol{w}')_i = 0$ at some index i. A sufficient condition to prevent this is to ensure that $g(X)$ is square-free. This is because in the square-free case $g(X)$ and $g'(X)$ will not have a common root, in particular, any ω^i. For the applications we consider in this paper the denominator polynomial will always be square-free.

Applying a linear algebra lens. Recall that $\mathbf{pow}(x)$ denotes the vector $[1\ x\ x^2 \cdots x^{n-1}]^\top$ where $n-1$ is an upper bound on the polynomial degrees. The evaluation of a polynomial $f(X)$ at a point x can be represented equivalently as:

$$f(x) = \mathbf{pow}(x)^\top \boldsymbol{f} = \mathbf{pow}(x)^\top \mathsf{DFT}^{-1}\boldsymbol{v}$$

Now observe that if $\mathsf{deg}(f) \leq (n-2)$, then $Xf(X)$ is a polynomial that shifts the coefficients from x^i to x^{i+1} for each i. This is a linear transform in the coefficient space, represented by the off-diagonal matrix M in Table 1. Equivalently:

$$xf(x) = \mathbf{pow}(x)^\top \mathsf{M}\boldsymbol{f} = \mathbf{pow}(x)^\top \mathsf{M} \cdot \mathsf{DFT}^{-1}\boldsymbol{v}$$

We can generalize this with the observation that multiplying powers of x corresponds to further applications of the M operator. For example, $x^2 f(x) = \mathbf{pow}(x)^\top \mathsf{M}^2 \boldsymbol{f}, \cdots, x^i f(x) = \mathbf{pow}(x)^\top \mathsf{M}^i \boldsymbol{f}$, and so on, for suitable restrictions on the degree of f. Carrying this to further generalization, we have that $p(x)f(x) = \mathbf{pow}(x)^\top p(\mathsf{M})\boldsymbol{f}$, with the condition that $\mathsf{deg}(p) + \mathsf{deg}(f) \leq (n-1)$.

Carrying this operation in reverse presents some problems. Observe that M is not full-ranked. As a result, writing $f(x)/x$ as $\mathbf{pow}(x)^\top \mathsf{M}^{-1}\boldsymbol{f}$ doesn't work as M^{-1} does not exist. Instead let's attempt to represent the quotient $\frac{f(X)-f(\omega^k)}{X-\omega^k}$, which is guaranteed to be a polynomial. Note that we can write $f(\omega^k) = \mathbf{pow}(x)^\top \mathsf{E}_{0,k} \cdot \mathsf{DFT}\boldsymbol{f}$, where $\mathsf{E}_{0,k}$ is the single-entry matrix defined in Table 1. This holds because the operator matrix $\mathsf{E}_{0,k} \cdot \mathsf{DFT}$ applies the k-th row of the DFT matrix to $\boldsymbol{f}$, thereby evaluating f at ω^k. Thus we can write the operator for $\frac{f(X)-f(\omega^k)}{X-\omega^k}$ as[5]:

$$\mathsf{CDiv}_k = (\mathsf{M} - \omega^k \mathsf{I})^{-1}(\mathsf{I} - \mathsf{E}_{0,k} \cdot \mathsf{DFT})$$

For familiar readers, a straightforward representation of bivariate polynomial $\frac{f(X)-f(Y)}{X-Y}$ is well-known in terms of a Toeplitz matrix obtained from coefficients of f (see e.g. [Con, Theorem 3.7] or [FK23]). Thus, $\mathsf{CDiv}_k \cdot \boldsymbol{f}$ is this polynomial with $Y = \omega^k$. We discuss more details in Appendix D.

However, this does not give us a sparse matrix operator representation. Surprisingly, its conjugate operator has a sparse representation. The conjugate of this matrix is the corresponding operator in the evaluation space:

$$\mathsf{EDiv}_k = \widehat{\mathsf{CDiv}}_k = (\widehat{\mathsf{M}} - \omega^k \mathsf{I})^{-1}(\mathsf{I} - \mathsf{DFT} \cdot \mathsf{E}_{0,k})$$

To derive this expression, we use the fact that the conjugation operation distributes over additions, multiplications, and inversions of matrices.

We show that the matrix EDiv_k is a sparse matrix with a special structure which is intimately related to the conjugate of the derivative D operator. The structure enables $O(n)$ computation of $\frac{f(X)-f(\omega^k)}{X-\omega^k}$ in the evaluation space. This novel result enables fast computation of openings of KZG vector commitments as we will see in a later section. Moreover, we show how to "stack" all the sparse EDiv_k matrices to result in matrices whose conjugates have a sparse structure, thus enabling the computation of all openings in $O(n \log n)$ time.

3.2 Useful Matrices and Transforms

We list below some special matrices in Table 1 and a correspondence between several polynomial operations in the coefficient space and evaluation space in Table 2.

Some observations useful for the derivations are detailed in the following theorem.

Theorem 1. *In any field F which contains a primitive n-th root of unity ω, we have:*

(i) *Let D be the derivative operator from Table 1. The derivative conjugate matrix $\widehat{\mathsf{D}}$ has the following explicit structure:*

$$(\widehat{\mathsf{D}})_{ij} = \begin{cases} \frac{\omega^{j-i}}{\omega^i - \omega^j}, & \text{for } i \neq j \\ \frac{(n-1)}{2\omega^i}, & \text{for } i = j \end{cases}$$

(ii) *The matrix EDiv_k is defined as $\mathsf{EDiv}_k = (\widehat{\mathsf{M}} - \omega^k \mathsf{I})^{-1}(\mathsf{I} - \mathsf{DFT} \cdot \mathsf{E}_{0,k})$. The k-th row of EDiv_k is same as k-th row of $\widehat{\mathsf{D}}$. That is, $(\mathsf{EDiv}_k)_{k,*} = (\widehat{\mathsf{D}})_{k,*}$. Equivalently, $\mathsf{E}_{0,k} \cdot \mathsf{EDiv}_k = \mathsf{E}_{0,k} \cdot \widehat{\mathsf{D}}$.*

[5] Assume degree of f is at most $n - 1$.

Matrix	Explicit Form of Entry (i,j)	Example with $n = 4$ and $\omega = \zeta_4$ a primitive 4-th root of unity.
$\mathsf{E}_{k,l}$	$\begin{cases} 1 & ,(i,j) = (k,l) \\ 0 & ,\text{otherwise} \end{cases}$	$\mathsf{E}_{2,3} = \begin{pmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&1 \\ 0&0&0&0 \end{pmatrix}$
DFT	ω^{ij}	$\mathsf{DFT} = \begin{pmatrix} 1&1&1&1 \\ 1&\zeta_4&-1&-\zeta_4 \\ 1&-1&1&-1 \\ 1&-\zeta_4&-1&\zeta_4 \end{pmatrix}$
M	$\begin{cases} 1 & ,i = j+1 \\ 0 & ,\text{otherwise} \end{cases}$	$\mathsf{M} = \begin{pmatrix} 0&0&0&0 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \end{pmatrix}$
$\widehat{\mathsf{M}} = \mathsf{DFT} \cdot \mathsf{M} \cdot \mathsf{DFT}^{-1}$	$\begin{cases} -\frac{1}{n}\omega^j & ,i \neq j \\ \frac{n-1}{n}\omega^j & ,i = j \end{cases}$	$\widehat{\mathsf{M}} = \frac{1}{4} \begin{pmatrix} 3&-\zeta_4&1&\zeta_4 \\ -1&3\zeta_4&1&\zeta_4 \\ -1&-\zeta_4&-3&\zeta_4 \\ -1&-\zeta_4&1&-3\zeta_4 \end{pmatrix}$
D	$\begin{cases} j & ,j = i+1 \\ 0 & ,\text{otherwise} \end{cases}$	$\mathsf{D} = \begin{pmatrix} 0&1&0&0 \\ 0&0&2&0 \\ 0&0&0&3 \\ 0&0&0&0 \end{pmatrix}$
$\widehat{\mathsf{D}} = \mathsf{DFT} \cdot \mathsf{D} \cdot \mathsf{DFT}^{-1}$	$\begin{cases} \frac{\omega^{j-i}}{\omega^i - \omega^j} & ,i \neq j \\ \frac{n-1}{2}\omega^{-i} & ,i = j \end{cases}$	$\widehat{\mathsf{D}} = \frac{1}{4} \begin{pmatrix} 6 & 2\zeta_4 - 2 & -2 & -2\zeta_4 - 2 \\ 2\zeta_4 - 2 & -6\zeta_4 & 2\zeta_4 + 2 & 2\zeta_4 \\ 2 & 2\zeta_4 + 2 & -6 & -2\zeta_4 + 2 \\ -2\zeta_4 - 2 & -2\zeta_4 & -2\zeta_4 + 2 & 6\zeta_4 \end{pmatrix}$
J	$\begin{cases} \frac{1}{\omega^i - \omega^j} & ,i \neq j \\ \frac{n-1}{2}\omega^{-i} & ,\text{otherwise} \end{cases}$	$\mathsf{J} = \frac{1}{4} \begin{pmatrix} 6 & 2\zeta_4 + 2 & 2 & -2\zeta_4 + 2 \\ -2\zeta_4 - 2 & -6\zeta_4 & -2\zeta_4 + 2 & -2\zeta_4 \\ -2 & 2\zeta_4 - 2 & -6 & -2\zeta_4 - 2 \\ 2\zeta_4 - 2 & 2\zeta_4 & 2\zeta_4 + 2 & 6\zeta_4 \end{pmatrix}$
$\widehat{\mathsf{J}} = \mathsf{DFT} \cdot \mathsf{J} \cdot \mathsf{DFT}^{-1}$	$\begin{cases} n - i & ,j = i-1 \\ & \text{ and } i \in [1, n-1] \\ 0 & ,\text{otherwise} \end{cases}$	$\widehat{\mathsf{J}} = \begin{pmatrix} 0&0&0&0 \\ 3&0&0&0 \\ 0&2&0&0 \\ 0&0&1&0 \end{pmatrix}$

Table 1: Matrix Notations

Polynomial Operation	Coefficient Basis	Evaluation Basis
$f(x) = \mathbf{pow}(x)^\top \boldsymbol{f} = \mathbf{pow}(x)^\top \mathsf{DFT}^{-1}\boldsymbol{v}$	$\boldsymbol{f}$	$\boldsymbol{v}$
$f(\omega^k)$	$\mathsf{E}_{0,k} \cdot \mathsf{DFT}\boldsymbol{f}$	$\mathsf{DFT} \cdot \mathsf{E}_{0,k}\boldsymbol{v}$
$f(x) + a$	$\boldsymbol{f} + a\boldsymbol{e}_0$	$\boldsymbol{v} + a\mathbf{1}$
$af(x)$	$a\boldsymbol{f}$	$a\boldsymbol{v}$
$xf(x),\ \mathsf{deg}(f) \leq n-2$	$\mathsf{M}\boldsymbol{f}$	$\widehat{\mathsf{M}}\boldsymbol{v}$
$p(x)f(x),\ \mathsf{deg}(p) + \mathsf{deg}(f) \leq n-1$	$p(\mathsf{M})\boldsymbol{f}$	$p(\widehat{\mathsf{M}})\boldsymbol{v}$
$\frac{d}{dx}f(x)$	$\mathsf{D}\boldsymbol{f}$	$\widehat{\mathsf{D}}\boldsymbol{v}$
$\frac{f(x) - f(\omega^k)}{x - \omega^k}$	$\mathsf{CDiv}_k\boldsymbol{f} = (\mathsf{M} - \omega^k\mathsf{I})^{-1}(\mathsf{I} - \mathsf{E}_{0,k} \cdot \mathsf{DFT})\boldsymbol{f}$	$\mathsf{EDiv}_k\boldsymbol{v} = (\widehat{\mathsf{M}} - \omega^k\mathsf{I})^{-1}(\mathsf{I} - \mathsf{DFT} \cdot \mathsf{E}_{0,k})\boldsymbol{v}$

Table 2: Representations of Polynomial Operations.

(iii) EDiv_k *is a star-shaped matrix with the following explicit form:*

$$(\mathsf{EDiv}_k)_{(i,j)} = \begin{cases} \frac{1}{\omega^i - \omega^k} & , i = j \textit{ and } i \neq k \\ \frac{-\omega^{j-k}}{\omega^j - \omega^k} & , i = k \textit{ and } j \neq k \\ -\frac{1}{\omega^i - \omega^k} & , j = k \textit{ and } i \neq k \\ \frac{n-1}{2}\omega^{-k} & , i = j = k \\ 0 & , \textit{otherwise} \end{cases}$$

Proof. Theorem 1(*i*) is proved in Appendix C.

Observe that $\mathbf{pow}(x)^\top \mathsf{E}_{0,k}\mathsf{EDiv}_k\boldsymbol{v}$ is the evaluation of $\frac{f(X)-f(\omega^k)}{X-\omega^k}$ at ω^k, which happens to have a $0/0$ form. By l'Hôpital's theorem, this evaluation is also equal to $f'(\omega^k) = \mathbf{pow}(x)^\top \mathsf{E}_{0,k}\mathsf{DFT}\cdot\mathsf{D}\boldsymbol{f} = \mathbf{pow}(x)^\top \mathsf{E}_{0,k}\mathsf{DFT}\cdot\mathsf{D}\cdot\mathsf{DFT}^{-1}\boldsymbol{v} = \mathbf{pow}(x)^\top \mathsf{E}_{0,k}\widehat{\mathsf{D}}\boldsymbol{v}$. This establishes Theorem 1(*ii*).

The other rows of EDiv_k induce the evaluation space computation of $\frac{v_i - v_k}{\omega^i - \omega^k}$, which do not have a $0/0$ form. This is represented by the rest of the structure of EDiv_k:

$$\begin{cases} \frac{1}{\omega^i - \omega^k} & , i = j \text{ and } i \neq k \\ -\frac{1}{\omega^i - \omega^k} & , j = k \text{ and } i \neq k \\ 0 & , \text{otherwise} \end{cases}$$

This establishes Theorem 1(*iii*).

