Faster Post-Quantum zkSNARK Provers Using the LCH Polynomial Basis

Mohammadtaghi Badakhshan[1], Susanta Samanta[2] and Guang Gong[2]

[1] LayerZero Labs mtbadakhshan@gmail.com
[2] University of Waterloo, Waterloo, Canada {ssamanta,ggong}@uwaterloo.ca

Abstract. An important class of plausibly post-quantum zkSNARKs uses univariate-polynomial interactive oracle proofs (IOPs) over binary extension fields $\mathbb{F}_{2^k}$, relying on the Fast Reed–Solomon (RS) Interactive Oracle Proof of Proximity (FRI) for low-degree testing. Their provers fold RS codewords in FRI and perform polynomial arithmetic, notably division and multiplication by subspace vanishing polynomials. Constraint polynomials and RS codewords are in evaluation form, but existing implementations perform this arithmetic in the monomial basis, requiring inverse additive fast Fourier transforms (FFTs) for coefficient recovery and forward FFTs for evaluation. On a general basis, Gao–Mateer and Lin–Chung–Han (LCH) additive FFTs incur $O(n(\log n)^2)$ field additions and $O(n \log n)$ field multiplications for basis conversion per n-point transform, and Lagrange-based FRI folding performs a field inversion per coset.

We carry out this arithmetic directly in the LCH polynomial basis for arbitrary $\mathbb{F}_2$-bases. We introduce a divide-and-conquer algorithm for division by subspace vanishing polynomials directly in the LCH basis. For n input coefficients, it requires $O(n)$ field operations, and on a Cantor basis it requires no field multiplications. We also show that the multiples of vanishing polynomials used for zero-knowledge blinding can be sampled directly through their LCH coefficients. Together, these remove the basis-conversion stage of the LCH additive FFT, leaving only its butterfly. We further express the FRI fold in the LCH basis: on each coset, it is an LCH inverse butterfly followed by an evaluation at the verifier challenge, and for higher-arity folds, the challenge of each stage is the value of a normalized subspace vanishing polynomial at the verifier challenge. Given constants that depend only on the domain, this yields a fold without field inversions whose folding loop performs $n(1 - 1/\kappa)$ field multiplications for a length-n codeword and arity κ; for every input word, it outputs the same next codeword as the classical fold.

We integrate these algorithms into the Aurora prover without changing the proof or the verifier. At 2^{19} constraints over $\mathbb{F}_{2^{256}}$, when all configurations use our FRI fold, the LCH-basis arithmetic speeds up the prover by 2.80× over `libiop` on a general basis and by 1.24× over a Cantor-basis prover that uses the LCH additive FFT. Against these baselines with their Lagrange-based FRI fold, the speedups are 3.63× and 2.05×.

Keywords: zkSNARK, Aurora, FRI, Additive FFT, LCH Polynomial Basis, Binary Extension Fields

1 Introduction

Zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [BCCT12] enable a prover to establish knowledge of a witness for an NP relation while preserving zero knowledge and providing proofs whose size is sublinear in the witness length. Polynomial interactive oracle proofs (IOPs) [BSCS16] form the foundation of a broad class of

these systems. A prominent family relies on the Fast Reed–Solomon Interactive Oracle Proof of Proximity (FRI) [BSBHR18a] for low-degree testing and hash-based polynomial commitment schemes (PCSs). These constructions combine succinctness, transparency, and plausible post-quantum security without requiring a trusted setup. They include STARKs [BSBHR18b], Aurora [BCR+19], and Fractal [COS20], and the underlying FRI-based approach is widely used in industry through production proving systems and libraries such as Plonky3 [Plo] and `Stwo` [Sta24].

The choice of the underlying finite field has direct consequences for prover efficiency. Hash-based commitments permit instantiations over binary extension fields $\mathbb{F}_{2^k}$, which support natural arithmetizations of Boolean computations and binary-field operations, including those arising in symmetric cryptographic primitives. Their arithmetic also admits efficient implementations using carry-less multiplication instructions [Fas26, DP25]. These properties motivate the binary-field instantiations of univariate-polynomial IOPs considered in this work. Their evaluation domains are typically affine $\mathbb{F}_2$-subspaces, whose algebraic structure governs both additive FRI folding and the polynomial arithmetic used to construct the prover's oracles.

In additive FRI, the prover repeatedly folds Reed–Solomon (RS) codewords into shorter codewords. Each round partitions the current domain into localizer cosets and evaluates the interpolant on each coset at a verifier challenge. The localizer maps are subspace vanishing polynomials, linking the folding operation to the structure of the evaluation domain. Although this process reduces the codeword length at each round, its arithmetic can account for a substantial fraction of prover time. In the `libiop` implementation [BSCK+19] of Aurora and Fractal, additive FRI folding incurs one field inversion per localizer coset through Lagrange interpolation, contributing substantially to the arithmetic cost of computing the folded codeword.

Subspace vanishing polynomials also govern the arithmetic required to construct these codewords. A constraint polynomial vanishes on a domain precisely when it is divisible by that domain's vanishing polynomial; computing the quotient therefore expresses constraint satisfaction as a polynomial identity. Adding a random multiple of a vanishing polynomial masks a polynomial for zero knowledge without changing its evaluations on the constraint domain. The constraint data and committed codewords are in evaluation form, whereas conventional implementations perform these intermediate operations in the monomial basis. Oracle construction consequently requires interpolation to recover coefficients, arithmetic on those coefficients, and evaluation over a larger codeword domain.

The common dependence on subspace vanishing polynomials suggests that the choice of polynomial representation is relevant to both computations. The Lin–Chung–Han (LCH) basis [LCH14] is particularly natural in this setting: its elements are normalized products of the subspace vanishing polynomials associated with an ordered evaluation basis. This representation supports efficient interpolation and evaluation through the LCH butterfly and its inverse. A Cantor basis [Can89] provides further simplifications for the LCH and Gao–Mateer [GM10] additive fast Fourier transforms (FFTs), whose practical benefits for Aurora are demonstrated in [BSG26]. However, an LCH additive FFT applied to monomial coefficients also requires conversion to the LCH basis; the inverse transform converts the recovered coefficients back to the monomial basis. Retaining monomial coefficients for the surrounding arithmetic therefore incurs repeated conversions.

Prior work demonstrates the value of the LCH representation for both folding and encoding. Diamond and Posen [DP26] use it in a multilinear PCS and describe the action of their folds on its coefficients. For RS encoding in their Ligero-based construction, Frigo and Shelat [Fas26] retain LCH coefficients between interpolation and evaluation, eliminating the intervening monomial-basis conversions. Ligero [AHIV17] is an IOP based on the MPC-in-the-head paradigm; it does not use FRI folding, and its RS encoding requires no intervening division or multiplication by subspace vanishing polynomials. The

univariate provers studied here require both classical FRI folding and these polynomial operations during oracle construction, so retaining LCH coefficients between transforms alone is insufficient. The question addressed here is whether these computations can be performed directly in the LCH basis for general evaluation domains, while preserving the protocol and verification checks.

We first develop division and multiplication algorithms for subspace vanishing polynomials directly on LCH coefficients. These operations eliminate the monomial-basis conversions in oracle construction, leaving only the inverse butterfly and butterfly for interpolation and evaluation. We then express classical additive FRI folding in the same basis. On each localizer coset, the fold applies an LCH inverse butterfly to the coset's evaluations and evaluates the resulting LCH-basis polynomial at the verifier challenge; for higher-arity folds, the challenge is carried through the subspace vanishing polynomials of the localizer, so that a single verifier challenge suffices. The fold requires no field inversions and $n(1 - 1/\kappa)$ field multiplications for a length-n codeword and arity κ, and it equals the classical fold on every input word. Together, the algorithms support FRI folding and the arithmetic of its input oracles within the LCH basis.

Supporting general evaluation bases is necessary to retain flexibility in the choice of domains. The ordered basis fixes the evaluation domains, and hence the committed codewords and the proof. When the domains are prescribed externally, for example by an existing protocol specification and verifier or by the field representation, a Cantor basis cannot be adopted without changing the proof. Moreover, $\mathbb{F}_{2^k}$ admits Cantor bases only of dimension up to the largest power of two dividing k [Can89]. Our algorithms therefore account explicitly for the normalization factors associated with arbitrary ordered $\mathbb{F}_2$-basis elements. When a Cantor basis is available, division additionally requires no field multiplications. Our arithmetic-cost analysis treats FRI folding and polynomial arithmetic on general bases and quantifies the additional savings provided by the Cantor structure. With our arithmetic, restricting the evaluation domain to a Cantor basis brings no further prover speedup: the remaining butterflies cost the same on every basis, and the prover is equally fast on general and Cantor bases, within 2% at every size (Section 8). A prover can therefore use the basis that the specification or the field representation prescribes.

We select Aurora as our implementation target because its support for rank-1 constraint systems (R1CS) and its transparent setup have enabled applications in post-quantum signatures, signature aggregation, and authorization protocols. Aurora provides quasi-linear prover complexity and polylogarithmic proof size [BCR+19] and underlies the Preon signature scheme [CCC+23]. Zhang et al. [ZSE+24] instantiate SNARK-based signature aggregation with Aurora and Fractal and evaluate the constructions using `libiop` [BSCK+19]. The authors of [BG25] also use `libiop` to implement an Aurora-based authentication circuit using Merkle hash trees. These applications establish Aurora as a practically relevant setting in which improvements to prover arithmetic can benefit several cryptographic tasks.