4 KZG Vector Commitments with Efficient Openings

Kate, Zaverucha, and Goldberg [KZG10] proposed a constant-sized commitment scheme for polynomials, known as KZG commitments. A KZG commitment allows one to commit to a polynomial $f(x)$ such that the commitment C can be opened to any value $f(\alpha)$ for a given α. Notably, if we have a vector of values $\boldsymbol{v}$, we can compute a vector commitment by first constructing a polynomial $V(x)$ such that $V(\alpha_i) = v_i$ for each $v_i \in \boldsymbol{v}$, and then using the KZG polynomial commitment scheme to commit to the polynomial $V(x)$.

4.1 Background

To create a KZG commitment, start with a polynomial $V(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$. The commitment C is defined as $[V(\tau)]_1$, where τ is a secret, and the powers of τ are generated during setup as $[\mathbf{pow}(\tau)]_1 = [[1]_1\ [\tau]_1\ \dots\ [\tau^{n-1}]_1]$.

To open the commitment at a point α and prove that $V(\alpha) = v$, the proof π is computed as follows. First, compute the quotient polynomial $Q(x) = \frac{V(x)-v}{x-\alpha}$, and then evaluate it at τ to obtain $[Q(\tau)]_1$. Verification involves checking that the provided proof π satisfies the equation

$$e(C - [v]_1, [1]_2) = e(\pi, [\tau]_2 - [\alpha]_2),$$

where e is a bilinear pairing. This equation holds because $V(\tau) - v = Q(\tau)(\tau - \alpha)$.

To compute the proof of opening, π, the prover needs to first compute the polynomial $Q(x)$ and then compute $[Q(\tau)]_1$. Naively, this approach requires first to compute the polynomial V which takes $O(n^2)$ time by Lagrange interpolation and do the polynomial division which takes $O(n^2)$ time. One could optimize this further by choosing the points of evaluation as the n-th roots of unity (denoted ω) and then use FFT transforms to interpolate the polynomial in $O(n \log n)$ time. In the following section we will present our approach to improve the efficiency of computing the proofs of openings and also updating commitments and updating the proofs of openings.

4.2 Our Approach

In this section we present our approach to improve the concrete running time for computing the proof of opening. Moreover, we show how to compute all openings in just $O(n \log n)$ time. The standard approach would have taken time $O(n^2 \log n)$.

Finally, we also present algorithms for updating the commitments and proofs of opening. We refer the reader to Figure 1 for all complete algorithms.

- **Setup**: The setup algorithm first computes the powers of τ exactly as in the original KZG commitment scheme. Along with that the algorithm also outputs two vectors $[\boldsymbol{w}]_1 \in \mathbb{G}_1^n$ and $[\boldsymbol{u}]_1 \in \mathbb{G}_1^n$. The vector $[\boldsymbol{w}]_1$ enables us to compute the commitment with just the vector of elements $\boldsymbol{v}$, without computing the polynomial that is interpolated by these elements. The $[\boldsymbol{w}]_1$ is computed as

$$[\boldsymbol{w}]_1 = \mathsf{DFT}^{-1} \cdot \mathbf{pow}(\tau)$$

We also compute another vector $[\boldsymbol{u}]_1$ which will be used to support fast update of openings, as we will see later. Let J be the matrix obtained by stacking all the k-th columns of EDiv_k across all k:

$$\mathsf{J} = \begin{cases} \frac{1}{\omega^i - \omega^j} & , i \neq j \\ \frac{n-1}{2}\omega^{-i} & , i = j \end{cases}$$

It turns out that the conjugate matrix $\widehat{\mathsf{J}}$ is a sparse matrix:

$$\widehat{\mathsf{J}} = \begin{cases} n - i & , j = i - 1 \text{ and } i \in [1, n-1] \\ 0 & , \text{otherwise} \end{cases}$$

Now we compute $[\boldsymbol{u}]_1$ in $O(n \log n)$ time as:

$$\begin{aligned} [\boldsymbol{u}]_1 = \mathsf{J} \cdot [\boldsymbol{w}]_1 &= \mathsf{DFT}^{-1} \cdot \widehat{\mathsf{J}} \cdot \mathsf{DFT} \cdot \mathsf{DFT}^{-1} \cdot [\mathbf{pow}(\tau)]_1 \\ &= \mathsf{DFT}^{-1} \cdot \widehat{\mathsf{J}} \cdot [\mathbf{pow}(\tau)]_1 \end{aligned}$$

- **Commit**: As mentioned above, we do not need to interpolate the vector $\boldsymbol{v}$, since we compute the vector $[\boldsymbol{w}]_1$ in the setup. Thus the commitment Com can be computed as

$$\mathsf{Com} = \boldsymbol{v}^\top [\boldsymbol{w}]_1$$

Setup(τ):
- Let $(n = 2^k)$ powers of τ : $[\mathbf{pow}(\tau)]_1 = ([1]_1, [\tau]_1, [\tau^2]_1, \ldots, [\tau^{n-1}]_1) \in \mathbb{G}^n$
- Let $[\boldsymbol{w}]_1 = \mathsf{DFT}^{-1} \cdot [\mathbf{pow}(\tau)]_1$
- Let $[\boldsymbol{u}]_1 = \mathsf{DFT}^{-1} \cdot \widehat{\mathsf{J}} \cdot [\mathbf{pow}(\tau)]_1$, where $\widehat{\mathsf{J}}$ is the sparse matrix defined as in Table 1.
- Output $pp = ([\tau]_2, [\boldsymbol{w}]_1, [\boldsymbol{u}]_1)$.

Commit $(pp, \boldsymbol{v})$: Output $\mathsf{Com}_V = \langle \boldsymbol{v}, [\boldsymbol{w}]_1 \rangle$

Open at index i $(pp, \boldsymbol{v}, i)$: Output:

$$\pi_i = [\boldsymbol{w}]_1^\top \mathsf{EDiv}_k \boldsymbol{v} = \sum_{j \neq i} \frac{v_j - v_i}{\omega^j - \omega^i} [(\boldsymbol{w})_j]_1 + \{(\widehat{\mathsf{D}})_{(i,*)} \boldsymbol{v}\} [(\boldsymbol{w})_i]_1,$$

where $\widehat{\mathsf{D}}$ is defined as in Table 1.

Open all indices $(pp, \boldsymbol{v})$: Output:

$$\boldsymbol{\pi}_{all} = [\boldsymbol{w}]_1 \circ \widehat{\mathsf{D}} \boldsymbol{v} + (\mathsf{ColEDiv} \cdot [\boldsymbol{w}]_1) \circ \boldsymbol{v} + \mathsf{DiaEDiv} \cdot ([\boldsymbol{w}]_1 \circ \boldsymbol{v}).$$

This algorithm is explained in Section 4.3.

Verify opening $(pp, \mathsf{Com}_V, v_i, \pi_i)$: Check:

$$e(\mathsf{Com}_V - [v_i]_1, [1]_2) = e(\pi_i, [\tau]_2 - [\omega^i]_2).$$

Update commitment $(pp, \mathsf{Com}_V, i, v_i', v_i)$: Output:

$$\mathsf{Com}_V' = \mathsf{Com}_V + (v_i' - v_i)[(\boldsymbol{w})_i]_1.$$

Update opening $(pp, \pi_j, j, i, v_i', v_i)$:
- If $j \neq i$, output $\pi_j' = \pi_j + (v_i' - v_i) \cdot (\frac{1}{\omega^i - \omega^j}[(\boldsymbol{w})_i]_1 + \frac{\omega^{i-j}}{\omega^j - \omega^i}[(\boldsymbol{w})_j]_1)$
- If $j = i$, output $\pi_j' = \pi_j + (v_i' - v_i)[(\boldsymbol{u})_i]_1$.

Fig. 1: KZG commitments with efficient openings

- **Open at index** i: To open at index i, the original KZG algorithm required to compute the polynomial $Q_i(x) = \frac{V(x)-v_i}{x-\omega^i}$ and then compute $[Q_i(\tau)]_1$. To compute Q_i we would first need to interpolate V using $\boldsymbol{v}$. We observe that we don't actually need to calculate these polynomials. Recall that the proof of opening is $[Q_i(\tau)]_1$. This can be evaluated by using $(n-1)$ points ω^j as

$$Q_i(\omega^j) = \frac{v_j - v_i}{\omega^j - \omega^i}$$

and one more point at ω^i. But note that the point at ω^i which is $\frac{V_i(\omega^i)-v_i}{\omega^i-\omega^i}$ is in the $\frac{0}{0}$ form. We therefore need to use l'Hôpital's rule, and just need to compute $V_i'(\omega^i)$. Then the polynomial $[Q_i(\tau)]_1$ can be computed as:

$$[Q_i(\tau)]_1 = \sum_{j \neq i} \frac{v_j - v_i}{\omega^j - \omega^i}[\boldsymbol{w}_j]_1 + V'(\omega^i)[\boldsymbol{w}_i]_1$$

As we have discussed before, we can compute $V'(\omega^i)$ directly in the evaluation space, without interpolating the polynomial and then computing the derivative. Given the explicit form of the derivative conjugate matrix $\widehat{\mathsf{D}} = \mathsf{DFT} \cdot \mathsf{D} \cdot \mathsf{DFT}^{-1}$, we can compute the evaluations of the polynomial V' by simply computing $\widehat{\mathsf{D}}\boldsymbol{v}$. Since $(\widehat{\mathsf{D}})_{i,j}$ has the form:

$$\frac{\omega^{j-i}}{\omega^i - \omega^j} \text{ if } i \neq j \text{ and } \frac{n-1}{2\omega^i}, \text{ if } i = j \tag{1}$$

we can compute

$$V'(\omega^i) = \sum_{j \neq i} v_j \cdot \frac{\omega^{j-i}}{\omega^i - \omega^j} + v_i \cdot \frac{(n-1)}{2\omega^i}$$

Overall, this is just explicitly computing the action of the operator EDiv_k, by noting that $[Q_i(\tau)]_1 = [\mathbf{pow}(\tau)]_1^\top \mathsf{EDiv}_k \boldsymbol{v}$.

- **Verify opening proofs:** The verification algorithm is exactly as in the original KZG construction with a single bilinear pairing check. This ensures full compatibility between the original scheme and our optimized version.

$$e(\mathsf{Com}_V - [v_i]_1, [1]_2) = e(\pi_i, [\tau]_2 - [\omega^i]_2)$$

- **Update commitment:** When the value at index i, i.e. v_i is updated to v_i', then the naive approach to compute the updated commitment would be to simply recompute $\langle \boldsymbol{v'}, [\boldsymbol{w}]_1 \rangle$, where $\boldsymbol{v'}$ is the same as $\boldsymbol{v}$ except at position i. We observe that using $[\boldsymbol{w}]_i$ we can update the commitment simply as:

$$\mathsf{Com}_V' = \mathsf{Com}_V + (v_i' - v_i)[\boldsymbol{w}_i]_1$$

- **Update proof of opening:** Consider the case above where the value at index i has been updated from v_i to v_i'. The proof of opening is now not valid

anymore. To this end, one could recompute the proof of opening at index i as in the opening algorithm, and this would cost $O(n \log n)$. We show a more efficient $O(1)$ algorithm to update the proof of opening.
Let us first consider the case when index i does not correspond to the index j at which the proof of opening is to be updated, then the new opening π'_j can be computed. Recall that the proof of opening is computed as

$$\pi_j = \sum_{k \neq j} \frac{v_k - v_j}{\omega^k - \omega^j} [\boldsymbol{w}_k]_1 + V'(\omega^j)[\boldsymbol{w}_j]_1$$

Substituting v'_i instead of v_i in the first half we get,

$$\sum_{k \neq j} \frac{v_k - v_j}{\omega^k - \omega^j} [\boldsymbol{w}_k]_1 + \frac{(v'_i - v_i)}{\omega^i - \omega^j} [\boldsymbol{w}_i]_1$$

Substituting v'_i instead of v_i in the derivative polynomial V':

$$\begin{aligned} V'(\omega^i)' &= \sum_{j \neq i} v_j \cdot \frac{\omega^{j-i}}{\omega^i - \omega^j} + v'_i \cdot \frac{(n-1)}{2\omega^i} \\ &= V'(\omega^i) + (v'_i - v_i) \cdot \frac{(n-1)}{2\omega^i} \end{aligned}$$

Combining these two equations we get:

$$\pi'_j = \pi_j + (v'_i - v_i) \cdot \left(\frac{1}{\omega^i - \omega^j} [(\boldsymbol{w})_i]_1 + \frac{\omega^{i-j}}{\omega^j - \omega^i} [(\boldsymbol{w})_j]_1 \right)$$

Now let us consider the case when index i is the index at which the proof must be updated. This is represented by the action of the i-th column of EDiv_i, which we already incorporated as the i-th element of the setup vector $[\boldsymbol{u}]_1$. In this case, the proof can be updated using the vector $[\boldsymbol{u}]_1$ as follows:

$$\pi_i = \pi_i + (v'_i - v_i)[(\boldsymbol{u})_i]_1$$

4.3 Computing all KZG Openings in $O(n \log n)$ time

Recall, we intend to compute $[\boldsymbol{w}]_1^\top \cdot \mathsf{EDiv}_k \cdot \boldsymbol{v}$, for all $k \in [0, n-1]$. Also, recall the structure of EDiv_k from Theorem 1 *(iii)*:

$$(\mathsf{EDiv}_k)_{(i,j)} = \begin{cases} \frac{1}{\omega^i - \omega^k} & , i = j \text{ and } i \neq k \\ \frac{-\omega^{j-k}}{\omega^j - \omega^k} & , i = k \text{ and } j \neq k \\ -\frac{1}{\omega^i - \omega^k} & , j = k \text{ and } i \neq k \\ \frac{n-1}{2}\omega^{-k} & , i = j = k \\ 0 & , \text{otherwise} \end{cases}$$

We now decompose and stack all the EDiv_k matrices as follows, leveraging their star structure:

1. The stacking of all the k-th rows of EDiv_k is just the derivative conjugate matrix $\widehat{\mathsf{D}}$ (by Theorem 1 (*ii*)).
2. Define $\mathsf{ColEDiv}$ as the stacking of all the k-th columns of EDiv_k, with the diagonal entries set to 0, over all k:

$$\mathsf{ColEDiv} = \begin{cases} -\frac{1}{\omega^j - \omega^i} & , i \neq j \\ 0 & , i = j \end{cases}$$

(since for $j = k$, $(\mathsf{EDiv}_k)_{(i,j)} = -\frac{1}{\omega^i - \omega^k}$).
3. Define $\mathsf{DiaEDiv}$ as the stacking of all the diagonals of EDiv_k converted to columns, with the diagonal entries set to 0:

$$\mathsf{DiaEDiv} = \begin{cases} \frac{1}{\omega^j - \omega^i} & , i \neq j \\ 0 & , i = j \end{cases}.$$

In fact, turns out that $\mathsf{DiaEDiv} = -\mathsf{ColEDiv}$.

To enable the reader to understand how we stack the rows, columns, and diagonals of each EDiv_k we present an illustration with $n = 4$ in Appendix F.

The vector of all openings is a careful sum over the three operators defined above:

1. The stacking of the rows operates on the evaluation vector. The resulting vector from this operation multiplies entry-wise to the powers-of-tau vector, that is, as a Hadamard product. More precisely, we compute $[\boldsymbol{w}]_1^\top \cdot \text{diagonal}(\widehat{\mathsf{D}}\boldsymbol{v})$. which is conveniently represented (as a columns vector) by $[\boldsymbol{w}]_1 \circ \widehat{\mathsf{D}}\boldsymbol{v}$.
2. The stacking of the columns operates on the powers-of-tau vector. The resulting vector from this operation multiplies entry-wise to the evaluation vector as a Hadamard product. This contribution is represented as $(\mathsf{ColEDiv} \cdot [\boldsymbol{w}]_1) \circ \boldsymbol{v}$.
3. The stacking of the diagonals as columns operates on the Hadamard product of the evaluation vector with the powers-of-tau vector. This contribution is represented as $\mathsf{DiaEDiv} \cdot ([\boldsymbol{w}]_1 \circ \boldsymbol{v})$.

Given the above observations the vector of all KZG openings is:

$$[\boldsymbol{w}]_1 \ \circ \ \widehat{\mathsf{D}}\boldsymbol{v} \ + (\mathsf{ColEDiv} \cdot [\boldsymbol{w}]_1) \ \circ \ \boldsymbol{v} \ + \mathsf{DiaEDiv} \ \cdot \ ([\boldsymbol{w}]_1 \circ \boldsymbol{v}) \tag{2}$$

The DFT conjugates of $\widehat{\mathsf{D}}, \mathsf{ColEDiv}, \mathsf{DiaEDiv}$ are all sparse matrices. A bit of algebra (see proof in Appendix C) shows that:

$$\begin{gathered} -(\widehat{\mathsf{ColEDiv}})_{i,j} = (\widehat{\mathsf{DiaEDiv}})_{i,j} = \\ \begin{cases} \frac{n-1}{2} & , (i,j) = (0, n-1) \\ i - \frac{n+1}{2} & , j = i - 1 \text{ and } i \in [1, n-1] \\ 0 & , \text{otherwise} \end{cases} \end{gathered} \tag{3}$$

So the vector of all openings can be computed in $O(n \log n)$ time.

This summarizes our approach to achieve concrete efficiency in computing openings and updates. We remark that in our construction the proof of opening can be computed with just multi-scalar multiplication operations and can be parallelized. In the next section we will compare our approach with two other approaches that also extend the KZG construction to achieve faster proofs of openings.

4.4 Other Approaches

Feist and Khovratovich [FK23] present a construction to compute n KZG proofs in $O(n \log n)$ time. They observe that the coefficients of the polynomial Q can be computed with just n scalar multiplications in the following way:

$$Q_v(x) = \sum_{i=0}^{n-1} q_i X_i, \quad q_{n-1} = V_n, \quad q_j = V_{j+1} + v \cdot q_{j+1}$$

Note that their approach requires a sequential computation of the coefficients q_i, whereas our approach is highly parallelizable using multi-scalar multiplications directly in the evaluation space. They also present a formula for computing n KZG proofs in $O(n \log n)$ time. While we achieve the same asymptotic computation time, our approach is more elegant, and simple.

Their technique leverages FFTs to handle polynomial evaluations efficiently. The key innovation lies in constructing a polynomial $h(X)$ whose evaluations at specific points yield the required KZG proofs. This method ensures that for n evaluation points, the proofs can be computed in $O((n + d) \log(n + d))$ group operations if the points are roots of unity, or $O(n \log^2 n + d \log d)$ otherwise.

The coefficients of the polynomial $h(X)$ are computed using a Toeplitz matrix formed from the coefficients of the original polynomial $V(X)$ and the evaluation points. Multiplying a vector by a Toeplitz matrix can be efficiently performed in $O(n \log n)$ time [FK23], reducing the complexity of the operations involved. Specifically, the technique involves computing the Discrete Fourier Transform (DFT) of the vector of polynomial coefficients and the vector of powers of the evaluation points, followed by element-wise multiplication and an inverse DFT to compute all the KZG proofs of openings.

Tomescu et al. [TAB$^+$20] present a construction for an aggregatable sub-vector commitment (aSVC) scheme. An aSVC scheme is a vector commitment that allows aggregation of multiple subvector proofs into a single small subvector proof. Specifically, they extend KZG commitments to allow for proving multiple proofs of opening. Their setup algorithm is similar to ours in that they generate $\boldsymbol{\ell} = [g^{\mathcal{L}_i \tau}]_{i \in [n]}$, which is the same as our $[\boldsymbol{w}]_1$ (here $\mathcal{L}_i$ is the Lagrange basis polynomial) and also $\boldsymbol{u} = [g^{\frac{\mathcal{L}_i(\tau)-1}{\tau - \omega^i}}]$ which is the same as our $\boldsymbol{u}$. They require another group element $a = g^{A(\tau)}$ and another vector of group elements $\boldsymbol{a} = g^{\frac{A(\tau)}{\tau - \omega^i}}$. Thus their setup is larger than ours.

Computing the KZG commitment is done similar to our approach, by making use of the vector $\boldsymbol{\ell}$. They present a construction to compute the opening for a single point using n exponentiations and n scalar multiplications by making

	Cost to open	Cost to open all indices	Cost to update ($j \neq i$)	Cost to update ($j = i$)	Setup size
[FK23]	seq $\mathcal{O}(n)$ mult	$\mathcal{O}(n \log n)$	-	-	$n\lvert\mathbb{G}\rvert$
[TAB+20]	$\mathcal{O}(1)$ exp + $\mathcal{O}(n)$ mult	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$4n\lvert\mathbb{G}\rvert$
Our approach	$\mathcal{O}(n)$ mult	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$2n^*\lvert\mathbb{G}\rvert$

Table 3: Comparing different approaches to computing KZG commitments. (*) If we precompute $\mathsf{ColEDiv} \cdot [\boldsymbol{w}]_1$ in the setup, our setup size is $3n|\mathbb{G}|$.

use of the public parameters $\boldsymbol{u}$ and $\boldsymbol{a}$. The main idea here is that the quotient polynomial $Q(\tau) = \frac{V(\tau)-V(\omega^i)}{\tau-\omega^i}$ can be rewritten as:

$$\sum_{j=0}^{n} \frac{\mathcal{L}_j(\tau)v_j - v_i}{\tau - \omega^i} = \sum_{j=0,j\neq i}^{n} \frac{\mathcal{L}_j(\tau)v_j}{\tau - \omega^i} + \frac{\mathcal{L}_i(\tau)v_i - v_i}{\tau - \omega^i} = \sum_{j=0,j\neq i}^{n} v_j \frac{\mathcal{L}_j(\tau)}{\tau - \omega^i} + v_i \frac{\mathcal{L}_i(\tau) - 1}{\tau - \omega^i}$$

Note that the right hand side of the expression can be computed in the exponent by using $\boldsymbol{u}$. Furthermore using a_i and a_j from $\boldsymbol{a}$, they show how to compute $g^{\frac{\mathcal{L}_j(\tau)}{\tau-\omega^i}}$.

Our approach is simpler and faster due to the avoidance of computing Lagrange basis polynomials at commit, opening, as well as updates. Moreover, due to the nature of explicit sparse matrices, the operations are highly parallelizable. We highlight that [TAB+20] needs an extra vector of group elements (denoted $\boldsymbol{a}$) to compute their openings and updates to the openings. Our construction does not need this, since we use a different approach of using n points (including the point at index i) to compute an opening for i. Moreover, our characterization in Theorem 1 shows a more elegant and parallelizable technique to compute all openings in $\mathcal{O}(n \log n)$ time. We provide a summary of comparisons in Table 3.

4.5 Applications of efficient computation of all openings

We present some concrete applications where all openings are required to be computed efficiently. More details are available in Appendix B.

Data Availability Sampling (DAS): Light clients in blockchain networks use DAS to verify data availability without storing full blocks. Ethereum's proposed DAS scheme employs KZG commitments [Res]. Integrating our algorithm enhances the efficiency of encoding and opening computations, making DAS more practical for light clients.

Efficient Proofs in SNARKs and Decentralized Storage: Protocols like Caulk [ZBK+22] and proof-of-replication schemes [ABC+23] require multiple openings of KZG commitments. Our algorithm accelerates the precomputation of these proofs, enhancing efficiency in auditing and verification phases for SNARK-based systems and decentralized storage. Similar techniques are also used in Baloo [ZGK$^+$22] and cq [EFG22]. The precomputation also finds applications in Protostar [BC23], SublonK [CGG$^+$24], improved lookup arguments [CFF$^+$24, DGP$^+$24], cqlin [EG23], zero-knowledge location privacy [EZC$^+$24], batching-efficient RAM [DGP$^+$24] etc.

Laconic Oblivious Transfer (OT): In laconic OT, receivers compress their choice bits into a digest using KZG commitments [FHAS24]. Our algorithm improves the efficiency of computing all necessary openings, reducing the computational burden on receivers when handling large databases.

Non-Interactive Aggregatable Lotteries: Schemes like Jackpot [FHASW23] involve participants computing proofs of openings to verify lottery outcomes. Our efficient computation enables participants to precompute these proofs effectively, enhancing the performance and scalability of the lottery system.

4.6 Implementation and Benchmarks

In this section we describe the implementation and evalutation of the different KZG commitment schemes.

Hardware. All benchmarks were performed on a MacBook Pro with Apple M3 Max chip, with 16 cores and 64 GB RAM.

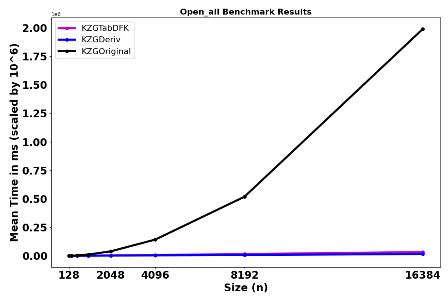
Code. All code is implemented in Rust, using the Arkworks [ac22] library. The criterion-rs crate was used for all benchmarks.

Methodology. We implemented the constructions of [FK23], [TAB$^+$20] and the original KZG construction [KZG10] and compare the run times of setup, committing, opening one position, opening all positions and updating commitments as well as updating a single proof of opening. We varied the size of the vector from 16 to 8192 and measured the time taken for each operation.

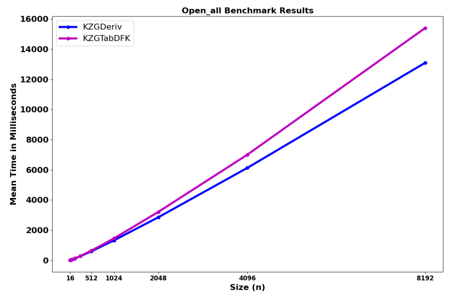
Setup: To setup, the work of [FK23] is the only one that matches the original KZG algorithm since they only need the powers of τ in setup. Our Setup algorithm is faster than that of [TAB$^+$20] by about 60% when we don't precompute $\mathsf{ColEDiv}$, and about 70% slower when do the precomputation. This is attributed to the fact that they need to compute extra vectors of group elements. See Figure 3.

Commit: Since the algorithm to compute commitment is the same in all the four constructions, the time taken to compute a commitment is exactly the same.

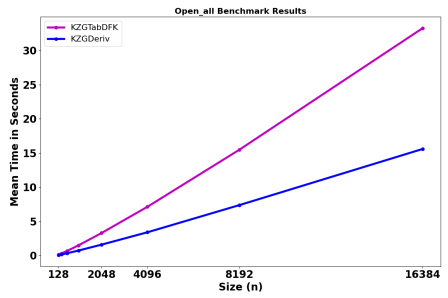
Open at index i: Our algorithms currently match the run times of the original KZG and [FK23] algorithms and are about 30× faster than that of [TAB$^+$20] and is about 7% faster than [FK23] and original KZG [KZG10]. This is primarily because [TAB$^+$20] make efforts to enable proofs of batch openings at once. Their algorithm for a single opening is therefore slower since it requires n field operations and n exponentiations.



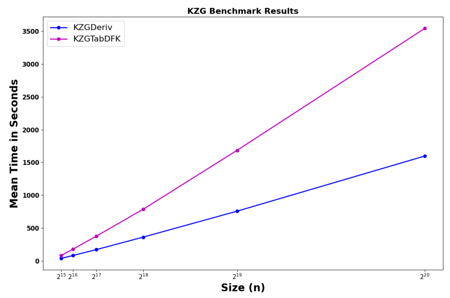
(a) Comparing run times to compute proofs of openings at all positions. Here KZGDeriv represents our implementation and overlaps with the algorithm of [TAB+20] and [FK23]



(b) Comparing run times to compute proofs of openings at all positions for smaller values $n < 2^{13}$.