Preon [CCC+23] already exploits Aurora's support for binary extension fields to arithmetize its AES-based one-way-function circuit, providing a concrete example of the relevance of binary-field proving. Further opportunities arise in signature aggregation: the comparison of Zhang et al. [ZSE+24] with SPHINCS+ [BHK+19] considers algebraic hashes such as Griffin [GHR+23] to reduce the R1CS cost of verification. Aggregating signatures of the standardized SPHINCS+ variant, SLH-DSA, however, requires proving correct evaluation of its prescribed SHA-2 or SHAKE functions [NIS24]. Their bit-oriented structure naturally motivates binary-field arithmetizations [Fas26]. The SHA-256 computations in the authentication circuit of [BG25] provide a further application for this approach. Efficient binary-field proving therefore offers a promising direction for Aurora in applications built from standard symmetric primitives. We target the arithmetic needed to realize this potential: additive FRI folding and the multiplication and division by subspace

Table 1: Aurora prover time in seconds over $\mathbb{F}_{2^{256}}$ with binary FRI folds ($\eta = 1$). The baseline columns use the Lagrange-based fold of `libiop`; the "+ LCH fold" columns replace it by Algorithm 4 and thus isolate the effect of the LCH-basis arithmetic. The general-basis columns are evaluated on the standard basis; other bases are supported by the algorithms. Lower is better.

	General basis			Cantor basis		
Constraints	Gao–Mateer AFFT [BSCK+19]	Gao–Mateer + LCH fold (this paper)	LCH-basis arithmetic + LCH fold (this paper)	LCH AFFT [BSG26]	LCH AFFT + LCH fold (this paper)	LCH-basis arithmetic + LCH fold (this paper)
2^9	0.461	0.272	0.128	0.349	0.163	0.129
2^{11}	1.850	1.202	0.569	1.361	0.693	0.572
2^{13}	8.417	5.728	2.552	5.840	3.118	2.512
2^{15}	40.363	30.223	11.339	24.901	14.132	11.362
2^{17}	175.925	134.748	47.553	102.653	60.562	47.760
2^{19}	773.369	597.090	213.070	439.412	266.223	213.945

vanishing polynomials used throughout oracle construction.

The scope of the algorithms extends beyond this case study. The underlying arithmetic is also relevant to binary-field instantiations of other univariate-polynomial proof systems, including STARKs, Fractal, and Polaris [BSBHR18b, COS20, FG22]: the proposed primitives apply wherever those provers use classical additive FRI folds and the corresponding subspace vanishing polynomials.

We integrate these algorithms into Aurora and keep all polynomials in the LCH basis, except the final FRI polynomial, which is sent in the monomial basis so that the proof format is unchanged. This improves prover performance on both general and Cantor evaluation bases. At 2^{19} constraints over $\mathbb{F}_{2^{256}}$ with binary folds and our FRI fold in all configurations, the LCH-basis arithmetic yields median prover speedups of 2.80× over `libiop` on a general basis and 1.24× over the Cantor-basis configuration of [BSG26]; against these baselines with their Lagrange-based fold, the speedups are 3.63× and 2.05× (Table 1); for Fractal, the speedup over `libiop` is 2.88× (Section 8). These gains come from removing the basis-conversion stage from the transforms and the per-coset inversions from FRI folding.

Contributions The algorithms and arithmetic-cost analyses underlying these improvements are summarized in the following contributions:

1. **Polynomial arithmetic in the LCH basis:** We develop division and multiplication algorithms for subspace vanishing polynomials directly in the LCH basis. Our recursive and iterative divide-and-conquer division algorithms support arbitrary $\mathbb{F}_2$-bases. With basis-dependent constants precomputed, the normalized variant uses fewer than n field multiplications and fewer than n additions, where $n = 2^{m+d+1}$ is the input length and 2^m is the divisor degree. With the Cantor basis, division requires no field multiplications (Table 2). Multiplying a polynomial of degree less than 2^m by $\mathbb{Z}_m$ reduces to shifting and uniformly scaling its coefficient vector. For uniformly random blinding polynomials, the scaling is absorbed into coefficient sampling on any basis, so blinding requires no field multiplications. Together, these primitives eliminate the basis-conversion stage of LCH additive FFTs and IFFTs, removing its $O(n(\log n)^2)$ field additions and $O(n \log n)$ field multiplications and leaving only the $O(n \log n)$ butterfly computation. At 2^{19} constraints, this speeds up witness oracle construction by 6.3×, the low-degree-test masking polynomial by 5.8×, and

the lincheck and sumcheck by 6.4× on a general basis, and by 1.6×, 1.5×, and 1.8× on a Cantor basis (Table 3). These stages consist of interpolations and evaluations of codewords together with blinding and division by vanishing polynomials; performing the latter in the LCH basis removes the basis conversion from every transform. The gain is smaller on a Cantor basis, where the conversion needs no multiplications.

2. **FRI folding in the LCH basis:** We express the classical additive FRI fold in the LCH basis. On each localizer coset of size $\kappa = 2^\eta$, the fold is an LCH inverse butterfly followed by an evaluation of the resulting LCH-basis polynomial at the verifier challenge; for higher-arity folds, the challenge of each stage is the value of a normalized subspace vanishing polynomial at the verifier challenge. For an arbitrary ordered $\mathbb{F}_2$-basis and given precomputed domain constants, the fold uses $n(1 - 1/\kappa)$ field multiplications in its folding loop for a length-n codeword and no field inversions, whereas the Lagrange-based fold of `libiop` [BSCK+19] performs one inversion per coset. It equals the classical fold on every input word, so the verifier is unchanged. At 2^{19} constraints and $\eta = 1$, the FRI folding loop takes 1.3 s instead of 174.0 s in `libiop` (Table 3).

2 Related Work

2.1 Additive FFT Algorithm

The analysis in [BSG26] shows that the Gao–Mateer [GM10] and LCH [LCH14] algorithms both consist of a basis-conversion stage followed by a butterfly evaluation stage. The butterfly evaluation costs $n \log_2 n$ additions and $\frac{1}{2} n \log_2 n$ multiplications for both algorithms. Over a general basis, Gao–Mateer basis conversion requires $\frac{1}{4} n (\log_2 n)^2 - \frac{1}{4} n \log_2 n$ additions and $n \log_2 n - n + 1$ multiplications, while LCH requires $O(n(\log_2 n)^2)$ additions and $O(n \log_2 n)$ multiplications. Switching to a Cantor basis eliminates all multiplications in the basis-conversion stage of both algorithms; the Gao–Mateer addition count remains $\frac{1}{4} n (\log_2 n)^2 - \frac{1}{4} n \log_2 n$, whereas the LCH addition count improves to $O(n \log_2 n \log_2 \log_2 n)$. The Cantor FFT, by contrast, has no basis-conversion stage and does not use a butterfly structure; it operates through a recursive polynomial-division procedure requiring $\frac{1}{2} n \log_2 n$ multiplications and $\frac{1}{2} n \log_2 n + \frac{1}{2} n \sum_{r=0}^{\log_2 n - 1} 2^{\mathrm{wt}(r)}$ additions in total [BSG26]. In end-to-end Aurora prover benchmarks for circuit sizes 2^{14} through 2^{19}, replacing the Gao–Mateer FFT with the Cantor FFT yielded speedups of up to ∼1.75×, growing with circuit size as the basis-conversion cost becomes increasingly dominant. The public-domain library `polyeval` [Che25] implements the Cantor-basis LCH additive FFT over $\mathbb{F}_{2^{32}}$ and $\mathbb{F}_{2^{64}}$ with AVX2 and NEON kernels; its basis conversion divides by the sparse Cantor subspace vanishing polynomials using only additions. The Aurora prover of [BSG26] integrates the Cantor-basis LCH additive FFT but performs its polynomial arithmetic in the monomial basis, so every transform still includes the basis-conversion stage. Frigo and Shelat [Fas26] retain LCH coefficients between interpolation and evaluation in a Ligero-based construction, whose RS encoding involves no division or multiplication by vanishing polynomials. In contrast, we keep the polynomials of an FRI-based prover in the LCH basis through division and multiplication by subspace vanishing polynomials, for an arbitrary ordered basis, so that every transform reduces to the LCH butterfly.

2.2 Polynomial Division

Univariate-polynomial-based IOPs reduce constraint satisfaction to polynomial divisibility: if a constraint polynomial $C(X)$ encodes the prover's computation trace over an evaluation domain H, then the statement "all constraints are satisfied" is equivalent to $C(\alpha) = 0$ for every $\alpha \in H$, which holds if and only if the vanishing polynomial $\mathbb{Z}_H(X) = \prod_{\alpha \in H} (X - \alpha)$

divides $C(X)$. The prover exhibits a low-degree quotient $Q(X) = C(X)/\mathbb{Z}_H(X)$, and the verifier checks the identity $C(\zeta) = Q(\zeta) \cdot \mathbb{Z}_H(\zeta)$ at a random challenge ζ; soundness follows from the Schwartz–Zippel lemma [Sch80, Zip79]. Computing this quotient is a core arithmetic primitive in every univariate-polynomial-based zkSNARK prover [BCCT12, Gro16, GWC19, BCR+19, COS20].

Prime-field. Over a prime field $\mathbb{F}_p$ with $n \mid p-1$, the evaluation domain is a multiplicative subgroup $H = \{1, \omega, \ldots, \omega^{n-1}\}$ with vanishing polynomial $\mathbb{Z}_H(X) = X^n - 1$. A natural approach to computing the quotient $Q(X) = C(X)/\mathbb{Z}_H(X)$ is to evaluate both polynomials on H and divide pointwise; however, $\mathbb{Z}_H(\omega^i) = 0$ for every $\omega^i \in H$, making the pointwise quotient an indeterminate $0/0$ form. Groth16 implementations (libsnark [BSCG+14], bellman [BG18], arkworks [ark21]) circumvent this by evaluating on a multiplicative coset $gH = \{g, g\omega, \ldots, g\omega^{n-1}\}$ with $g \notin H$, where $\mathbb{Z}_H$ takes the nonzero constant value $g^n - 1$ at every point, reducing division to scaling all n evaluations by the single precomputed inverse $(g^n - 1)^{-1}$. PLONK-family systems [GWC19] (halo2 [BLGH21], gnark [BGP20], Plonky2 [BWL22]) combine gate, permutation, and lookup constraints into a numerator of degree up to kn, so the quotient has degree up to $(k-1)n$ and can no longer be captured by n evaluation points on a single coset. These systems therefore work on an extended coset gH' of size kn, where the shift $g \notin H'$ ensures that $\mathbb{Z}_H$ is nonzero at every evaluation point (since $H \subset H'$, evaluating on H' itself would encounter zeros of $\mathbb{Z}_H$); on this coset $\mathbb{Z}_H$ takes only k distinct nonzero values, and a precomputed table of their batch-inverted values [Mon87] reduces the per-proof division to $O(kn)$ pointwise multiplications. Chalkias et al. [CJL+24] propose a unified approach that resolves the $0/0$ indeterminacy directly via L'Hôpital's rule for formal derivatives, enabling quotient computation on H without a coset shift. Orthogonally, snarkjs [BP18] performs division in the monomial coefficient basis, where the sparsity of the denominator $X^n - 1$ (only two nonzero terms) reduces the schoolbook algorithm to $O(kn)$ additions for a dividend of degree kn.