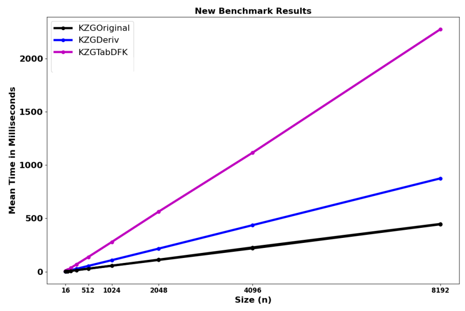
(c) Comparing run times to compute proofs of openings at all positions with precomputation of ColEDiv.



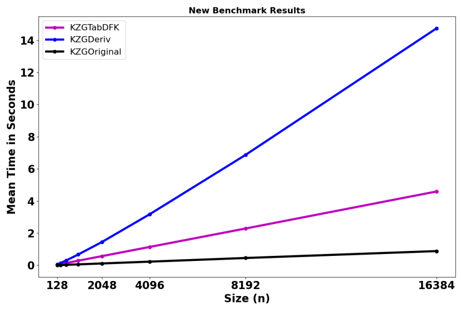
(d) Comparing run times to compute proofs of openings at all positions for larger sizes of $n \in [2^{15}, 2^{20}]$.

Fig. 2: Comparison of run times for computing proofs of openings at all positions.

Open all indices: The naive way of opening all indices would be to compute the opening proof for each index. This will take $\mathcal{O}(n^2)$ time. As mentioned earlier through FFT transforms both [FK23] and [TAB+20] show how to compute all proofs in $\mathcal{O}(n \log n)$ time. For $n = 2^{14}$, their algorithms are $60\times$ faster than the naive algorithm. Asymptotically our constructions also achieve $\mathcal{O}(n \log n)$ computation time, but since we can compute the openings by multiplying sparse matrices we can achieve better concrete numbers. See Figure 2a for a comparison with the naive opening strategy. Our algorithms are about $2.13\times$ faster for $n = 2^{14}$ and about $2.22\times$ faster for $n = 2^{20}$ than that of the approaches by [FK23] and [TAB+20] (See Figures 2b , 2c and 2d; in the latter two figures we use an optimized approach where the ColEDiv matrix is pre-computed in the setup phase). We estimate that as n grows larger and larger our algorithm will perform better than that of [FK23].



(a) With no optimizations for open-all



(b) Optimized version: ColEDiv is precomputed

Fig. 3: Comparing run times to do setup of public parameters. Note that since we precompute ColEDiv in the setup, it is slower than previous work.

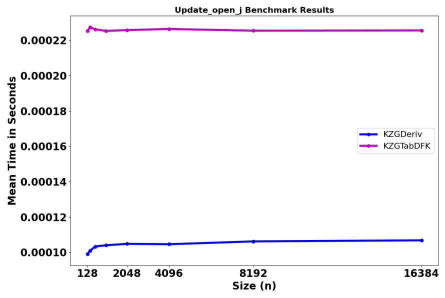


Fig. 4: Update Opening at index $i \neq j$

Updating commitments and proofs: Since updating a commitment is the same operation across all algorithms there is no difference in running times. When considering updates to a proof of opening in the case $i = j$, (i.e. to update a proof π_i when v_i has been updated), the algorithms of [TAB$^+$20] and ours are exactly the same, but on the other hand, our algorithm is twice as fast as that of [TAB$^+$20] for the case when the opening of index j is updated when index i is updated. See Figure 4.

Verifying proofs of opening: Since the verification algorithm is the same pairing check across all algorithms, the computation time is also the same.

5 Polynomial Division in Groth16

In this section we will present the necessary background on the Groth16 [Gro16] scheme and Quadratic Arithmetic Programs. Then we explain our approach leveraging l'Hôpital rule and provide implementation and benchmarks.

5.1 Background

Quadratic Arithmetic Programs. Gennaro et al [GGPR13, PHGR16] presented a characterization of the complexity class NP called Quadratic Span Programs. They also defined Quadratic Arithmetic Programs, a similar notion for arithmetic circuits.

A QAP $\mathcal{Q}$ over the field $\mathbb{F}_{|}$ contains three sets of $m+1$ polynomials $\mathcal{U} = \{u_i(x)\}, \mathcal{V} = \{v_i(x)\}, \mathcal{W} = \{w_i(x)\}$ for $i \in [0, m]$ and a target polynomial $t(x)$.

This QAP defines a language of statements $(a_1, \ldots, a_l) \in F^l$ and witnesses $(a_{l+1}, \ldots, a_m) \in F^{m-l}$, such that with $a_0 = 1$:

$$\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) = \sum_{i=0}^{m} a_i w_i(X) + h(X)t(X),$$

for some degree $n-2$ quotient polynomial $h(X)$, where n is the degree of $t(X)$, $F = \mathbb{F}_p$, l is the number of field elements in the public statement, m is the number of total field elements in the public statement, private witness and wire values together and n is the total number of gates in the arithmetic circuit. These values constitute the public parameters pp.

Groth16 Overview. The Groth16 proof system is a zk-SNARK that enables succinct and efficient verification of computations. It transforms a given computation into polynomial form, with constraints encoded as an R1CS. Polynomial division plays a crucial role by ensuring that the witness polynomial is divisible by a structured divisor polynomial representing the circuit's constraints. This is required to guarantee that the prover's input satisfies the computation without revealing private data. We present an overview of the Groth16 [Gro16] protocol in Figure 5.

5.2 Our Approach

Rank-1 Constrained System (R1CS) [BCR+19] provides an alternate way to view QAPs, by way of three R1CS matrices $U^{n_g \times n_v}$, $V^{n_g \times n_v}$ and $W^{n_g \times n_v}$, where n_g is the number of gates and n_v is the number of variables. A vector $\boldsymbol{a}^{n_v}$ satisfies the circuit iff:

$$U\boldsymbol{a} \circ V\boldsymbol{a} = W\boldsymbol{a},$$

where $\circ$ is the Hadamard product. These matrices have entries in the field $\mathbb{F}_q$, where q is the order of the bilinear groups used for instantiating the the proof system. Without loss of generality after sufficient padding, assume that $n = n_g$ is a power of 2 that divides the order of $\mathbb{F}_q^*$, that is, $n \mid q-1$. Let ω be a primitive n-th root of unity in $\mathbb{F}_q$.

Let $t(X) = \prod_{i=0}^{n_g}(X - \omega^i) = X^{n_g} - 1$, where intuitively ω^i is the x-coordinate assigned to the i-th gate. We have the relations:

$$\forall i \in [0, n_g], j \in [0, n_v] : \begin{array}{r} u_j(\omega^i) = (U)_{ij} \\ v_j(\omega^i) = (V)_{ij} \\ w_j(\omega^i) = (W)_{ij} \end{array}$$

```
**Setup**$(QAP, pp)$:
1. Sample $\alpha, \beta, \gamma, \delta, \tau \leftarrow \mathbb{F}_p$
2. Compute Prover Key $pk_{zk}$:
    (a) Compute $[\alpha]_1,\ [\beta]_1,\ [\beta]_2,\ [\delta]_1,\ [\delta]_2$
    (b) Compute $\{[\zeta_i]_1 = [\frac{\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau)}{\delta}]_1\}_{i=l+1}^{m}$
    (c) Compute $\{[\theta_j]_1 = [\frac{\tau^j t(\tau)}{\delta}]_1\}_{j=0}^{n-2}$
    (d) Compute $\{[\psi_i]_1 = \sum_{j=0}^{n-1} u_{i,j}[\tau^j]_1\}_{i=0}^{m}$
    (e) Compute $\{[\varphi_i]_2 = \sum_{j=0}^{n-1} v_{i,j}[\tau^j]_2\}_{i=0}^{m}$
    (f) Output $pk_{zk} = ([\zeta_i]_1, [\theta_j]_1, [\psi_i]_1, [\varphi_i]_2)$
3. Compute Verifier Key $vk_{zk}$:
    (a) Compute $[\alpha]_1,\ [\beta]_2,\ [\gamma]_2,\ [\delta]_2$
    (b) Output $vk_{zk} = \{[\chi_i]_1 = [\frac{\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau)}{\gamma}]_1\}_{i=0}^{l}$

**Prove**$(pk_{zk}, QAP, \langle a_i \rangle_{i=0}^{m})$:
1. Sample $r, s \leftarrow \mathbb{F}_p$
2. Compute polynomial $h(X) = \frac{(\sum_{i=0}^{m} a_i u_i(X))\cdot(\sum_{i=0}^{m} a_i v_i(X)) - \sum_{i=0}^{m} a_i w_i(X)}{t(X)}$
3. Compute:
    (a) $[A]_1 = [\alpha]_1 + r[\delta]_1 + \sum_{i=0}^{m} a_i[\psi_i]_1$
    (b) $[B]_2 = [\beta]_2 + s[\delta]_2 + \sum_{i=0}^{m} a_i[\varphi_i]_2$
    (c) $[C]_1 = s[\alpha]_1 + r[\beta]_1 + rs[\delta]_1 + \sum_{i=l+1}^{m} a_i[\zeta_i]_1 + \sum_{j=0}^{n-2} h_j[\theta_j]_1$
4. Output $\pi = [A]_1, [B]_2, [C]_1$

**Verify**$(vk_{zk}, \langle a_i \rangle_{i=0}^{l}, \pi)$
1. Compute $[V]_1 = \sum_{i=0}^{l} a_i[\chi_i]_1$
2. Parse $\pi$ as $[A]_1, [B]_2, [C]_1$
3. Check: $[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + [C]_1 \cdot [\delta]_2 + [V]_1 \cdot [\gamma]_2$
```

Fig. 5: Overview of Groth16

Typically the circuit frontend, the public statement, and witnesses are processed to produce the vectors $U\boldsymbol{a}, V\boldsymbol{a}, W\boldsymbol{a}$. Let's denote the interpolated polynomial of these evaluation vectors over the points ω^i:

$$\begin{aligned} u(X) &= \textstyle\sum_{j=0}^{n_v} a_j u_j(X) \\ v(X) &= \textstyle\sum_{j=0}^{n_v} a_j v_j(X) \\ w(X) &= \textstyle\sum_{j=0}^{n_v} a_j w_j(X) \end{aligned}$$

The asymptotically most complex operation in the computation of a Groth16 proof is the computation of the polynomial quotient $h(X)$:

$$h(X) = \frac{f_{\boldsymbol{a}}(X)}{t(X)} = \frac{u(X)\cdot v(X) - w(X)}{t(X)}, \tag{4}$$

where $f_{\boldsymbol{a}}(X) = u(X)\cdot v(X) - w(X)$.

Note that both $t(\omega^i)$ and $f_{\boldsymbol{a}}(\omega^i)$ are 0 for all $i \in [0, n_g]$. For this reason we cannot directly evaluate $h(\omega^i)$ using the quotient equation (4). However we can use l'Hôpital's Rule (Theorem 2) and get

$$h(\omega^i) = \frac{f'_{\boldsymbol{a}}(\omega^i)}{t'(\omega^i)}$$

Now, we have:

$$\begin{aligned} f'_{\boldsymbol{a}}(X) &= \frac{d}{dX}\left[u(X)v(X) - w(X)\right] \\ &= u(X)v'(X) - u'(X)v(X) - w'(X) \end{aligned}$$

Therefore, we can write for all $i \in [0, n_g]$:

$$h(\omega^i) = \frac{u(\omega^i)v'(\omega^i) + u'(\omega^i)v(\omega^i) - w'(\omega^i)}{t'(\omega^i)}$$

Denoting $\boldsymbol{\eta} \in \mathbb{F}_q^{n_g}$, such that $(\boldsymbol{\eta})_i = h(\omega^i)$, we can write:

$$\begin{gathered} \boldsymbol{\eta} = U\boldsymbol{a} \circ V'\boldsymbol{a} + U'\boldsymbol{a} \circ V\boldsymbol{a} - W'\boldsymbol{a}, \text{ where:} \\ (U')_{ij} = \tfrac{u'_j(\omega^i)}{t'(\omega^i)}, (V')_{ij} = \tfrac{v'_j(\omega^i)}{t'(\omega^i)}, (W')_{ij} = \tfrac{w'_j(\omega^i)}{t'(\omega^i)} \end{gathered}$$

Denoting DFT^{-1} as the inverse Vandermonde matrix with powers of ω, we have $\mathsf{DFT}^{-1}\boldsymbol{\eta} = \boldsymbol{h}$, the coefficient vector of $h(X)$. Then we could preprocess the CRS as follows. Let $\boldsymbol{t} \in \mathbb{F}_q^{n_g}$ be such that $(\boldsymbol{t})_i = [\tau^i t(\tau)/\delta]_1$, and $\boldsymbol{\theta} = (\mathsf{DFT}^{-1})^\top \boldsymbol{t}$. Then we have:

$$\left[\frac{h(\tau)t(\tau)}{\delta}\right]_1 = \boldsymbol{h}^\top \boldsymbol{t} = \boldsymbol{\eta}^\top (\mathsf{DFT}^{-1})^\top \boldsymbol{t} = \boldsymbol{\eta}^\top \boldsymbol{\theta}$$

This gives us a blueprint for an algorithm: We publish $\boldsymbol{\theta}$ in the CRS, instead of $\boldsymbol{t}$ and also publish matrices U', V', W' in addition to the R1CS matrices U, V, W. Prover computes $\boldsymbol{\eta} = U\boldsymbol{a} \circ V'\boldsymbol{a} + U'\boldsymbol{a} \circ V\boldsymbol{a} - W'\boldsymbol{a}$ and then adds $\boldsymbol{\eta}^\top\boldsymbol{\theta}$ to the C component of the Groth16 proof, instead of computing $h(X)$ and then computing $\left[\frac{h(\tau)t(\tau)}{\delta}\right]_1$.

However, typically U, V, W are sparse matrices due to the typically bounded fan-in of practical circuit gates, whereas U', V', W' maybe dense matrices. So computing $U'\boldsymbol{a}, V'\boldsymbol{a}, W'\boldsymbol{a}$ might end up taking quadratic time. Hence we apply our familiar transform of first going to coefficient space, taking derivatives, and coming back to evaluation space for point-wise multiplications and divisions. We summarize this in Algorithm 1.