Binary-field. Over $\mathbb{F}_{2^k}$ the evaluation domain is an $\mathbb{F}_2$-linear subspace $W \subseteq \mathbb{F}_{2^k}$ of dimension m, whose vanishing polynomial $\mathbb{Z}_m(X) = \prod_{a \in W}(X + a)$ is $\mathbb{F}_2$-linearized with at most $m + 1$ nonzero terms (Section 3). Since $\mathbb{Z}_m$ is monic, schoolbook long division touches at most m non-leading coefficients per step, requiring at most $(L - 2^m) \cdot m$ field multiplications and as many additions for a dividend of length L, a factor of $2^m/m$ fewer than for a dense divisor of the same degree. Under a Cantor basis [Can89] all coefficients of $\mathbb{Z}_m(X)$ lie in $\mathbb{F}_2$, with exactly $2^{\mathrm{wt}(m)}$ nonzero terms, where $\mathrm{wt}(m)$ denotes the Hamming weight of m [BSG26]. Schoolbook division of a polynomial of length L by such a $\mathbb{Z}_m$ requires no field multiplications, and only $(L - 2^m) \cdot (2^{\mathrm{wt}(m)} - 1)$ additions. The basis conversion of `polyeval` [Che25] applies such multiplication-free divisions within the Cantor-basis additive FFT. Table 2 compares both schoolbook counts with our algorithms.

Similar to prime-field coset approaches, division can also proceed in the evaluation domain: an additive inverse FFT recovers the numerator in coefficient form, a forward additive FFT evaluates the numerator on an additive coset $\theta + W$ with $\theta \notin W$, where $\mathbb{Z}_m(\theta + x) = \mathbb{Z}_m(\theta)$ for all $x \in W$ by the linearized property, reducing division to single scalar multiplications by $\mathbb{Z}_m(\theta)^{-1}$. This requires the numerator to have degree less than 2^{m+1}, so that the quotient has degree $< 2^m$ and is uniquely determined by 2^m evaluation points. When the quotient degree is at least 2^m and evaluation on an extended coset $\theta + W'$ with $W' \supset W$ and $\theta \notin W'$ is needed, $\mathbb{Z}_m$ is no longer constant but takes only $|W'|/|W|$ distinct nonzero values on the coset, and a precomputed table of their inverses reduces division to $|W'|$ pointwise multiplications.

In FRI-based zkSNARKs [BSBHR18a, BCR+19, BSBHR18b], every committed polynomial is evaluated on a codeword domain L of size $|H|/\rho$. When the numerator is already available on L, the quotient can be computed pointwise there. For example,

`libiop` [BSCK+19] evaluates the Aurora rowcheck quotient $(f_{Az}f_{Bz} - f_{Cz})/\mathbb{Z}_H$ in this way, using a table of the $|L|/|H|$ distinct values of $\mathbb{Z}_H$ on L. In oracle construction, however, the numerator is given by its evaluations on the constraint domain, and the quotient must be committed on L. Exact division by $\mathbb{Z}_S$ can also be computed pointwise on L, by multiplying the numerator's evaluations with a precomputed table of the $|L|/|S|$ distinct values of $\mathbb{Z}_S^{-1}$ on L. This costs $|L|$ multiplications per proof and a table of up to $|L|$ field elements for a small S, such as the public-input subspace in Aurora, whereas division in coefficient form costs $O(\rho|L|)$ multiplications and no table. Division with remainder, as in the sumcheck, is not determined pointwise at all. In these cases, the quotient is computed in coefficient form between the inverse FFT on the constraint domain and the FFT on L. This is where a fast division algorithm operating directly on coefficients applies.

Lin, Al-Naffouri, and Han [LAH16] showed that in the alternative (unnormalized) LCH basis (Section 3.2), division by $\mathbb{Z}_m$ reduces to coefficient splitting for dividends of degree less than 2^{m+1}. They also give a division algorithm in the alternative basis for an arbitrary divisor, which adapts Newton iteration to that basis and requires $O(n \log n)$ operations with FFT-based polynomial multiplications. Our algorithms are specific to the divisors $\mathbb{Z}_m$ and require no polynomial multiplications. We extend the base case to recursive and iterative divide-and-conquer algorithms (Algorithms 1–2) and a normalized variant (Algorithm 3), avoiding the $O(n)$ multiplications for conversion between the two LCH bases. For input length $n = 2^{m+d+1}$ and precomputed basis-dependent constants, the alternative variant uses $O(nd)$ multiplications and additions; the normalized variant uses $O(n)$ multiplications and additions. Both variants require only additions on a Cantor basis.

2.3 FRI Folding

In each FRI round [BSBHR18a], the prover folds a codeword of length n into one of length n/κ, where $\kappa = 2^\eta$ and η is the localization parameter. On each localizer coset S_j, the next symbol is $P_j(r^{(i)})$, where P_j is the degree-$< \kappa$ interpolant on S_j (Section 3.3). The Aurora implementation in `libiop` [BSCK+19] computes this value directly using an optimized Lagrange formula, including for $\eta > 1$. For $\eta = 1$, Plonky3 [Plo] applies a two-point inverse multiplicative NTT over the size-$\kappa = 2$ cosets and then evaluates the resulting polynomial at the verifier challenge. Similarly, `Stwo` [Sta24] applies the same approach over circle domains using an inverse circle-FFT [HLP24]. For larger folding arities ($\eta > 1$), these implementations use successive binary folds, with the stage challenges derived from a single verifier challenge. These routines fold the current codeword, whose symbols are evaluations over multiplicative or circle domains.

Related Reed–Solomon proximity-testing protocols include STIR [ACFY24] and WHIR [ACFY25]. Folding has also been combined with multilinear techniques. BaseFold, introduced by Zeilberger, Chen, and Fisch [ZCF24], constructs a multilinear polynomial commitment scheme by interleaving FRI folding with multivariate sumcheck. Diamond and Posen [DP26] adapt this approach to binary fields using the LCH polynomial basis [LCH14], defined in Section 3.2. They establish how additive FRI folding acts on coefficients represented in this basis. Their binary fold ($\eta = 1$) applies an inverse LCH butterfly to each two-point coset and combines the resulting coefficients with weights $(1 - r, r)$, corresponding to the binding of a multilinear variable. This differs from classical univariate FRI folding, which evaluates the local polynomial $P_j(r)$ using weights $(1, r)$ on its monomial coefficients. For higher arities ($\eta > 1$), Diamond and Posen compose η successive binary folds, each using an independent challenge, and omit commitments to intermediate codewords through *oracle skipping*. Binius64 [Bin] uses a Cantor basis for its evaluation domains, under which the binary FRI domain map takes the form $\mathbb{Z}_1^{(i)}(X) = X^2 + X$ at every round, avoiding the normalization factors required for general additive bases.

Section 7 develops the corresponding evaluation-form fold for additive domains with an arbitrary ordered basis, using the LCH basis. On each coset, the fold applies one level of the LCH inverse butterfly and evaluates the result at the verifier challenge, as in the binary fold of Diamond and Posen, but with the classical univariate weights. For $\eta > 1$, it derives the stage challenges from a single verifier challenge through the normalized subspace vanishing polynomials $\bar{\mathbb{Z}}_s$, as Plonky3 and `Stwo` derive them by squaring on multiplicative and circle domains. The normalization factors of a general basis become domain constants, so the fold requires no field inversions. It computes the classical fold on every input word and therefore preserves the verifier checks. The per-pair butterfly is thus shared with these folds; what is specific to our fold is its combination of classical single-challenge semantics with additive domains over a general ordered basis, where the stage challenges are the values $\bar{\mathbb{Z}}_s(r)$ and the normalization factors are domain constants.

3 Preliminaries

In this section, we recall the basic definitions and algebraic notions used throughout the paper. These concepts provide the foundation for the constructions and analysis developed in the subsequent sections.

3.1 Algebraic Foundations

Finite Field Let $\mathbb{F}_{2^k}$ be a degree-k extension field of $\mathbb{F}_2$, and let $1_{\mathbb{F}_{2^k}}$ denote the multiplicative identity. Fix an $\mathbb{F}_2$-basis $\{\beta_0, \beta_1, \ldots, \beta_{k-1}\}$ of $\mathbb{F}_{2^k}$. Thus, the elements $\beta_0, \beta_1, \ldots, \beta_{k-1}$ are linearly independent over $\mathbb{F}_2$, and every element of $\mathbb{F}_{2^k}$ admits a unique representation as an $\mathbb{F}_2$-linear combination of these basis elements. For each integer $0 \le i < 2^k$, let $i = \sum_{j=0}^{k-1} i_j 2^j$, with $i_j \in \{0, 1\}$, be its binary expansion. We enumerate the elements of $\mathbb{F}_{2^k}$ as $\{\omega_0, \omega_1, \ldots, \omega_{2^k-1}\}$, where $\omega_i = \sum_{j=0}^{k-1} i_j \beta_j$. Addition in $\mathbb{F}_{2^k}$ is the bitwise XOR of the elements' coordinate vectors with respect to the chosen $\mathbb{F}_2$-basis.