5.3 Comparison with SnarkJS and Arkworks

Popular implementations as found in for example Arkworks and SnarkJS [SNA] avoid the 0/0 form by computing the polynomials at evaluation points shifted by

Algorithm 1 Compute $\eta_i = (h/t)(\omega^i)$ for $i = 0, \ldots, n-1$ using l'Hôpital's Rule

```
Let the inputs be $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \leftarrow (U\boldsymbol{a}, V\boldsymbol{a}, W\boldsymbol{a})$
$\boldsymbol{u}' \leftarrow \mathsf{DFT} \cdot \mathsf{D} \cdot \mathsf{DFT}^{-1}\boldsymbol{u}$
$\boldsymbol{v}' \leftarrow \mathsf{DFT} \cdot \mathsf{D} \cdot \mathsf{DFT}^{-1}\boldsymbol{v}$
$\boldsymbol{w}' \leftarrow \mathsf{DFT} \cdot \mathsf{D} \cdot \mathsf{DFT}^{-1}\boldsymbol{w}$
$\boldsymbol{invt}' \leftarrow n^{-1} \cdot \mathbf{pow}(\omega)$        $\triangleright\ t(X) = X^n - 1$ so $t'(\omega^i) = n\omega^{-i}$
$\boldsymbol{\eta} \leftarrow (\boldsymbol{u} \circ \boldsymbol{v}' + \boldsymbol{u}' \circ \boldsymbol{v} - \boldsymbol{w}') \circ \boldsymbol{invt}'$
return $\boldsymbol{\eta}$.
```

Algorithm 2 Compute $\eta_i = (h/t)(\zeta\omega^i)$ for $i = 0, \ldots, n-1$ using coset FFT

```
Let the inputs be $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \leftarrow (U\boldsymbol{a}, V\boldsymbol{a}, W\boldsymbol{a})$
Let $\zeta$ be a primitive $2n$-th root of unity.
Let $\mathsf{S}$ be the $n \times n$ diagonal matrix with non-zero entries $\mathsf{S}_{i,i} = \zeta^i$
$\boldsymbol{u}^* \leftarrow \mathsf{DFT} \cdot \mathsf{S} \cdot \mathsf{DFT}^{-1}\boldsymbol{u}$
$\boldsymbol{v}^* \leftarrow \mathsf{DFT} \cdot \mathsf{S} \cdot \mathsf{DFT}^{-1}\boldsymbol{v}$
$\boldsymbol{w}^* \leftarrow \mathsf{DFT} \cdot \mathsf{S} \cdot \mathsf{DFT}^{-1}\boldsymbol{w}$
$\boldsymbol{invt}^* \leftarrow (\zeta^n - 1)^{-1} \cdot \mathbf{1}$
$\boldsymbol{\eta} \leftarrow (\boldsymbol{u}^* \circ \boldsymbol{v}^* - \boldsymbol{w}^*) \circ \boldsymbol{invt}^*$
return $\boldsymbol{\eta}$.
```

a $2n$-the root of unity ζ, that is, at points of the form $\zeta\omega^i$. This is summarized in Algorithm 2 below.

Like SnarkJS, our protocol also needs to perform 3 inverse FFTs and 3 FFTs to get $U'a, V'a, W'a$, but we are avoiding computations at the $2n$-th roots of unity. We just use n-th roots of unity. This means we can take n to be the highest number such that 2^n divides $p-1$ for the derivative approach. Therefore this approach can support upto 2^n gates. In contrast, the coset approach can only support upto 2^{n-1} gates. This gives us a more expansive choice for group orders. That is, we can support *twice* as many gates as SnarkJS, while instantiating with the same bilinear group.

5.4 Implementation and Benchmarks

The above algorithm has been implemented in a fork of the Groth16 implementation by Arkworks and has been compared with the existing implementation which uses coset FFT's to compute h. Our algorithm is slightly faster, about 2-3%. This comparison was done using the most recent release of the *ark-groth16* crate (version 0.4.0) on a demo circuit with 24320 constraints which proves knowledge of a pre-image of a Blake2b hash. Our implementation of the computation of h takes 1.06s to compute h compared to 1.09s using the baseline implementation. Since the number of FFT's and inverse FFT's is the same in the two implementation, the small difference in performance is due to the linear operations being a bit faster.

6 Inner Product Arguments

We apply our techniques to inner product arguments (IPA) based on univariate sumchecks.

IPA. Given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, an IPA enables a prover to convince a verifier that $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \mu$, where the verifier has access to only the commitment c_a and c_b of $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively.

Univariate Sum-check. A sumcheck protocol is an interactive protocol that enables a prover to convince a verifier that $\sum_{\boldsymbol{a} \in H^m} f(\boldsymbol{a}) = 0$, where f is a given polynomial in $\mathbb{F}[X_1, \ldots, X_m]$ of individual degree d and H is a subset of $\mathbb{F}$. The univariate analogue was developed in [BCR+19, CNR+22] that enables a prover to convince a verifier that $\sum_{a \in H} f(a) = 0$ for a given polynomial $f \in \mathbb{F}[X]$ of degree d and subset $H \subseteq \mathbb{F}$.

Very recently, Das et al [DCX+23] present a threshold signature scheme from a new and efficient IPA. This IPA is in turn based on the univariate sumcheck protocols of [BCR+19, CNR+22]. Specifically, the protocol uses the following observation: Let $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^n$. Let $a(X), b(X) \in \mathbb{F}[X]$ be the unique degree $\leq (n-1)$ polynomials, such that $a(\omega^i) = (\boldsymbol{a})_i$ and $b(\omega^i) = (\boldsymbol{b})_i$.

This implies

$$\mu = \langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_{i \in [n]} a(\omega^i) b(\omega^i)$$

The vanishing polynomial $Z(X)$ is defined as

$$Z(X) = \prod_{i \in [n]} (X - \omega^i) = X^n - 1$$

Now the sumcheck lemma of [BCR+19] says that we must have:

$$a(X)b(X) = q(X)Z(X) + Xr(X) + \mu/n, \tag{5}$$

where $Z(X) = X^n - 1$ and as above $\mu = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$, for some $q(X), r(X)$ which are polynomials of degree $n-2$. Also, denote:

$$p(X) = Xr(X) + \mu/n$$

In their IPA protocol the CRS is set as $[\mathbf{pow}(\tau)]_1$ and $[\mathbf{pow}(\tau)]_2$. Note here $[1]_1$ and $[1]_2$ are distinct generators of a symmetric bilinear group $\mathbb{G}$. Then IPA π for $\mu = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$ is finally output as

$$\pi = (\pi_1, \pi_2, \pi_3) = ([q(\tau)]_1, [r(\tau)]_1, [p(\tau)]_2)$$

and the verification is done using the following pairing checks:

$$e(c_a, c_b) = e(\pi_1, [Z(\tau)]_1) \cdot e(\pi_2, [\tau]_1) \cdot e([\mu]_1, [1/n]_1)$$

and

$$e(\pi_3, [1]_1) = e(\pi_2, [\tau]_2) \cdot e([\mu]_1, [1/n]_2)$$

Applying our polynomial division technique. Recall from Equation 5 that:

$$a(X)b(X) = q(X)Z(X) + Xr(X) + \mu/n$$

Our goal is to compute the polynomials q and r efficiently by doing division in the evaluation space without having to compute the polynomials explicitly. To this end, first observe that

$$a(\omega^i)b(\omega^i) = q(\omega^i)Z(\omega^i) + \omega^i r(\omega^i) + \mu/n = \omega^i r(\omega^i) + \mu/n,$$

since $Z(\omega^i) = (\omega^i)^n - 1 = 0$. Therefore:

$$r(\omega^i) = \omega^{-i}(a(\omega^i)b(\omega^i) - \mu/n)$$

Now:

$$q(X) = \frac{a(X)b(X) - Xr(X) - \mu/n}{Z(X)}$$

Now observe that RHS has a $0/0$ form at ω^i. Let the numerator $N(X) = a(X)b(X) - Xr(X) - \mu/n$. We again apply l'Hôpital rule to evaluate $q(X)$ at ω^i.

$$N'(X) = a'(X)b(X) + a(X)b'(X) - Xr'(X) - r(X)$$

$$Z'(X) = nX^{n-1}$$

Therefore,

$$q(\omega^i) = \frac{N'(\omega^i)}{Z'(\omega^i)} = \frac{a'(\omega^i)b(\omega^i) + a(\omega^i)b'(\omega^i) - \omega^i r'(\omega^i) - r(\omega^i)}{n\omega^{i(n-1)}}$$

$$= \frac{\omega^i}{n}\left(a'(\omega^i)b(\omega^i) + a(\omega^i)b'(\omega^i) - \omega^i r'(\omega^i) - r(\omega^i)\right)$$

Therefore we can compute the proof using evaluation vectors as:

$$\boldsymbol{p} = \boldsymbol{a} \circ \boldsymbol{b}, \qquad \boldsymbol{r} = \mathsf{pow}(\omega^{-1}) \circ (\boldsymbol{p} - \mu/n \cdot \mathbf{1})$$

$$\boldsymbol{q} = \frac{1}{n}\mathsf{pow}(\omega) \circ \left(\widehat{\mathsf{D}}\boldsymbol{a} \circ \boldsymbol{b} + \boldsymbol{a} \circ \widehat{\mathsf{D}}\boldsymbol{b} - \mathsf{pow}(\omega) \circ \widehat{\mathsf{D}}\boldsymbol{r} - \boldsymbol{r}\right)$$

The dominant computation above are the 6 DFTs in computing $\boldsymbol{q}$. The IPA protocol in [DCX$^+$23] computes the proof by essentially computing vector commitments of the above quantities, which as we have seen in the KZG section can be performed by MSMs with DFT transformed powers of tau.

We can also compute $\boldsymbol{q}$ (evaluations at $\zeta\omega^i$ for this version) with the coset strategy as follows:

$$\boldsymbol{q} = \frac{1}{\zeta^n - 1}\left(\widehat{\mathsf{S}}\boldsymbol{a} \circ \widehat{\mathsf{S}}\boldsymbol{b} - \zeta \cdot \mathsf{pow}(\omega) \circ \widehat{\mathsf{S}}\boldsymbol{r} - \mu/n \cdot \mathbf{1}\right)$$

Recall that $\widehat{\mathsf{S}}$ is defined as the DFT conjugate of S, which is an $n \times n$ diagonal matrix with non-zero entries $\mathsf{S}_{i,i} = \zeta^i$, where ζ^i is the $2n$-th root of unity. Just like the SnarkJS implementation of Groth16, this also has the dominant cost of 6 FFTs, but uses a higher root of unity. In addition, this needs an additional $O(n)$ setup elements to account for the shifted basis of $\boldsymbol{q}$, with respect to $\boldsymbol{r}$. Concretely, this additional setup vector is $(\mathsf{S}^{-1})^\top(\mathsf{DFT}^{-1})^\top[\mathbf{pow}(\tau)]_1$.

Acknowledgment

The authors would like to thank Deepak Maram, Ben Riva, and Aayush Yadav for helpful feedback and pointer to references.

References

ABC$^+$23. Giuseppe Ateniese, Foteini Baldimtsi, Matteo Campanelli, Danilo Francati, and Ioanna Karantaidou. Advancing scalability in decentralized storage: A novel approach to proof-of-replication via polynomial evaluation. *Cryptology ePrint Archive*, 2023.

ac22. arkworks contributors. `arkworks` zksnark ecosystem, 2022.

BBHR18. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

BC23. Benedikt Bünz and Binyi Chen. Protostar: generic efficient accumulation/folding for special-sound protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 77–110. Springer, 2023.

BCR$^+$19. Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 103–128. Springer, Cham, May 2019.

Ber07. Daniel J. Bernstein. The tangent FFT. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lecture Notes in Computer Science 4851*, page 291–300, 2007.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM Press, May 1988.

BSBHR18. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

CFF$^+$24. Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: improvements, extensions and applications to zero-knowledge decision trees. In *IACR International Conference on Public-Key Cryptography*, pages 337–369. Springer, 2024.

CGG$^+$24. Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. *Proceedings on Privacy Enhancing Technologies*, 2024.

CHM$^+$20. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, Cham, May 2020.

CNR$^+$22. Matteo Campanelli, Anca Nitulescu, Carla Ràfols, Alexandros Zacharakis, and Arantxa Zapico. Linear-map vector commitments and their practical applications. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part IV*, volume 13794 of *Lecture Notes in Computer Science*, pages 189–219. Springer, Cham, December 2022.

Con. Keith Conrad. The different ideal. Expository papers/Lecture notes. Available at: `https://kconrad.math.uconn.edu/blurbs/gradnumthy/different.pdf`, year=2009, publisher=Citeseer.

CT65. James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

DCX$^+$23. Sourav Das, Philippe Camacho, Zhuolun Xiang, Javier Nieto, Benedikt Bünz, and Ling Ren. Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 356–370. ACM Press, November 2023.

DFGK14. George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 532–550. Springer, Berlin, Heidelberg, December 2014.

DGP$^+$24. Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. Batching-efficient ram using updatable lookup arguments. *Cryptology ePrint Archive*, 2024.

EFG22. Liam Eagen, Dario Fiore, and Ariel Gabizon. cq: Cached quotients for fast lookups. *Cryptology ePrint Archive*, 2022.

EG23. Liam Eagen and Ariel Gabizon. cqlin: Efficient linear operations on kzg commitments with cached quotients. *Cryptology ePrint Archive*, 2023.

EHK$^+$13. Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge Villar. An algebraic framework for Diffie-Hellman assumptions. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 129–147. Springer, Berlin, Heidelberg, August 2013.

EZC$^+$24. Jens Ernstberger, Chengru Zhang, Luca Ciprian, Philipp Jovanovic, and Sebastian Steinhorst. Zero-knowledge location privacy via accurate floating point snarks. *arXiv preprint arXiv:2404.14983*, 2024.

FHAS24. Nils Fleischhacker, Mathias Hall-Andersen, and Mark Simkin. Extractable witness encryption for kzg commitments and efficient laconic ot. *Cryptology ePrint Archive*, 2024.