Univariate Polynomial Let $\mathbb{F}_{2^k}[x]$ denote the polynomial ring over $\mathbb{F}_{2^k}$ in the indeterminate x. A univariate polynomial of degree less than n is denoted as

$$f(x) = \sum_{i=0}^{n-1} f_i x^i \in \mathbb{F}_{2^k}[x], \quad f_i \in \mathbb{F}_{2^k}.$$

Subspace Vanishing Polynomial For $m < k$, let $W_m = \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$ denote an m-dimensional $\mathbb{F}_2$-linear subspace of $\mathbb{F}_{2^k}$. The *vanishing polynomial* of the subspace W_m is defined by $\mathbb{Z}_m(x) = \prod_{i=0}^{2^m-1}(x + \omega_i)$. Thus, $\deg(\mathbb{Z}_m) = 2^m$. Moreover, $\mathbb{Z}_m(x)$ is an $\mathbb{F}_2$-linearized polynomial and hence satisfies

$$\mathbb{Z}_m(x + y) = \mathbb{Z}_m(x) + \mathbb{Z}_m(y), \forall x, y \in \mathbb{F}_{2^k}$$

Since $W_{m+1} = W_m \cup (\beta_m + W_m)$, the vanishing polynomials satisfy the recurrence

$$\mathbb{Z}_{m+1}(x) = \big(\mathbb{Z}_m(x)\big)^2 - \mathbb{Z}_m(\beta_m)\,\mathbb{Z}_m(x). \tag{1}$$

3.2 LCH Polynomial Basis

The LCH polynomial basis [LCH14] for $\mathbb{F}_{2^k}[x]$ is constructed from subspace vanishing polynomials, and is defined as

$$\bar{\mathbb{X}} = \{\bar{X}_0(x), \bar{X}_1(x), \ldots, \bar{X}_{2^k-1}(x)\}, \qquad \bar{X}_i(x) = \frac{X_i(x)}{p_i},$$

where $\deg\big(\bar{X}_i(x)\big) = i$. Here,

$$X_i(x) := \prod_{j=0}^{k-1} \big(\mathbb{Z}_j(x)\big)^{i_j}, \qquad p_i := \prod_{j=0}^{k-1} \big(\mathbb{Z}_j(\beta_j)\big)^{i_j}, \tag{2}$$

and $(i_0, i_1, \ldots, i_{k-1}) \in \{0,1\}^k$ denotes the binary representation of i.

A univariate polynomial $f(x) \in \mathbb{F}_{2^k}[x]$ of degree less than n can be expressed in the LCH basis as $f(x) = \sum_{i=0}^{n-1} f_i^{\bar{\mathbb{X}}} \bar{X}_i(x)$. The *LCH basis-conversion* algorithms [LAHC16] convert the monomial-basis coefficient vector $(f_0, f_1, \ldots, f_{n-1})$ of a polynomial to its LCH-basis coefficient vector $(f_0^{\bar{\mathbb{X}}}, f_1^{\bar{\mathbb{X}}}, \ldots, f_{n-1}^{\bar{\mathbb{X}}})$.

The *LCH alternative polynomial basis* [LAH16] is defined as

$$\mathbb{X} = \{X_0(x), X_1(x), \ldots, X_{2^k-1}(x)\}. \tag{3}$$

The *LCH Butterfly* [LCH14, LAH16] is the evaluation stage of the LCH additive FFT: given the LCH-basis coefficient vector $(f_0^{\bar{\mathbb{X}}}, \ldots, f_{n-1}^{\bar{\mathbb{X}}})$ of a polynomial $f(x)$ and an affine coset $\theta + W$ with $W = \langle \beta_0, \ldots, \beta_{\log_2 n - 1} \rangle$, it computes the evaluation vector $\big(f(\alpha)\big)_{\alpha \in \theta + W}$ via a Cooley–Tukey-style butterfly network in $n \log_2 n$ additions and $\frac{1}{2} n \log_2 n$ multiplications,

$$\mathrm{Butterfly}_{\theta+W}\big(f_0^{\bar{\mathbb{X}}}, \ldots, f_{n-1}^{\bar{\mathbb{X}}}\big) = \big(f(\alpha)\big)_{\alpha \in \theta + W}. \tag{4}$$

The *LCH IButterfly* is its inverse: given evaluations $\big(f(\alpha)\big)_{\alpha \in \theta + W}$, it recovers the LCH-basis coefficients $(f_0^{\bar{\mathbb{X}}}, \ldots, f_{n-1}^{\bar{\mathbb{X}}})$ with the same operation count,

$$\mathrm{IButterfly}_{\theta+W}\big((f(\alpha))_{\alpha \in \theta + W}\big) = \big(f_0^{\bar{\mathbb{X}}}, \ldots, f_{n-1}^{\bar{\mathbb{X}}}\big). \tag{5}$$

Cantor basis. An ordered basis $(\beta_0, \ldots, \beta_{m-1})$ is called a *Cantor basis* [Can89] if $\beta_0 = 1$ and $\beta_j^2 + \beta_j = \beta_{j-1}$ for $1 \le j < m$. The field $\mathbb{F}_{2^k}$ admits a Cantor basis of dimension m if and only if $2^{\lceil \log_2 m \rceil}$ divides k. On a Cantor basis, $\mathbb{Z}_j(\beta_j) = 1$ for all j [GM10], so the normalization factors in (2) satisfy $p_i = 1$ for all i, and hence $\bar{X}_i = X_i$. Moreover, $\mathbb{Z}_1(x) = x^2 + x$, and $\mathbb{Z}_j(x)$ has exactly $2^{\mathrm{wt}(j)}$ nonzero coefficients, each equal to one [BSG26]. We call any other ordered basis a *general basis*.

3.3 FRI Fold

Definition 1 (Reed–Solomon code)**.** Let $L \subseteq \mathbb{F}_{2^k}$ be an affine $\mathbb{F}_2$-coset with $|L| = n$, and let $D \le n$ be a degree bound. Write $\rho = D/n$. The Reed–Solomon (RS) code [RS60] of rate ρ on L is the $\mathbb{F}_{2^k}$-linear code

$$\mathrm{RS}[L, D] = \big\{ f \colon L \to \mathbb{F}_{2^k} \;\big|\; \exists\, \hat{f} \in \mathbb{F}_{2^k}[X] \text{ with } \deg \hat{f} < D \text{ and } f = \hat{f}|_L \big\} \tag{6}$$

of length n, dimension D, and minimum distance $n - D + 1$. A word $f \in \mathrm{RS}[L, D]$ is the restriction to L of a unique interpolant of degree less than D.

FRI [BSBHR18a] tests that a purported codeword is close to some such RS word by a sequence of folds. In the i-th fold the prover folds $f^{(i)} \colon L^{(i)} \to \mathbb{F}_{2^k}$ into a shorter word $f^{(i+1)} \colon L^{(i+1)} \to \mathbb{F}_{2^k}$. Let η be the localization parameter of that fold and write $\kappa = 2^\eta$, so

$$\frac{|L^{(i)}|}{|L^{(i+1)}|} = \kappa.$$

Each evaluation set is an affine $\mathbb{F}_2$-coset, given by a shift and an ordered basis. Write

$$L^{(i)} = \theta^{(i)} + \langle \beta_0^{(i)}, \beta_1^{(i)}, \ldots, \beta_{m-1}^{(i)} \rangle$$

($m = \log_2 |L^{(i)}|$), where $\theta^{(i)} \in \mathbb{F}_{2^k}$ is the shift. The *localizer subspace* is the leading η-dimensional linear piece

$$L_0^{(i)} = \langle \beta_0^{(i)}, \ldots, \beta_{\eta-1}^{(i)} \rangle, \qquad |L_0^{(i)}| = \kappa.$$

Let $\mathbb{Z}_\eta^{(i)}$ be the (linearized) vanishing polynomial of $L_0^{(i)}$. This map is κ-to-1 on $L^{(i)}$. Set

$$\beta_s^{(i+1)} := \mathbb{Z}_\eta^{(i)}\big(\beta_{\eta+s}^{(i)}\big),$$

so the next domain is the image coset

$$\theta^{(i+1)} := \mathbb{Z}_\eta^{(i)}\big(\theta^{(i)}\big), \qquad L^{(i+1)} = \theta^{(i+1)} + \big\langle \beta_0^{(i+1)},\ \beta_1^{(i+1)},\ \ldots,\ \beta_{m-\eta-1}^{(i+1)} \big\rangle.$$

Let α_j run over the complementary subspace $\langle \beta_\eta^{(i)}, \ldots, \beta_{m-1}^{(i)} \rangle$ for $j = 0, \ldots, 2^{m-\eta} - 1$. The map $\mathbb{Z}_\eta^{(i)}$ partitions $L^{(i)}$ into the additive cosets $S_j = \theta^{(i)} + \alpha_j + L_0^{(i)}$. Because $\mathbb{Z}_\eta^{(i)}$ is linearized and vanishes on $L_0^{(i)}$, it is constant on each such coset: for every $e \in L_0^{(i)}$,

$$\mathbb{Z}_\eta^{(i)}\big(\theta^{(i)} + \alpha_j + e\big) = \mathbb{Z}_\eta^{(i)}\big(\theta^{(i)} + \alpha_j\big) =: y_j. \tag{7}$$

Linearity then gives

$$y_j = \mathbb{Z}_\eta^{(i)}\big(\theta^{(i)} + \alpha_j\big) = \mathbb{Z}_\eta^{(i)}\big(\theta^{(i)}\big) + \mathbb{Z}_\eta^{(i)}(\alpha_j).$$

As α_j runs over $\langle \beta_\eta^{(i)}, \ldots, \beta_{m-1}^{(i)} \rangle$, the second summand runs over $\langle \beta_0^{(i+1)}, \ldots, \beta_{m-\eta-1}^{(i+1)} \rangle$, so the y_j are exactly the points of $L^{(i+1)}$. The vanishing polynomial of the coset S_j is therefore

$$\mathbb{Z}_{S_j}(X) = \mathbb{Z}_\eta^{(i)}(X) + y_j.$$

Let $F^{(i)}$ interpolate $f^{(i)}$ on $L^{(i)}$ (i.e., $F^{(i)}(x) = f^{(i)}(x)$ for $x \in L^{(i)}$), and write

$$F^{(i)}(X) = Q_j(X)\big(\mathbb{Z}_\eta^{(i)}(X) + y_j\big) + P_j(X), \qquad \deg P_j < \kappa. \tag{8}$$

On S_j the vanishing polynomial is zero, so $P_j(a) = F^{(i)}(a)$ for every $a \in S_j$. Since $S_j \subset L^{(i)}$, we also have $P_j(a) = f^{(i)}(a)$ on S_j. A polynomial of degree less than $\kappa = |S_j|$ is uniquely determined by its values on S_j, hence P_j is the interpolant of $f^{(i)}$ on that coset.

Definition 2 (FRI fold)**.** For a verifier challenge $r^{(i)} \in \mathbb{F}_{2^k}$, the fold of $f^{(i)}$ is the word $f^{(i+1)}$ on $L^{(i+1)}$ with

$$f^{(i+1)}(y_j) := P_j\big(r^{(i)}\big) = \big(F^{(i)}(X) \bmod \big(\mathbb{Z}_\eta^{(i)}(X) + y_j\big)\big)\Big|_{X=r^{(i)}}. \tag{9}$$

If $f^{(i)} \in \mathrm{RS}[L^{(i)}, D]$ and $\kappa \mid D$, then $f^{(i+1)} \in \mathrm{RS}[L^{(i+1)}, D/\kappa]$ [BSBHR18a].

3.4 Aurora IOP

Aurora [BCR+19] is based on a univariate-polynomial IOP for R1CS satisfiability. Here, we consider its instantiation over $\mathbb{F}_{2^k}$. An R1CS instance consists of three sparse matrices $A, B, C \in \mathbb{F}_{2^k}^{m \times (n+1)}$ and a public input $v \in \mathbb{F}_{2^k}^{\ell}$; a prover holding witness $w \in \mathbb{F}_{2^k}^{n-\ell}$ demonstrates that the full assignment $z = (1, v, w)$ satisfies

$$Az \circ Bz = Cz \tag{10}$$

where $\circ$ denotes the entrywise (Hadamard) product. The parameter ℓ denotes the number of public-input field elements and is independent of the field-extension degree k. In this

subsection, m and n denote the numbers of constraints and variables, as in [BCR+19]; elsewhere, m is a subspace dimension and n a length.

The constraint domain H_1 and variable domain H_2 are $\mathbb{F}_2$-linear subspaces of $\mathbb{F}_{2^k}$ with $|H_1| = m$ and $|H_2| = n + 1$, after padding these sizes to powers of two if necessary. We choose these subspaces to be nested, so that $H := H_1 \cup H_2$ is a linear subspace of size $t = \max(m, n + 1)$. The codeword domain L is an affine subspace satisfying $L \cap H = \varnothing$. All domains are derived from one ordered basis $(\beta_0, \ldots, \beta_{k-1})$ of $\mathbb{F}_{2^k}$: H_1, H_2, and H are spanned by its first $\log_2 m$, $\log_2(n+1)$, and $\log_2 t$ elements, and $L = \theta + \langle \beta_0, \ldots, \beta_{\log_2 |L|-1} \rangle$ with $\theta \notin \langle \beta_0, \ldots, \beta_{\log_2 |L|-1} \rangle$. Since the LCH basis is determined by the ordered basis $(\beta_j)_j$ and not by the shift (Section 3.2), coefficients obtained by an IButterfly on any of these subspaces can be evaluated by a Butterfly on L. Let b denote the degree slack used for zero-knowledge masking, chosen according to the protocol's query bound. Let D be the degree bound tested by FRI in Phase 5: the bound $\max(2t + b - 1, m + 2b - 1)$ of the sumcheck masking polynomial (Phase 3) and the rowcheck quotient (Phase 4), rounded up to a bound that FRI can test. We require $|L| \geq D/\rho$ for the RS rate ρ of Section 3.3. We use $\hat{f}$ to denote a polynomial and $f = \hat{f}|_L$ to denote its evaluation vector on L. At the arithmetization stage, prover-supplied polynomial oracles are represented by such evaluation vectors and committed using Merkle trees; virtual oracles are derived from existing oracles rather than committed separately. Verifier challenges are derived by Fiat–Shamir hashing (BCS transformation [BSCS16]).

Once the encodings have been checked for consistency with the instance and assignment, the R1CS relation (10) is equivalent to

$$\hat{f}_{Az}(a)\, \hat{f}_{Bz}(a) - \hat{f}_{Cz}(a) = 0, \qquad \text{for all } a \in H_1. \tag{11}$$

We define

$$F(X) := \hat{f}_{Az}(X)\, \hat{f}_{Bz}(X) - \hat{f}_{Cz}(X).$$

Relation (11) holds if and only if F is divisible by the vanishing polynomial $\mathbb{Z}_{H_1}(X) = \prod_{a \in H_1}(X - a)$; equivalently, there exists a polynomial G such that

$$F(X) = G(X)\, \mathbb{Z}_{H_1}(X). \tag{12}$$

Aurora checks this divisibility relation through a virtual quotient oracle and a low-degree proximity test, together with the checks that link the encodings to the R1CS instance.

We organize the prover pipeline into five computational phases, which need not correspond to five separate communication rounds. We describe each phase in terms of the polynomial primitives it requires and their costs.

Phase 1 (Oracle construction). The prover first computes the matrix-vector products Az, Bz, Cz and interpolates their entries on H_1 using additive IFFTs. The resulting polynomials are evaluated on L using additive FFTs, after any required zero-knowledge masking. The witness oracle f_w requires additional work. Let $S := H_2^{\leq \ell}$ be the subset of H_2 indexing the constant entry and the public input, so that $|S| = \ell + 1$. We take $\ell + 1$ to be a power of two and S to be the subspace spanned by the first $\log_2(\ell + 1)$ basis vectors of H_2, so that $\mathbb{Z}_S$ is a subspace vanishing polynomial. Let $\hat{f}_{(1,v)}$ be the polynomial of degree less than $\ell + 1$ interpolating $(1, v)$ on S. Initially, $\hat{f}_z$ is the polynomial interpolating z on H_2, and

$$\hat{f}'_w(X) := \hat{f}_z(X) - \hat{f}_{(1,v)}(X)$$

vanishes on S. Hence, computing $\hat{f}_w = \hat{f}'_w / \mathbb{Z}_S$ requires an *exact polynomial division by a vanishing polynomial* in coefficient form.

For zero knowledge, the witness-dependent encodings are randomized by adding $\mathbb{Z}_T \cdot \hat{\mu}$, with $\deg(\hat{\mu}) < b$, where $T = H_1$ for each matrix-product encoding and $T = H_2 \setminus S$ for the

witness quotient. These additions preserve the encoded values on the relevant domains and require a *polynomial multiplication by a vanishing polynomial*. For the witness, the prover adds $\mathbb{Z}_{H_2} \cdot \hat{\mu}$ to $\hat{f}'_w$ before dividing by $\mathbb{Z}_S$. Since $S \subset H_2$, $\mathbb{Z}_S$ divides $\mathbb{Z}_{H_2}$, and the quotient is $\hat{f}'_w/\mathbb{Z}_S + \mathbb{Z}_{H_2 \setminus S} \cdot \hat{\mu}$. In the LCH basis, this blinding requires no polynomial multiplication: it suffices to sample LCH coefficients directly (Section 5). We reuse the same notation for the randomized polynomials. Their degree bounds are $\deg(\hat{f}_{Mz}) < m + b$ for $M \in \{A, B, C\}$ and $\deg(\hat{f}_w) < n - \ell + b$. The four codewords $f_w, f_{Az}, f_{Bz}, f_{Cz}$ are committed in the first BCS round, together with masking oracles for the sumcheck (Phase 3) and the combined low-degree test (Phase 5). The virtual oracle $f_z(a) = f_w(a)\mathbb{Z}_S(a) + \hat{f}_{(1,v)}(a)$, for $a \in L$, is then assembled on L. Its underlying polynomial has degree less than $n + 1 + b$ and agrees with the public input on S by construction.

Phase 2 (Lincheck). A verifier challenge α, sent together with random weights s_A, s_B, s_C, defines interpolation polynomials $\hat{p}_\alpha$ and $\hat{p}_\alpha^{(M)}$ for $M \in \{A, B, C\}$. $\hat{p}_\alpha$ interpolates the powers $(\alpha^0, \ldots, \alpha^{m-1})$ on H_1; each $\hat{p}_\alpha^{(M)}$ interpolates $M^\top \vec{\alpha}$ on H_2, requiring one sparse matrix-vector product per matrix. To obtain polynomials on the common summation domain H, the values of $\hat{p}_\alpha$ are set to zero on $H \setminus H_1$, and those of $\hat{p}_\alpha^{(M)}$ are set to zero on $H \setminus H_2$. The prover combines the three matrices before interpolation, so only $\hat{p}_\alpha$ and $\sum_M s_M \hat{p}_\alpha^{(M)}$ are recovered via IFFT on H (of size t) and evaluated on L via FFT. For each $M \in \{A, B, C\}$, define

$$\hat{q}_M(X) := \hat{f}_{Mz}(X)\, \hat{p}_\alpha(X) - \hat{f}_z(X)\, \hat{p}_\alpha^{(M)}(X).$$

The resulting claims $\sum_{a \in H} \hat{q}_M(a) = 0$ are checked jointly in Phase 3.