FHASW23. Nils Fleischhacker, Mathias Hall-Andersen, Mark Simkin, and Benedikt Wagner. Jackpot: Non-interactive aggregatable lotteries. *Cryptology ePrint Archive*, 2023.

FK23. Dankrad Feist and Dmitry Khovratovich. Fast amortized kzg proofs. *Cryptology ePrint Archive*, 2023.

For65. G. D. Jr. Forney. On Decoding BCH Codes. *IEEE Trans. Inf. Theor.*, IT-11:549–557, 1965.

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, Berlin, Heidelberg, May 2013.

GMNO18. Rosario Gennaro, Michele Minelli, Anca Nitulescu, and Michele Orrù. Lattice-based zk-SNARKs from square span programs. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 556–573. ACM Press, October 2018.

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, Berlin, Heidelberg, May 2016.

GWC19. Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

HASW23. Mathias Hall-Andersen, Mark Simkin, and Benedikt Wagner. Foundations of data availability sampling. *Cryptology ePrint Archive*, 2023.

Jou00. Antoine Joux. A one round protocol for tripartite diffie–hellman. In *International algorithmic number theory symposium*, pages 385–393. Springer, 2000.

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, Berlin, Heidelberg, December 2010.

Lip13. Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part I*, volume 8269 of *Lecture Notes in Computer Science*, pages 41–60. Springer, Berlin, Heidelberg, December 2013.

MBKM19. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 2111–2128. ACM Press, November 2019.

MGW87. Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229. ACM New York, 1987.

MVO91. Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field. In *Proceedings of the* *twenty-third annual ACM symposium on Theory of computing*, pages 80–89, 1991.

PHGR16. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2):103–112, 2016.

Res. Ethereum Research. Data availability sampling. https://notes.ethereum.org/ReasmW86SuKqC2FaX83T1g. Accessed: 2024-08-05.

SCP$^+$22. Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3001–3018, 2022.

Set20. Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, Cham, August 2020.

Sha79. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

SNA. SNARKJS. https://geometry.xyz/notebook/the-hidden-little-secret-in-snarkjs.

TAB$^+$20. Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.

WB83. Lloyd R Welch and Elwyn R Berlekamp. Error correction for algebraic block codes, 1983.

ZBK$^+$22. Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3121–3134, 2022.

ZGK$^+$22. Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Rafols. Baloo: nearly optimal lookup arguments. *Cryptology ePrint Archive*, 2022.

A l'Hôpital's Rule for polynomials over arbitrary fields

Recall that l'Hôpital's Rule, named after the French mathematician Guillaume de l'Hôpital (1661-1704), states that given $c \in \mathbb{R}$ and functions $f, g : \mathbb{R} \to \mathbb{R}$ which are differentiable on a open interval around c but not necessarily in c, we have

$$\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$$

if $\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0$. As stated here, this is only valid for real functions, but it is also true over arbitrary fields if we restrict f and g to be polynomials. Throughout the paper, we let $\mathbb{F}$ denote an arbitrary field and let $\mathbb{F}[x]$ denote the polynomial ring over $\mathbb{F}$. We define the formal derivative as follows.

Definition 1. *Let $f \in \mathbb{F}[x]$. If we write $f(x) = \sum_{i=0}^{n} a_i x^i$ for $a_0, \dots, a_n \in \mathbb{F}$, we define the* derivative of f *as*

$$f'(x) = \sum_{i=0}^{n-1} a_{i+1}(i+1)x^i \in \mathbb{F}[x].$$

Now, l'Hôpital's Rule for polynomials over arbitrary fields can be stated as follows:

Theorem 2. *Let $f, g, h \in \mathbb{F}[x]$ such that $f(x) = g(x)h(x)$. Let $\alpha \in \mathbb{F}$ and assume that $f(\alpha) = g(\alpha) = 0$. Then*

$$f'(\alpha) = g'(\alpha)h(\alpha).$$

To prove this, we will first need a few basic results.

Lemma 1. *Let $f \in \mathbb{F}[x]$ and assume $f(\alpha) = 0$ for some $\alpha \in \mathbb{F}$. Then there is a unique polynomial $f_\alpha \in \mathbb{F}[x]$ such that $f(x) = f_\alpha(x)(x - \alpha)$ for all $x \in \mathbb{F}$.*

Proof. Since $\mathbb{F}$ is a field, $\mathbb{F}[x]$ is a Euclidean domain, so there are $q, r \in \mathbb{F}[x]$ with $\deg(r) < \deg(x - \alpha) = 1$ such that

$$f(x) = q(x)(x - \alpha) + r(x). \tag{6}$$

Now, $\deg(r) = 0$ so it is constant, and setting $x = \alpha$ in (6) implies that $r(x) = 0$. Letting $f_\alpha = q$ concludes the proof.

Lemma 2. *Let $f \in \mathbb{F}[x]$ and assume that $f(0) = 0$. Then $f'(0) = f_0(0)$.*

Proof. If f is constant, the statement is true, so we may assume that f has positive degree. Since $f(0) = 0$, the constant term of f is zero, so

$$f(x) = a_1 x + \cdots + a_n x^n$$

for some coefficients $a_1, \dots, a_n \in \mathbb{F}$. Using the definition of the derivative we get that $f'(0) = a_1$. On the other hand, we see that $f_0(x)$ as defined in Lemma 1 is

$$f_0(x) = a_1 + \cdots + a_n x^{n-1},$$

so $f_0(0) = a_1 = f'(0)$ as desired.

Corollary 1. *Let $f \in \mathbb{F}[x]$ and let $\alpha \in \mathbb{F}$ be given such that $f(\alpha) = 0$. Then $f'(\alpha) = f_\alpha(\alpha)$.*

Proof. Define $g(x) = f(x + \alpha)$. Now, $g(0) = f(\alpha) = 0$ and from Lemma 2 we get that $g'(0) = g_0(0)$. However, $f_\alpha(\alpha) = g_0(0)$ and $f'(\alpha) = g'(0)$ by the definition of g, so we get

$$f_\alpha(\alpha) = g_0(0) = g'(0) = f'(\alpha)$$

which finishes the proof.

We are now ready prove the main theorem.

Proof (Proof of Theorem 2). Let f, g and h be given as in the theorem. Since α is a root for both f and g we get from Lemma 1 that

$$f_\alpha(x)(x - \alpha) = g_\alpha(x)(x - \alpha)h(x)$$

for all $x \in \mathbb{F}[x]$. Since $\mathbb{F}[x]$ is an integral domain, this implies that

$$f_\alpha(x) = g_\alpha(x)h(x),$$

and applying Corollary 1 with both f and g we that

$$f'(\alpha) = g'(\alpha)h(\alpha)$$

as desired.

B Applications of efficient computation of all openings

Data Availability Sampling: In blockchain networks, participants can join as full nodes or light clients. Full nodes store and verify all block data and headers, while light clients only store block headers and rely on full nodes for data verification through fraud proofs. However, fraud proofs only help detect invalid data, not unavailable data. Data Availability Sampling (DAS) schemes, formalized by Hall-Anderson et. al. [HASW23], allow a block proposer to encode block content into a commitment and codeword. The light clients can then verify data availability by sampling parts of the codeword, ensuring the entire data is available if a sufficient number of light clients successfully probe it. The encoding of this data entails computing all openings of the commitment scheme. Ethereum has proposed to use the KZG commitment scheme for their DAS construction [Res]. Using our scheme in conjunction with their DAS scheme will improve the efficiency of the encoding function.
A related application is that of proof-serving nodes (PSNs), as described in [SCP+22]. These nodes assist light clients by maintaining proofs of openings for a commitment, which represents the state of a cryptocurrency. Any update to the state reflects a change in the commitment, necessitating the update of the proof of opening for all users. This process can impose a computational overhead on light clients, as they need to update their openings with every change to the commitment. PSNs alleviate this burden by updating each proof with every state change. This incurs a computational cost of $\mathcal{O}(n)$ for each state change. Using our scheme, however, PSNs can delay updating proofs until after a set of changes, and then update all proofs in $\mathcal{O}(n \log n)$ time, which may be more efficient depending on the frequency of required proof updates.
A recent work by Ateniese et al [ABC+23] aim to improve the scalability of decentralized storage by presenting efficient proof-of-replication protocols. In their construction the prover is required to prove openings of vector commitment

during the auditing phase. Using our scheme the prover can precompute all proofs, and provide the corresponding proof accordingly.

Improving run time of Lookup arguments for SNARKs: Lookup arguments such as Caulk [ZBK+22] present a scheme to prove membership of a subset within a public set in zero-knowledge. The main idea here is to represent the set as KZG commitment, and then to prove knowledge of openings efficiently. To prove a subset (that is multiple openings), the prover can precompute all openings and thereafter batch the openings to compute a a constant sized proof for the entire subset. Using our algorithm, we can improve the efficiency of this pre-computation of all proofs. Similar techniques are also used in Baloo [ZGK+22] and cq [EFG22]. The precomputation also finds applications in Protostar [BC23], SublonK [CGG+24], improved lookup arguments [CFF+24, DGP+24], cqlin [EG23], zero-knowledge location privacy [EZC+24], batching-efficient RAM [DGP+24] etc.

Laconic OT: In laconic oblivious transfer, the receiver holds a database $D \in 0, 1^n$ of n choice bits and publishes a digest $\mathsf{digest} \leftarrow H(D)$, whose size is independent of the size of D. The sender can then repeatedly choose a message pair (m_0, m_1), an index $i \in [n]$, and use the digest to compute a short message for the receiver, which allows them to obtain $m_{D[i]}$. The construction of Fleischhacker et al [FHAS24] uses the KZG commitment scheme to compute the digest. More specifically, receiver computes the digest (as a KZG commitment), and all openings, and sends the digest to the sender. The sender witness encrypts the messages using the digest, such that the receiver is able to decrypt using only the proof of opening at the corresponding index. Since the receiver computes all openings of the KZG commitment, it can be done efficiently using our scheme.

Non-interactive Aggregatable Lotteries: Fleischhacker et al [FHASW23] present Jackpot, which is a lottery scheme based on vector commitments. More specifically, they present a construction of a verifiable random function (VRF) using the KZG vector commitment. In their scheme, each party P_j initially commits to a random vector $v^{(j)} \in [k]^T$ to participate in T lotteries. In the i-th lottery round a per party challenge x_j is derived from a random seed and party P_j wins iff $v(j) = x_j$. Each party can prove that they won by revealing an opening for position i of their commitment. The authors note that the most time-critcal part for the parties is in the computation of the proofs. But all the openings can be computed immediately after key generation and before the lotteries. Using our scheme the efficiency of this computation can be improved.

C Proofs of Equations

Theorem 1(i). (Restated) Let field F contain a primitive n-th root of unity ω, Let D be the derivative operator from Table 1. The derivative conjugate matrix

$\widehat{\mathsf{D}}$ has the following explicit structure:

$$(\widehat{\mathsf{D}})_{ij} = \begin{cases} \frac{\omega^{j-i}}{\omega^i - \omega^j}, \text{ for } i \neq j \\ \frac{(n-1)}{2\omega^i}, \text{ for } i = j \end{cases}$$

Proof. Recall $\widehat{\mathsf{D}} = \mathsf{DFT} \cdot \mathsf{D} \cdot \mathsf{DFT}^{-1}$, where D is the off-diagonal derivative matrix $(\mathsf{D})_{i,i-1} = i$ and $\mathsf{DFT}_{ij} = \omega^{ij}$. Now we have,
1. $(\mathsf{D})_{i,i+1} = i + 1$ and 0 elsewhere.
2. $\mathsf{DFT}_{ij} = \omega^{ij}$.
3. $\mathsf{DFT}^{-1}_{ij} = \frac{1}{n}\omega^{-ij}$.

To start with, let's compute

$$\mathsf{E}' = \mathsf{DFT} \cdot \mathsf{D}, \quad \mathsf{E}'_{ij} = \sum_{k=0}^{n-1} \mathsf{DFT}_{ik} \cdot \mathsf{D}_{kj} = \mathsf{DFT}_{i(j-1)}\mathsf{D}_{(j-1)j} \; = \; \omega^{i(j-1)} \cdot (j)$$

Since, $\widehat{\mathsf{D}} = \mathsf{E}' \cdot \mathsf{DFT}^{-1}$, we have:

$$\widehat{\mathsf{D}}_{ij} = \sum_{k=0}^{n-1} \mathsf{E}'_{ik}\mathsf{DFT}^{-1}{}_{kj} = \sum_{k=0}^{n-1} \omega^{i(k-1)} \cdot (k) \cdot \frac{1}{n}\omega^{-kj} = \frac{1}{n}\omega^{-i} \sum_{k=0}^{n-1} (k) \cdot \omega^{k(i-j)}$$

Let $\omega^{i-j} = a$, then using geometric series and its derivative, for $a \neq 1$, i.e. $i \neq j$, we have

$$\widehat{\mathsf{D}}_{ij} = \frac{1}{n}\omega^{-i} \frac{(n-1)a^{n+1} - na^n + a}{(a-1)^2}$$

Since $a^n = \omega^{(i-j)n} = 1$, the above is same as:

$$\frac{1}{n}\omega^{-i} \frac{(n-1)a^1 - n + a}{(a-1)^2} = \frac{1}{n}\omega^{-i} \frac{(na - n)}{(a-1)^2} \; = \; \omega^{-i} \frac{1}{(a-1)}$$

Substituting $a = \omega^{i-j}$, the above becomes:

$$\frac{\omega^{j-i}}{\omega^i - \omega^j}$$

In the case that $i = j$, we have:

$$\widehat{\mathsf{D}}_{ii} = \frac{1}{n}\omega^{-i} \sum_{k=0}^{n-1} (k) \cdot \omega^{k(i-i)} = \frac{1}{n}\omega^{-i} \frac{n(n-1)}{2} \; = \; \frac{(n-1)\omega^{-i}}{2}$$

Thus,

$$(\widehat{\mathsf{D}})_{ij} = \begin{cases} \frac{\omega^{j-i}}{\omega^i - \omega^j}, \text{ for } i \neq j \\ \frac{(n-1)}{2\omega^i}, \text{ for } i = j \end{cases}$$

Remark. Let

$$(\mathsf{D}')_{ij} = \begin{cases} \frac{\omega^{j-i}\cdot\omega^j}{\omega^i-\omega^j}, & \text{for } i \neq j \\ \frac{(n-1)}{2}, & \text{for } i = j \end{cases}$$

and let D'' be the diagonal matrix with entries ω^{-j}, so that $\widehat{\mathsf{D}} = \mathsf{D}' \cdot \mathsf{D}''$. It is not difficult to see that D' is a multiplication matrix of the polynomial $d(X) = (n-1)/2 + \sum_{i=1}^{n-1} \frac{X^i}{\omega^{2i}-\omega^i}$ (set $j = 0$ in the above definition of D'). Hence, by Lemma 3, $\mathsf{DFT} \cdot \mathsf{D}' \cdot \mathsf{DFT}^{-1}$ is a diagonal matrix. However, the current lemma shows that $\mathsf{DFT}^{-1} \cdot \mathsf{D}' \cdot \mathsf{D}'' \cdot \mathsf{DFT}$ is a shifted-diagonal non-full ranked matrix[6], which is a surprising result (note, the similarity transform is with DFT^{-1} instead of DFT). While relationships between differential operators and Fourier transforms are well known for functions over complete fields (such as complex numbers), to the best of our knowledge the above characterization is new for finite extensions of Q and finite fields.