Phase 3 (Combined lincheck). The three lincheck claims $\sum_{a \in H} \hat{q}_M(a) = 0$ are amortized with the weights s_A, s_B, s_C into a single univariate sumcheck on $\hat{f} = \sum_M s_M \hat{q}_M + \hat{m} = \big(\sum_M s_M \hat{f}_{Mz}\big)\hat{p}_\alpha - \hat{f}_z \sum_M s_M \hat{p}_\alpha^{(M)} + \hat{m}$, where $\hat{m}$ is a masking polynomial for zero knowledge (defined below). The prover decomposes

$$\hat{f}(X) = \hat{g}(X) + \sigma\, X^{t-1} + \mathbb{Z}_H(X) \cdot \hat{h}(X) \tag{13}$$

Here $\deg(\hat{g}) < t - 1$ and $\sigma \in \mathbb{F}_{2^k}$. Since $\hat{g}$ and $\mathbb{Z}_H \hat{h}$ sum to zero over H and $\sum_{a \in H} a^{t-1} \neq 0$ [BCR+19, Lemma 5.4], the claim $\sum_{a \in H} \hat{f}(a) = 0$ holds if and only if $\sigma = 0$. Since $\deg(\hat{f}_z) < n + 1 + b$ and $\deg(\hat{p}_\alpha^{(M)}) < t$, we have $\deg(\hat{f}) < 2t + b - 1$; the prover recovers the coefficients of $\hat{f}$ by an IFFT on $\theta + \langle \beta_0, \ldots, \beta_{c-1} \rangle \subseteq L$, where c is the smallest integer with $2^c \geq 2t + b - 1$. *Polynomial division with remainder* by $\mathbb{Z}_H$ is then performed in coefficient form. Only $\hat{h}$ is evaluated on L via a forward FFT; $h = \hat{h}|_L$ is committed in the second BCS round, while g is a virtual oracle computed pointwise from f, h, and σ by (13) and enters the low-degree test. For zero knowledge, a masking polynomial $\hat{m} = \mathbb{Z}_H \cdot \hat{h}' + \hat{g}'$ is constructed, where $\hat{h}'$ is uniformly random of degree less than $t + b - 1$ and $\hat{g}'$ has degree less than t with $\sum_{a \in H} \hat{g}'(a) = 0$, so that $\deg(\hat{m}) < 2t + b - 1$, the degree bound of $\hat{f}$. Computing $\mathbb{Z}_H \cdot \hat{h}'$ requires a *polynomial multiplication by a vanishing polynomial*, and enforcing $\sum_{a \in H} \hat{g}'(a) = 0$ requires characterizing which polynomials sum to zero over H. Hence $\sum_{a \in H} \hat{m}(a) = 0$. The masking oracle $\hat{m}|_L$ is committed in the first BCS round.

Phase 4 (Rowcheck). To check the Hadamard constraint, the prover forms the virtual quotient oracle

$$r(a) := \frac{f_{Az}(a)\, f_{Bz}(a) - f_{Cz}(a)}{\mathbb{Z}_{H_1}(a)}, \qquad a \in L.$$

Since $L \cap H_1 = \varnothing$, the denominator is nonzero throughout L, so these evaluations are well defined. If z satisfies the R1CS constraints and the encodings are consistent with z, the numerator polynomial F vanishes on H_1. It is therefore divisible by $\mathbb{Z}_{H_1}$, and $r = G|_L$ for the quotient polynomial in (12). In an honest execution, the bounds $\deg(\hat{f}_{Mz}) < m + b$ imply $\deg(G) < m + 2b - 1$. The quotient evaluation vector is included in the combined low-degree proximity test in Phase 5. Thus, disjointness makes the evaluations well defined; satisfaction of the constraints makes the quotient a polynomial. Phases 2 and 3 link the matrix-product encodings to the assignment, whereas Phase 4 checks their Hadamard relation, with the required low-degree conditions enforced through Phase 5.

Phase 5 (Combined low-degree test). The required oracle and virtual-oracle codewords are batched using random linear combinations and the protocol's degree adjustments. In particular, the stricter bound $\deg(\hat{g}) < t - 1$ must not be lost when batching with codewords of larger degree. With the common degree bound D, the batch includes both g and the shifted codeword

$$\left\{a^{D-(t-1)} g(a)\right\}_{a \in L}.$$

The resulting combined codeword is tested for proximity to the Reed–Solomon code of degree bound D using FRI [BSBHR18a]. For zero knowledge, a random masking codeword, obtained from a polynomial of degree less than D and committed before the batching challenges, is added to the linear combination. This commitment is included in the first BCS round.

4 Polynomial Division Algorithm

In the Aurora prover pipeline (Section 3.4), the LCH IButterfly recovers LCH-basis coefficients from evaluations, and the LCH Butterfly evaluates from LCH-basis coefficients. If polynomial division can be performed directly in the LCH basis, the entire pipeline, i.e., interpolation, division, and re-evaluation, requires no conversion to or from the monomial basis, which removes the basis-conversion stage from every transform. This is the goal of the algorithms developed in this section.

Given univariate polynomials $A(x)$ and $B(x)$ with $\deg(B) < \deg(A)$, the polynomial division algorithm finds a unique quotient $Q(x)$ and remainder $R(x)$ such that $A(x) = B(x)Q(x) + R(x)$, where $\deg(R) < \deg(B)$. In this section, we present an efficient divide-and-conquer algorithm for division by the subspace vanishing polynomial $\mathbb{Z}_m(x)$, first in the LCH alternative basis and then in the LCH basis. The latter is particularly important in practice, as the LCH IButterfly produces and the LCH Butterfly consumes normalized LCH-basis coefficients, so performing division directly in the LCH basis avoids the $O(n)$-multiplication cost of converting to the alternative basis and back.

We first recall a property of the LCH alternative basis that characterizes the quotient when the dividend degree is bounded; our algorithms rely on it.

Lemma 1. *[LAH16] Let $m \geq 0$, and let $A(x)$ be a polynomial of degree $\deg(A) < 2^{m+1}$, expressed in the LCH alternative basis as $A(x) = \sum_{j=0}^{h-1} a_j^{\mathbb{X}} X_j(x)$, where $h \leq 2^{m+1}$. The quotient $Q(x)$ obtained by dividing $A(x)$ by $\mathbb{Z}_m(x)$ is given by $Q(x) = \sum_{\ell=0}^{2^m-1} a_{\ell+2^m}^{\mathbb{X}} X_\ell(x)$.*

Proof. For $0 \leq i < 2^m$, the definition of the LCH alternative basis gives $X_{2^m+i}(x) = \mathbb{Z}_m(x) X_i(x)$. Splitting the expansion of $A(x)$ at index 2^m, we obtain

$$A(x) = \sum_{i=0}^{2^m-1} a_i^{\mathbb{X}} X_i(x) + \sum_{i=2^m}^{h-1} a_i^{\mathbb{X}} X_i(x) = \sum_{i=0}^{2^m-1} a_i^{\mathbb{X}} X_i(x) + \mathbb{Z}_m(x) \sum_{i=2^m}^{h-1} a_i^{\mathbb{X}} X_{i-2^m}(x).$$

The first sum has degree less than $2^m = \deg \mathbb{Z}_m$, so it is the remainder. The second sum is $\mathbb{Z}_m(x)$ times the stated quotient. □

Lemma 1 gives a direct coefficient split when the dividend has degree less than 2^{m+1}. The lower 2^m coefficients form the remainder, while the upper coefficients form the quotient. This works because, for $0 \le i < 2^m$, the basis element $X_{2^m+i}(x)$ factors as $\mathbb{Z}_m(x)X_i(x)$. For a larger dividend, however, the same split cannot be applied to every high-degree basis element. In particular, $X_{2^{m+d}}(x)$ is divisible by $\mathbb{Z}_m(x)$, but its quotient generally expands across several elements in an alternative basis rather than consisting of a single basis element. Dividing such terms one at a time would therefore require repeated updates to lower coefficient blocks.

Since $p_{2^j} = \mathbb{Z}_j(\beta_j)$, the recurrence for vanishing polynomials takes the form $\mathbb{Z}_{j+1}(x) = \mathbb{Z}_j(x)\big(\mathbb{Z}_j(x) + p_{2^j}\big)$. Iterating this recurrence for $j = m, \ldots, m+d-1$ yields

$$X_{2^{m+d}}(x) = \mathbb{Z}_m(x) \prod_{j=m}^{m+d-1} \big(\mathbb{Z}_j(x) + p_{2^j}\big).$$

Expanding the product gives one term for each $i = \sum_{r=0}^{d-1} i_r 2^r$ with $i_r \in \{0,1\}$, namely

$$\prod_{r=0}^{d-1} \mathbb{Z}_{m+r}(x)^{i_r}\, p_{2^{m+r}}^{1-i_r} = X_{2^m i}(x)\, p_{2^m(2^d-1-i)}$$

by (2), since $2^d - 1 - i = \sum_{r=0}^{d-1}(1-i_r)2^r$. Summing over $0 \le i < 2^d$ gives the explicit quotient. We summarize the above discussion in the following proposition.

Proposition 1. *Let $m \ge 0$ and $d \ge 1$, with $m + d < k$. Define*

$$D_{d,m}(x) = \sum_{i=0}^{2^d-1} p_{2^m(2^d-1-i)} X_{2^m i}(x).$$

Then, in the LCH alternative basis, $X_{2^{m+d}}(x) = \mathbb{Z}_m(x) D_{d,m}(x)$. In particular, $D_{d,m}$ is the quotient of $X_{2^{m+d}}$ by $\mathbb{Z}_m$, with zero remainder.

Algorithm 1 avoids these repeated updates by recursively splitting the dividend and, during each merge step, applying the expansion of Proposition 1 once to the corresponding block. The following theorem establishes the correctness of Algorithm 1. Its proof is deferred to Appendix A.1.

Theorem 1. *Algorithm 1 correctly computes the quotient $Q(x)$ and remainder $R(x)$ of the division of $A(x)$ by the vanishing polynomial $\mathbb{Z}_m(x)$.*

The following lemma establishes the equivalence between Algorithms 1 and 2. Its proof is deferred to Appendix A.2.

Lemma 2. *For every valid input polynomial $A(x) = \sum_{i=0}^{2^{d+m+1}-1} a_i^{\mathbb{X}} X_i(x)$, Algorithm 2 returns the same pair $(Q(x), R(x))$ as Algorithm 1. Hence, Algorithm 2 is an iterative realization of Algorithm 1.*

4.1 Detailed Cost Analysis of Algorithm 2

Algorithm 2 represents the explicit iterative unrolling of the recursive bisection defined in Algorithm 1. To quantify the exact arithmetic cost of the polynomial division, we analyze the finite field operations. Let the input polynomial $A(x)$ have degree less than $n = 2^{m+d+1}$, which is processed as an array of 2^{d+1} distinct blocks, each containing exactly 2^m field elements.

Proposition 2. *Algorithm 2 performs exactly $2^m\big(d \cdot 2^d - 2^d + 1\big)$ field multiplications and the same number of field additions.*

Algorithm 1 Recursive polynomial division by $\mathbb{Z}_m(x)$ in the LCH alternative basis

Require: $A(x) = \sum_{i=0}^{2^{d+m+1}-1} a_i^{\mathbb{X}} X_i(x) \in \mathbb{F}_{2^k}[x]$ of degree $< 2^{d+m+1}$ with $d+m+1 \le k$, expressed in the LCH alternative basis.

Ensure: $Q(x), R(x) \leftarrow \mathsf{RecDivLCHAlt}\big(A(x), d, m\big)$: $Q(x)$ of degree $< 2^{d+m+1} - 2^m$, and $R(x)$ of degree $< 2^m$ are in the LCH alternative basis and $A(x) = \mathbb{Z}_m(x) \cdot Q(x) + R(x)$.

1: **if** $d = 0$ **then**

2: Let

$$R(x) = \sum_{i=0}^{2^m-1} a_i^{\mathbb{X}} X_i(x) \quad \text{and} \quad Q(x) = \sum_{i=0}^{2^m-1} a_{i+2^m}^{\mathbb{X}} X_i(x).$$

3: **return** $Q(x)$, $R(x)$

4: **end if**

5: Let

$$A_0(x) = \sum_{i=0}^{2^{m+d}-1} a_i^{\mathbb{X}} X_i(x) \quad \text{and} \quad A_1(x) = \sum_{i=0}^{2^{m+d}-1} a_{i+2^{m+d}}^{\mathbb{X}} X_i(x).$$

6: Compute

$$Q_0(x), R_0(x) \leftarrow \mathsf{RecDivLCHAlt}\big(A_0(x), d-1, m\big),$$
$$Q_1(x), R_1(x) \leftarrow \mathsf{RecDivLCHAlt}\big(A_1(x), d-1, m\big).$$

7: Compute

$$R(x) = R_0(x),$$
$$Q(x) = Q_0(x) + R_1(x) \times \left(\sum_{i=0}^{2^d-1} p_{2^m \cdot (2^d - i - 1)} \cdot X_{2^m \cdot i}(x) \right) + Q_1(x) \times X_{2^{d+m}}(x).$$

8: **return** $Q(x)$, $R(x)$

Proof. The computational cost is entirely determined by the innermost operation in line 8, which performs one field multiplication and one field addition per execution:

$$\vec{\mathbf{q}}[w \cdot 2^m + i] \;\leftarrow\; \vec{\mathbf{q}}[w \cdot 2^m + i] \;+\; \vec{\mathbf{q}}[t \cdot 2^m + i] \cdot \gamma.$$

The scalar $\gamma \leftarrow p_{2^m(2^s - u - 1)}$ is fetched once per (s, u)-pair at line 4 and reused across all v and i iterations, contributing no additional multiplications.

The total execution count of line 8 follows directly from the four nested loops. At each level $s \in \{1, \ldots, d\}$, the second loop (line 3) runs $2^s - 1$ times, the third loop (line 5) runs 2^{d-s} times, and the inner loop (line 7) runs 2^m times, giving a contribution of $(2^s - 1) \cdot 2^{d-s} \cdot 2^m$ field multiplications at level s. Summing over $s = 1, \ldots, d$, we have the exact multiplication cost of the algorithm as

$$\begin{aligned} \sum_{s=1}^{d} (2^s - 1) \cdot 2^{d-s} \cdot 2^m &= 2^m \sum_{s=1}^{d} (2^d - 2^{d-s}) \\ &= 2^m \left(d \cdot 2^d - 2^d + 1 \right). \end{aligned}$$

Since each execution of line 8 contributes exactly one multiplication and one addition, both totals equal to $2^m(d \cdot 2^d - 2^d + 1)$. □

Corollary 1. *If the evaluation domain is constructed using a Cantor basis (i.e., $\beta_0 = 1$ and $\beta_i^2 + \beta_i = \beta_{i-1}$ for $1 \le i \le k-1$), the scalar constants satisfy $p_i = 1$ for all i (see [GM10, Lemma 1]). Consequently, the scaling factor becomes universally $\gamma = 1$. This eliminates the need for multiplication entirely, reducing the multiplication cost of Algorithm 2 to exactly 0 while preserving the required $2^m(d \cdot 2^d - 2^d + 1)$ field additions.*

Remark 1. In terms of the total input size $n = 2^{d+m+1}$, substituting $d = \log_2 n - (m+1)$ into the exact operation count yields

$$2^m(d \cdot 2^d - 2^d + 1) = \frac{n}{2} \left(\log_2 n - (m+2) \right) + 2^m = O(n \log_2 n).$$

Algorithm 2 Polynomial division by $\mathbb{Z}_m(x)$ in the LCH alternative basis

Require: $A(x) = \sum_{i=0}^{2^{d+m+1}-1} a_i^{\mathbb{X}} X_i(x) \in \mathbb{F}_{2^k}[x]$ of degree $< 2^{d+m+1}$ with $d+m+1 \le k$, expressed in the LCH alternative basis.
Ensure: $Q(x), R(x) \leftarrow \mathsf{DivLCHAlt}(A(x), d, m)$: $Q(x)$ of degree $< 2^{d+m+1} - 2^m$, and $R(x)$ of degree $< 2^m$ are in the LCH alternative basis and $A(x) = \mathbb{Z}_m(x) \cdot Q(x) + R(x)$.
$\vec{\mathbf{r}} := [a_i^{\mathbb{X}} \mid 0 \le i < 2^m]$, and $\vec{\mathbf{q}} := [a_i^{\mathbb{X}} \mid 2^m \le i < 2^{m+d+1}]$.
for $s = 1, \ldots, d$ **do**
  for $u = 0, 1, \ldots, 2^s - 2$ **do**
    $\gamma \leftarrow p_{2^m \cdot (2^s - u - 1)}$
    for $v = 0, 1, \ldots, 2^{d-s} - 1$ **do**
      $t \leftarrow v \cdot 2^{s+1} + 2^s - 1, \quad w \leftarrow v \cdot 2^{s+1} + u$
      for $i = 0, \ldots, 2^m - 1$ **do**
        $\vec{\mathbf{q}}[w \cdot 2^m + i] \leftarrow \vec{\mathbf{q}}[w \cdot 2^m + i] + \vec{\mathbf{q}}[t \cdot 2^m + i] \cdot \gamma$
      end for
    end for
  end for
end for
Compute $Q(x)$, $R(x)$ from $\vec{\mathbf{q}}$ and $\vec{\mathbf{r}}$
return $Q(x)$, $R(x)$

Thus, Algorithm 2 achieves quasi-linear complexity in the input size n.

While Algorithms 1 and 2 operate in the LCH alternative basis, the LCH Butterfly and IButterfly produce and consume coefficients in the normalized LCH basis. Using the alternative-basis algorithms would therefore require $O(n)$ multiplications (scaling each coefficient by p_i or p_i^{-1}) to convert before and after each division step. Algorithm 3 avoids these conversions. Figure 1 contrasts the data flow of both algorithms for $d = 3$.

Algorithm 3 Polynomial division by $\mathbb{Z}_m(x)$ in the LCH basis

Require: $A(x) = \sum_{i=0}^{2^{d+m+1}-1} a_i^{\bar{\mathbb{X}}} \bar{X}_i(x) \in \mathbb{F}_{2^k}[x]$ of degree $< 2^{d+m+1}$ with $d+m+1 \le k$, expressed in the LCH basis.
Ensure: $Q(x), R(x) \leftarrow \mathsf{DivLCH}(A(x), d, m)$: $Q(x)$ of degree $< 2^{d+m+1} - 2^m$, and $R(x)$ of degree $< 2^m$ are in the LCH basis and $A(x) = \mathbb{Z}_m(x) \cdot Q(x) + R(x)$.
$\vec{\mathbf{r}} := [a_i^{\bar{\mathbb{X}}} \mid 0 \le i < 2^m]$, and $\vec{\mathbf{q}} := [a_i^{\bar{\mathbb{X}}} \mid 2^m \le i < 2^{m+d+1}]$.
for $s = d, d-1, \ldots, 0$ **do**
  $\gamma \leftarrow p_{2^m (2^s - 1)} \cdot (p_{2^{m+s}})^{-1}$ ▷ where $p_{2^{m+s}} = \mathbb{Z}_{m+s}(\beta_{m+s})$ by (2)
  for $v = 0, 1, \ldots, 2^{d-s} - 1$ **do**
    $t \leftarrow v \cdot 2^{s+1} + 2^s - 1, \quad w \leftarrow t + 2^s$
    for $i = 0, \ldots, 2^m - 1$ **do**
      $\vec{\mathbf{q}}[t \cdot 2^m + i] \leftarrow \vec{\mathbf{q}}[t \cdot 2^m + i] \cdot \gamma$
      if $w < 2^{d+1} - 1$ **then**
        $\vec{\mathbf{q}}[t \cdot 2^m + i] \leftarrow \vec{\mathbf{q}}[t \cdot 2^m + i] + \vec{\mathbf{q}}[w \cdot 2^m + i]$
      end if
    end for
  end for
end for
Compute $Q(x)$, $R(x)$ from $\vec{\mathbf{q}}$ and $\vec{\mathbf{r}}$
return $Q(x)$, $R(x)$

The following theorem establishes the correctness of Algorithm 3. Its proof is deferred to Appendix A.3.