Lemma 3. *For any $F(X)$, and its corresponding vandermonde matrix over Z_q, for any $f(X) \in R_q = Z_q[X]/(F(X))$, $\mathsf{V}\mathsf{M}_f\mathsf{V}^{-1} = \mathsf{diag}_f$, where diag_f is the diagonal matrix with entries $f(w_i)$ ($i \in [0..n-1]$).*

Theorem 3. *Given the the following matrix J:*

$$\mathsf{J} = \begin{cases} \frac{1}{\omega^i-\omega^j} & , i \neq j \\ \frac{n-1}{2}\omega^{-i} & , i = j \end{cases}$$

we have that the conjugate matrix $\widehat{\mathsf{J}}$ is a sparse matrix of the following explicit form:

$$\widehat{\mathsf{J}} = \begin{cases} n-i & , j = i-1 \text{ and } i \in [1, n-1] \\ 0 & , \text{otherwise} \end{cases}$$

Proof. Recall that $\widehat{\mathsf{J}} = \mathsf{DFT} \cdot \mathsf{J} \cdot \mathsf{DFT}^{-1}$. Alternatively,

$$\mathsf{J} = \mathsf{DFT}^{-1} \cdot \widehat{\mathsf{J}} \cdot \mathsf{DFT}$$

Let's start with

$$\mathsf{E}' = \mathsf{DFT}^{-1} \cdot \widehat{\mathsf{J}}, \quad \mathsf{E}'_{ij} = \sum_{k=0}^{n-1} \mathsf{DFT}^{-1}{}_{ik} \cdot \widehat{\mathsf{J}}_{kj} = \mathsf{DFT}^{-1}{}_{i(j+1)} \cdot \widehat{\mathsf{J}}_{(j+1)j} = \frac{1}{n}\omega^{-i(j+1)} \cdot (n-j-1)$$

Since $\mathsf{J} = \mathsf{E}' \cdot \mathsf{DFT}$, we have:

$$\mathsf{J}_{ij} = \sum_{k=0}^{n-1} \mathsf{E}'_{ik} \cdot \mathsf{DFT}_{kj} = \sum_{k=0}^{n-1} \frac{n-k-1}{n}\omega^{-i(k+1)} \cdot \omega^{kj} = \frac{n-1}{n}\omega^{-i}\sum_{k=0}^{n-1}\omega^{(j-i)k} - \frac{\omega^{-i}}{n}\sum_{k=0}^{n-1} k\omega^{(j-i)k}$$

Note that $\sum_{k=0}^{n-1} \omega^{(j-i)k} = 0$, thus we have

[6] Since D is singular, it must be the case that D' is singular; indeed it can be checked that the polynomial $d(X)$ has $X = 1$ as a root, which is also a root of $X^n - 1$.

$$\mathsf{J}_{ij} = -\frac{\omega^{-i}}{n}\sum_{k=0}^{n-1} k\omega^{(j-i)k}$$

Let $\omega^{j-i} = a$, and using the geometric series and its derivative as above we have:

$$\mathsf{J}_{ij} = -\frac{\omega^{-i}}{n}\sum_{k=0}^{n-1} ka^k = -\frac{\omega^{-i}}{n}\frac{(n-1)a^{n+1} - na^n + a}{(a-1)^2}$$

Since $a^n = \omega^{(j-i)n} = 1$, the above is same as:

$$-\frac{1}{n}\omega^{-i}\frac{(n-1)a^1 - n + a}{(a-1)^2} = -\frac{1}{n}\omega^{-i}\frac{(na-n)}{(a-1)^2} = -\omega^{-i}\frac{1}{(a-1)}$$

Substituting $a = \omega^{j-i}$, the above becomes:

$$\mathsf{J}_{ij} = -\frac{1}{\omega^j - \omega^i} = \frac{1}{\omega^i - \omega^j}$$

Moreover, when $i = j$, we have

$$\mathsf{J}_{ij} = \frac{n-1}{n}\omega^{-i}\sum_{k=0}^{n-1}\omega^{(i-i)k} - \frac{\omega^{-i}}{n}\sum_{k=0}^{n-1} k\omega^{(i-i)k} = \omega^{-i}(n-1) - \omega^{-i}\frac{n-1}{2} = \omega^{-i}\frac{n-1}{2}$$

$$\mathsf{J} = \begin{cases} \frac{1}{\omega^i - \omega^j} & , i \neq j \\ \frac{n-1}{2}\omega^{-i} & , i = j \end{cases}$$

Theorem 4. *Given the following matrix* $\mathsf{ColEDiv}$*:*

$$\mathsf{ColEDiv} = \begin{cases} -\frac{1}{\omega^j - \omega^i} & , i \neq j \\ 0 & , i = j \end{cases}$$

we have the conjugate matrix $\widehat{\mathsf{ColEDiv}}$ *is a sparse matrix with the following explicit form:*

$$(\widehat{\mathsf{ColEDiv}})_{i,j} = \begin{cases} -\frac{n-1}{2} & , (i,j) = (0, n-1) \\ \frac{n+1}{2} - i & , j = i-1 \textit{ and } i \in [1, n-1] \\ 0 & , \textit{otherwise} \end{cases} \tag{7}$$

Proof. Recall that

$$\widehat{\mathsf{ColEDiv}}) = \mathsf{DFT} \cdot \mathsf{ColEDiv} \cdot \mathsf{DFT}^{-1}$$

To prove the theorem that the conjugate matrix $\widehat{\mathsf{ColEDiv}} = \mathsf{DFT} \cdot \mathsf{ColEDiv} \cdot \mathsf{DFT}^{-1}$ has the specified explicit form, we will break down the multiplication step by step. We will compute $E' = \mathsf{DFT} \cdot \mathsf{ColEDiv}$ and then compute $\widehat{\mathsf{ColEDiv}} = E' \cdot \mathsf{DFT}^{-1}$.

Lets start with computing $E' = \mathsf{DFT} \cdot \mathsf{ColEDiv}$

The element $(E')_{k,j}$ is given by:

$$(E')_{k,j} = \sum_{i=0}^{n-1} (\mathsf{DFT})_{k,i} \cdot (\mathsf{ColEDiv})_{i,j}.$$

Since $(\mathsf{ColEDiv})_{i,j} = 0$ when $i = j$, we have:

$$(E')_{k,j} = -\sum_{\substack{i=0\\ i\neq j}}^{n-1} \omega^{-ki} \cdot \frac{1}{\omega^j - \omega^i} = -\sum_{\substack{i=0\\ i\neq j}}^{n-1} \frac{\omega^{-ki}}{\omega^j - \omega^i}.$$

Next we compute $\widehat{\mathsf{ColEDiv}} = E' \cdot \mathsf{DFT}^{-1}$

The element $(\widehat{\mathsf{ColEDiv}})_{k,\ell}$ is given by:

$$\begin{aligned}
(\widehat{\mathsf{ColEDiv}})_{k,\ell} &= \sum_{j=0}^{n-1} (E')_{k,j} \cdot (\mathsf{DFT}^{-1})_{j,\ell} \\
&= \frac{1}{n} \sum_{j=0}^{n-1} (E')_{k,j} \omega^{j\ell} \\
&= -\frac{1}{n} \sum_{j=0}^{n-1} \left(\sum_{\substack{i=0\\ i\neq j}}^{n-1} \frac{\omega^{-ki}}{\omega^j - \omega^i} \right) \omega^{j\ell} \\
&= -\frac{1}{n} \sum_{i=0}^{n-1} \omega^{-ki} \sum_{\substack{j=0\\ j\neq i}}^{n-1} \frac{\omega^{j\ell}}{\omega^j - \omega^i}.
\end{aligned}$$

Now we simplify the inner sum:

Observe that $\omega^j - \omega^i = \omega^i(\omega^{j-i} - 1)$, and $\omega^{j\ell} = \omega^{i\ell}\omega^{(j-i)\ell}$. Thus, the inner sum becomes:

$$\begin{aligned}
\sum_{\substack{j=0\\ j\neq i}}^{n-1} \frac{\omega^{j\ell}}{\omega^j - \omega^i} &= \omega^{i\ell} \sum_{\substack{d=1\\ d\neq 0}}^{n-1} \frac{\omega^{d\ell}}{\omega^i(\omega^d - 1)} \\
&= \omega^{i(\ell-1)} \sum_{d=1}^{n-1} \frac{\omega^{d\ell}}{\omega^d - 1}.
\end{aligned}$$

To compute the total sum, substitute back into the expression for $(\widehat{\mathsf{ColEDiv}})_{k,\ell}$:

$$\begin{aligned}(\widehat{\mathsf{ColEDiv}})_{k,\ell} &= -\frac{1}{n}\sum_{i=0}^{n-1}\omega^{-ki}\left(\omega^{i(\ell-1)}\sum_{d=1}^{n-1}\frac{\omega^{d\ell}}{\omega^d-1}\right)\\ &= -\frac{1}{n}\left(\sum_{i=0}^{n-1}\omega^{i(\ell-k-1)}\right)\left(\sum_{d=1}^{n-1}\frac{\omega^{d\ell}}{\omega^d-1}\right).\end{aligned}$$

The sum over i simplifies using the orthogonality of roots of unity:

$$\sum_{i=0}^{n-1}\omega^{i(\ell-k-1)} = \begin{cases} n, & \text{if } \ell \equiv k+1 \mod n,\\ 0, & \text{otherwise.}\end{cases}$$

Let

$$\sum_{i=0}^{n-1}\omega^{i(\ell-k-1)} = n\delta_{\ell,k+1},$$

where δ is the Kronecker delta function.

Thus, the total sum simplifies to:

$$(\widehat{\mathsf{ColEDiv}})_{k,\ell} = -\frac{1}{n}\cdot n\delta_{\ell,k+1}\left(\sum_{d=1}^{n-1}\frac{\omega^{d\ell}}{\omega^d-1}\right) = -\delta_{\ell,k+1}\sum_{d=1}^{n-1}\frac{\omega^{d\ell}}{\omega^d-1}.$$

We need to evaluate the sum:

$$S_\ell = \sum_{d=1}^{n-1}\frac{\omega^{d\ell}}{\omega^d-1}.$$

Case 1: $\ell = 0$

When $\ell = 0$, $\omega^{d\ell} = 1$, so:

$$S_0 = \sum_{d=1}^{n-1}\frac{1}{\omega^d-1}.$$

Since $\omega^d = e^{2\pi id/n}$, the terms are complex conjugates and sum to:

$$S_0 = \frac{n-1}{2}.$$

Case 2: $1 \le \ell \le n-1$

For $\ell \neq 0$, we can use the identity:

$$\sum_{d=1}^{n-1}\frac{\omega^{d\ell}}{\omega^d-1} = \frac{n+1}{2} - \ell.$$

Combining the results, we find that $\widehat{\mathsf{ColEDiv}}$ is a sparse matrix with entries:

$$(\widehat{\mathsf{ColEDiv}})_{k,\ell} = \begin{cases} -\frac{n-1}{2}, & \text{if } k=0,\ \ell=n-1, \\ \frac{n+1}{2}-k, & \text{if } \ell=k-1 \text{ and } 1\le k\le n-1, \\ 0, & \text{otherwise.} \end{cases}$$

D Toeplitz Matrices

Let $\boldsymbol{M}$ be the following n-by-n Toeplitz matrix:

$$\boldsymbol{M} = \begin{bmatrix} f_1 & f_2 & \cdots & f_n \\ f_2 & \iddots & f_n & 0 \\ \vdots & f_n & \ddots & \vdots \\ f_n & 0 & \cdots & 0 \end{bmatrix}$$

i.e. where $\boldsymbol{M}_{i,j} = f_{i+j+1}$ if $i+j<n$ and $\boldsymbol{M}_{i,j}=0$ otherwise.

It is well known that [Con, Theorem 3.7], [FK23]:

$$\mathsf{\textbf{pow}}(X)^\top \cdot \boldsymbol{M} \cdot \mathsf{\textbf{pow}}(Y) = \frac{f(X)-f(Y)}{X-Y}$$

Using this, and for X using τ and for Y using *powers of* ω, we get

$$\begin{aligned} \mathsf{\textbf{pow}}(\tau)^\top \cdot \boldsymbol{M} \cdot \mathsf{DFT} &= \left\langle \frac{f(\omega^i)-f(\tau)}{\omega^i-\tau} \right\rangle_{i=0}^{n-1} & (8) \\ &= \langle (\mathsf{CDiv}_{\omega^i}[f])\,(\tau)\rangle_{i=0}^{n-1} & (9) \\ &= \mathsf{\textbf{pow}}(\tau)^\top \cdot \langle \mathsf{CDiv}_{\omega^i}\rangle_{i=0}^{n-1} \cdot \boldsymbol{f} & (10) \\ &= \mathsf{\textbf{pow}}(\tau)^\top \cdot \mathsf{DFT}^{-1} \cdot \langle \mathsf{EDiv}_{\omega^i}\rangle_{i=0}^{n-1} \cdot \mathsf{DFT} \cdot \boldsymbol{f} & (11) \end{aligned}$$

Further, recall from (2) that the above in column form is same as (recalling $\mathsf{DFT}\cdot\boldsymbol{f} = \boldsymbol{v}$, $[\boldsymbol{w}]_1 = \mathsf{DFT}^{-1}\cdot[\mathsf{\textbf{pow}}(\tau)]_1$, and $\mathsf{DFT}^\top = \mathsf{DFT}$, being vandermonde matrix of roots of X^n-1)

$$[\boldsymbol{w}]_1 \circ \widehat{\mathsf{D}}\boldsymbol{v} + (\mathsf{ColEDiv}\cdot[\boldsymbol{w}]_1) \circ \boldsymbol{v} + \mathsf{DiaEDiv} \cdot ([\boldsymbol{w}]_1 \circ \boldsymbol{v}) \qquad (12)$$

The above also shows that $[\boldsymbol{w}]_1$ does not have to be DFT-inverse of powers of a τ, but can be an arbitrary vector of groups elements. As remarked earlier the conjugates of $\widehat{\mathsf{D}}$, $\mathsf{ColEDiv}$ and $\mathsf{DiaEDiv}$ are all sparse, and in fact, if $(\mathsf{ColEDiv}\cdot[\boldsymbol{w}]_1)$ is given pre-computed as a vector of group elements, then the above can be computed with just $3n+2n\log n$ scalar-multiplications[7] (the $2n\log n$ scalar-multiplications coming from computing the last term, since $\mathsf{DiaEDiv}$ is not sparse, but only its conjugate is sparse).

[7] also referred as group exponentiations in mutiplicative group notation.

Actually, if we use the remarks about Cooley-Tukey in Section 2, the total number of scalar multiplications is only $1/2 * n \log n$ (both for DFT and DFT^{-1}). Thus, the total number of scalar multiplications is $n \log n$. Note, that this cost is same whether the result is needed in evaluation basis or power basis.

We should also investigate if the **Feist-Khovratovich** [FK23] method described in Section 4.4 itself can be improved to get $n \log n$ scalar multiplications. Recall, they expand the matrix M to be a $2n \times 2n$ multiplication matrix M' modulo $X^{2n} - 1$. Thus, $M' \cdot \mathbf{pow}(\tau)$ can be computed by polynomial multiplication modulo $X^{2n} - 1$. So, M' being a multiplication matrix of say polynomial $\tilde{f}(X)$, and $\mathbf{pow}(\tau)$ extended to $2n$ elements by appending n zeroes, to be viewed as another polynomial $T(X)$, we need to compute $\tilde{f}(X) * T(X)$. The DFT of $T(X)$ can be provided in the CRS, and further the DFT of $\tilde{f}(X)$ can be computed as a field DFT (of size $2n$). At this point, one can do a Hadamard product of the two DFTs, which would require $2n$ scalar multiplications.

In other words, so far we have computed $\mathsf{DFT}^* \cdot \tilde{f} \circ \mathsf{DFT}^* \cdot T$, where DFT^* denotes discrete fourier transform w.r.t. $X^{2n} - 1$, i.e. using $2n$-th roots of unity. This is same as $\mathsf{DFT}^* \cdot (\tilde{f}(X) * T(X))$ which is same as $\mathsf{DFT}^* \cdot (M' \cdot [\mathbf{pow}(\tau) \mid \mathbf{0}])$. This may seem same as $\mathsf{DFT}^* \cdot (M \cdot \mathbf{pow}(\tau))$, but that is not true, as the first n columns of M' have extra elements in the bottom n rows, and these are contributing to the above. So, instead one needs to run DFT^*-inverse on the above which would take $2n/2 * \log n$ scalar multiplications. Then, one would get $(M' \cdot [\mathbf{pow}(\tau) \mid 0])$. Now, the first n components of this is same as $M \cdot \mathbf{pow}(\tau)$. One has to run a final DFT on this, to get the openings at the roots of unities. This would require another $n/2 * \log n$ scalar multiplications. So, the total cost is $3n/2 * \log n$ scalar multiplications.

Thus, while for our method the cost is the same $n \log n$ (elliptic curve) scalar multiplications whether we need the result in the evaluation basis or power basis, for the modified **Feist-Khovratovich** [FK23] method described above, the cost for computation in the evaluation basis is $3n/2 * \log n$ scalar multiplications (while the cost in the power basis is $n \log n$).

E Other systems

In this section, we briefly describe the applicability of our techniques to two other systems: STARK and PLONK. We defer detailed technical descriptions and benchmark evaluations to future work, while providing a high-level blueprint here.

E.1 STARK

A STARK [BSBHR18] prover generates the execution trace of the program on a given set of inputs and does the following[8]:

1. Interpolate the execution trace to obtain *trace* polynomials.

[8] https://aszepieniec.github.io/stark-anatomy/stark

2. Interpolate the boundary points to obtain the boundary interpolants, and compute the boundary zerofiers along the way.
3. Subtract the boundary interpolants from the trace polynomials, and divide out the boundary zerofier, giving rise to the boundary quotients.
4. Commit to the boundary quotients.
5. Get r random coefficients from the verifier.
6. Compress the r transition constraints into one master constraint that is the weighted sum.
7. Symbolically evaluate the master constraint in the trace polynomials, thus generating the transition polynomial.
8. Divide out the transition zerofier to get the transition quotient.
9. Commit to the transition zerofier.
10. Run FRI [BSBHR18] on all the committed polynomials: the boundary quotients, the transition quotients, and the transition zerofier.
11. Supply the Merkle leafs and authentication paths that are requested by the verifier.

We now use the observation that the transition polynomial evaluations and the zerofier evaluations are all 0 at each row in the trace. Therefore, we can use l'Hôpital's rule again: instead of computing $m(X)/z(X)$, we instead compute $m'(X)/z'(X)$. Just like as in Groth16, we can do much of this computation in the evaluation space, and avoid division in the coefficient space altogether.

1. We additionally describe the derivative transition polynomials $m'(X)$ in the circuit setup.
2. We can optimize the description of $z'(X)$ by having the evaluation domain be a suitable subgroup of roots of unity, with padding if necessary.
3. The prover evaluates the transition derivatives while generating the trace.
4. Compute the derivative of the trace using FFT and the D matrix, as in our Groth16 optimization.
5. Compute point-wise division in the derivative evaluation space
6. Finally, we use a DFT to migrate the division evaluations to the coefficient space.
7. Now we can use FRI as usual over this quotient polynomial.
8. Analogous optimizations can be done for the boundary quotient evaluation as well.

E.2 PLONK

PLONK [GWC19] has a general strategy similar to STARKs, but uses a different arithmetization. Instead of transition polynomials, PLONK uses *selector* polynomials to specify circuits. In addition, PLONK uses *prescribed permutation checks* to prove consistency of wire values between execution rows. The rough blueprint is similar now:

1. Specify the derivative of the selector and permutation check polynomials at circuit-based setup.
2. Precompute the derivative of the zerofier polynomial, again optimizing through careful selection of subgroups of roots of unity.

3. Compute the trace as usual.
4. Compute the derivative of the trace using FFT and the D matrix, as in our Groth16 optimization.
5. Compute point-wise division in the derivative evaluation space using the circuit polynomial derivatives and trace polynomial derivative.
6. If KZG is used for polynomial commitments, then we can use our optimizations in this paper to compute proofs and openings in the evaluation space itself.

F Example computation from EDiv_k

In this section, we will show how for $n = 4$ all KZG openings can be computed by stacking the different EDiv_k matrices as described in Section 4.3.

We will first present the different EDiv_k matrices for $k \in \{0, 1, 2, 3\}$. First of all note that each matrix is star-shaped with the center of the star being the pink-colored intersection of all lines (column, row, and diagonal)

$\frac{3}{2}$	$-\frac{\omega}{\omega-1}$	$-\frac{\omega^2}{\omega^2-1}$	$-\frac{\omega^3}{\omega^3-1}$
$-\frac{1}{\omega-1}$	$\frac{1}{\omega-1}$	0	0
$-\frac{1}{\omega^2-1}$	0	$\frac{1}{\omega^2-1}$	0
$-\frac{1}{\omega^3-1}$	0	0	$\frac{1}{\omega^3-1}$

(a) EDiv_0

$\frac{1}{1-\omega}$	$-\frac{1}{1-\omega}$	0	0
$-\frac{\omega^3}{1-\omega}$	$\frac{3}{2\omega}$	$-\frac{\omega}{\omega^2-\omega}$	$-\frac{\omega^2}{\omega^3-\omega}$
0	$-\frac{1}{\omega^2-\omega}$	$\frac{1}{\omega^2-\omega}$	0
0	$-\frac{1}{\omega^3-\omega}$	0	$\frac{1}{\omega^3-\omega}$

(b) EDiv_1

$\frac{1}{1-\omega^2}$	0	$-\frac{1}{1-\omega^2}$	0
0	$\frac{1}{\omega-\omega^2}$	$-\frac{1}{\omega-\omega^2}$	0
$-\frac{\omega^2}{1-\omega^2}$	$-\frac{\omega^3}{\omega-\omega^2}$	$\frac{3}{2}\omega^{-2}$	$-\frac{\omega}{\omega^3-\omega^2}$
0	0	$-\frac{1}{\omega^3-\omega^2}$	$\frac{1}{\omega^3-\omega^2}$

(c) EDiv_2

$\frac{1}{1-\omega^3}$	0	0	$-\frac{1}{1-\omega^3}$
0	$\frac{1}{-\omega^3+\omega}$	0	$-\frac{1}{-\omega^3+\omega}$
0	0	$\frac{1}{-\omega^3+\omega^2}$	$-\frac{1}{-\omega^3+\omega^2}$
$-\frac{\omega}{1-\omega^3}$	$-\frac{\omega^2}{-\omega^3+\omega}$	$-\frac{\omega^3}{-\omega^3+\omega^2}$	$\frac{3}{2\omega^3}$

(d) EDiv_3

Fig. 6: The four matrices EDiv_0, EDiv_1, EDiv_2, and EDiv_3

Now let us consider the case when we stack all the k-th rows from each EDiv_k. Note that this corresponds to just collecting the green-colored rows.

$\frac{3}{2}$	$-\frac{\omega}{\omega-1} = \frac{\omega-1}{2}$	$-\frac{\omega^2}{\omega^2-1} = -\frac{1}{2}$	$-\frac{\omega^3}{\omega^3-1} = -\frac{1+\omega}{2}$
$-\frac{\omega^3}{1-\omega} = \frac{\omega-1}{2}$	$\frac{3}{2\omega} = -\frac{3\omega}{2}$	$-\frac{\omega}{\omega^2-\omega} = \frac{1+\omega}{2}$	$-\frac{\omega^2}{\omega^3-\omega} = \frac{\omega}{2}$
$-\frac{\omega^2}{1-\omega^2} = \frac{1}{2}$	$-\frac{\omega^3}{-\omega^2+\omega} = \frac{\omega+1}{2}$	$\frac{3}{2\omega^2} = -\frac{3}{2}$	$-\frac{\omega}{\omega^3-\omega^2} = \frac{\omega+1}{2}$
$-\frac{\omega}{1-\omega^3} = \frac{\omega-1}{2}$	$-\frac{\omega^2}{-\omega^3+\omega} = \frac{\omega}{2}$	$-\frac{\omega^3}{-\omega^3+\omega^2} = \frac{\omega+1}{2}$	$\frac{3}{2\omega^3} = \frac{3\omega}{2}$

Fig. 7: Stacking the rows, i.e. the k-th row from EDiv_k forms the k-th row of the new matrix.

One can also observe that indeed matrix described in Fig 7 matches the values of $\widehat{\mathsf{D}}$ described in Table 1.

The next observation is that upon stacking the k-th columns of EDiv_k, we compute the $\mathsf{ColEDiv}$ matrix, that has the form:

$$\mathsf{ColEDiv} = \begin{cases} -\frac{1}{\omega^j - \omega^i} & , i \neq j \\ 0 & , i = j \end{cases}$$

This corresponds to stacking the violet-colored columns from each of the EDiv_k, but replacing the diagonal elements with 0.

0	$-\frac{1}{1-\omega}$	$-\frac{1}{1-\omega^2}$	$-\frac{1}{1-\omega^3}$
$-\frac{1}{\omega-1}$	0	$-\frac{1}{-\omega^2+\omega}$	$-\frac{1}{-\omega^3+\omega}$
$-\frac{1}{\omega^2-1}$	$-\frac{1}{\omega^2-\omega}$	0	$-\frac{1}{-\omega^3+\omega^2}$
$-\frac{1}{\omega^3-1}$	$-\frac{1}{\omega^3-\omega}$	$-\frac{1}{\omega^3-\omega^2}$	0

Fig. 8: Stacking the columns with diagonal set to 0.

One can observe that this matrix has exactly the form of $\mathsf{ColEDiv}$ described above.

Finally, upon stacking the diagonals (yellow-colored cells) as columns but keeping the diagonal of the new matrix as zeros, we get the DiaEDiv matrix

0	$\frac{1}{1-\omega}$	$\frac{1}{1-\omega^2}$	$\frac{1}{1-\omega^3}$
$\frac{1}{\omega-1}$	0	$\frac{1}{-\omega^2+\omega}$	$\frac{1}{-\omega^3+\omega}$
$\frac{1}{\omega^2-1}$	$\frac{1}{\omega^2-\omega}$	0	$\frac{1}{-\omega^3+\omega^2}$
$\frac{1}{\omega^3-1}$	$\frac{1}{\omega^3-\omega}$	$\frac{1}{\omega^3-\omega^2}$	0

We can see that DiaEDiv $=$ $-$ColEDiv as was observed in Section 4.3.