Theorem 2. *Algorithm 3 returns $Q(x), R(x)$ in the LCH basis with $A(x) = \mathbb{Z}_m(x)\, Q(x) + R(x)$, $\deg(R) < 2^m$, and $\deg(Q) < 2^{d+m+1} - 2^m$.*

4.2 Detailed Cost Analysis of Algorithm 3

Algorithm 3 executes the identical mathematical division as Algorithm 2, but operates in the LCH basis instead of the LCH alternative basis. In the LCH basis, the expansion polynomial has uniform coefficients (Lemma 4), so each block is scaled once, and each quotient block is obtained from one other quotient block with a single block addition. Let the input polynomial $A(x)$ have degree less than $n = 2^{m+d+1}$, processed as an array of 2^{d+1} distinct blocks, each containing exactly 2^m field elements.

Proposition 3. *Given the scaling factors γ of all stages, which depend only on the basis and are precomputed, Algorithm 3 performs exactly $2^m\left(2^{d+1}-1\right)$ and $2^m\left(2^{d+1}-d-2\right)$ field multiplications and field additions, respectively.*

Proof. At any stage $s \in \{0,\dots,d\}$, the v-loop runs 2^{d-s} times and the i-loop runs 2^m times, and the multiplication at line 7 is executed once per iteration. Summing across all stages yields the multiplication count

$$\sum_{s=0}^{d} 2^{d-s}\cdot 2^m = 2^m\left(2^{d+1}-1\right).$$

The addition at line 9 is executed only when $w = (v+1)\cdot 2^{s+1}-1 < 2^{d+1}-1$, that is, for $v < 2^{d-s}-1$. Therefore, the total addition count is

$$\sum_{s=0}^{d}\left(2^{d-s}-1\right)\cdot 2^m = 2^m\left(2^{d+1}-1-(d+1)\right) = 2^m\left(2^{d+1}-d-2\right). \qquad \square$$

Remark 2. On a Cantor basis, $p_i = 1$ for all i (Corollary 1), so the LCH basis elements $\bar{X}_i(x) = X_i(x)/p_i$ coincide with the alternative basis elements $X_i(x)$, and the scaling factor of Algorithm 3 satisfies $\gamma = 1$ at every stage. The multiplication at line 7 then becomes the identity, and Algorithm 3 requires no field multiplications and $2^m(2^{d+1}-d-2)$ field additions. Table 2 compares the exact number of additions and multiplications for both algorithms.

An Aurora proof performs two divisions, with dividends of about $|H|$ and $2|H|$ coefficients, whereas each transform on the codeword domain in our experiments runs over $|L| = 128|H|$ points (Section 8), so division is a small part of the prover time. The benefit of Algorithm 3 is that it keeps every polynomial in the LCH basis and thereby removes the basis conversion from every transform, as the stage times in Table 3 show. Compared with the general division of [LAH16], which applies Newton iteration with FFT-based polynomial multiplications and requires $O(n\log n)$ operations, Algorithm 3 requires fewer than n multiplications, fewer than n additions, and no polynomial multiplications.

5 Multiplication by Vanishing Polynomials

Let $h(x) = \sum_{i=0}^{b-1} h_i^{\bar{\mathbb{X}}}\,\bar{X}_i(x)$ with $b \le 2^m$. Since $\mathbb{Z}_m(x) = p_{2^m}\,\bar{X}_{2^m}(x)$ and, for $0 \le i < 2^m$, the indices 2^m and i have disjoint binary representations, the basis elements satisfy $\bar{X}_{2^m}(x)\,\bar{X}_i(x) = \bar{X}_{2^m+i}(x)$, and therefore

$$\mathbb{Z}_m(x)\cdot h(x) = \sum_{i=0}^{b-1} p_{2^m}\,h_i^{\bar{\mathbb{X}}}\,\bar{X}_{2^m+i}(x).$$

That is, the LCH coefficient vector of h is shifted from positions $0,\dots,b-1$ to positions $2^m,\dots,2^m+b-1$, and each coefficient is scaled by p_{2^m}.

Table 2: Arithmetic counts for dividing a polynomial of degree less than $n = 2^{m+d+1}$ by $\mathbb{Z}_m(x)$, which has degree 2^m, over $\mathbb{F}_{2^k}$, excluding the precomputation of basis-dependent constants such as the scaling factors. Schoolbook division performs $n - 2^m = 2^m(2^{d+1} - 1)$ elimination steps, one per quotient coefficient. On a general basis, $\mathbb{Z}_m$ has at most $m + 1$ nonzero terms, and the schoolbook counts are attained when all of them are nonzero. On a Cantor basis, $\mathbb{Z}_m$ has $2^{\mathrm{wt}(m)}$ nonzero terms, each with coefficient one [BSG26]. Pointwise division (Section 2) multiplies the evaluations of the dividend on an affine subspace by a precomputed table of the inverses of $\mathbb{Z}_m$ there, using one multiplication per point; it requires the dividend's evaluations on that subspace and does not compute a remainder.

Algorithm	**Polynomial Basis**	**Evaluation Domain Basis**	**Field Multiplications**	**Field Additions**
Schoolbook	Monomial	General	$m \cdot 2^m(2^{d+1} - 1)$	$m \cdot 2^m(2^{d+1} - 1)$
		Cantor	0	$(2^{\mathrm{wt}(m)} - 1) \cdot 2^m(2^{d+1} - 1)$
Algorithm 2	LCH Alternative	General	$2^m(d \cdot 2^d - 2^d + 1)$	$2^m(d \cdot 2^d - 2^d + 1)$
		Cantor	0	
Algorithm 3	LCH	General	$2^m(2^{d+1} - 1)$	$2^m(2^{d+1} - d - 2)$
		Cantor	0	

In zero-knowledge proof systems, a polynomial $\hat{f}$ of degree less than 2^m is blinded as $\hat{f} + \mathbb{Z}_m \cdot h$ for a uniformly random polynomial h; since $\mathbb{Z}_m$ vanishes on the constraint domain, the values of $\hat{f}$ there are unchanged. With $\hat{f}(x) = \sum_{i=0}^{2^m-1} f_i^{\bar{\mathbb{X}}} \bar{X}_i(x)$, the coefficients of $\hat{f}$ and of $\mathbb{Z}_m \cdot h$ occupy disjoint positions, so the blinded polynomial is obtained by concatenating the LCH coefficients of $\hat{f}$ with the shifted coefficients of h. Since the scaled coefficients $p_{2^m} h_i^{\bar{\mathbb{X}}}$ are uniformly random, the blinding reduces to appending random field elements at LCH positions $\geq 2^m$, without any polynomial multiplication.

The blinding does not require $b \leq 2^m$. For $j \geq 2^m$, some bit j_i with $i \geq m$ is nonzero, so by (2) $\bar{X}_j$ contains the factor $\mathbb{Z}_i$, which vanishes on $\langle \beta_0, \ldots, \beta_{i-1} \rangle \supseteq H$. Hence $\bar{X}_{2^m}, \ldots, \bar{X}_{2^m+b-1}$ are b polynomials of distinct degrees less than $2^m + b$ that vanish on H, and they span the multiples of $\mathbb{Z}_m$ of degree less than $2^m + b$. Sampling their coefficients uniformly therefore gives the same distribution as $\mathbb{Z}_m \cdot h$ for uniformly random h; for $b \leq 2^m$, this is exactly the shift above.

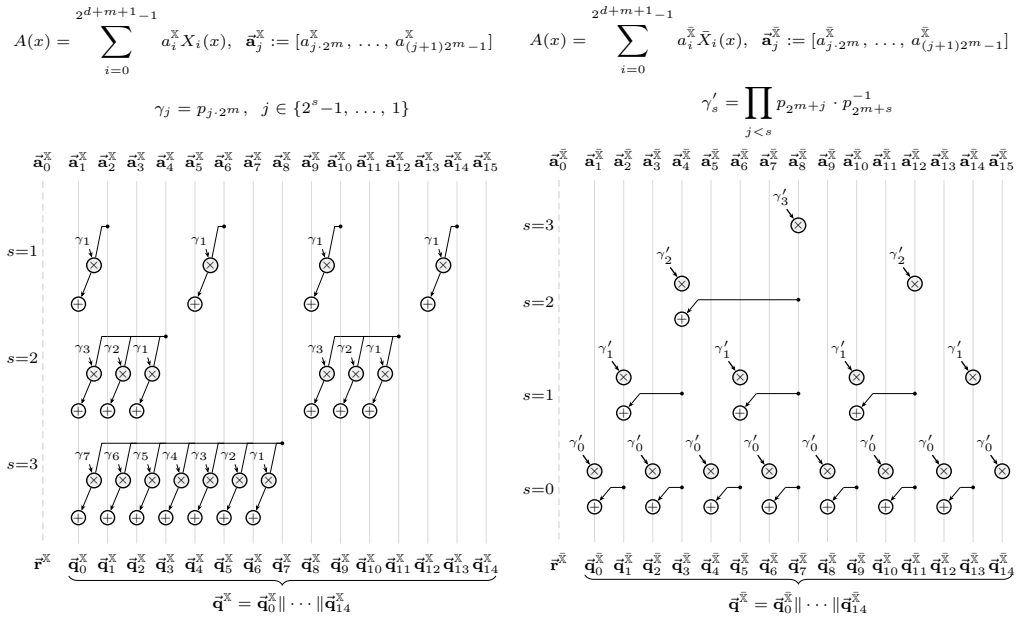
Alternatively, the blinding could be defined in the evaluation domain, by assigning random values on a larger domain containing H. Recovering the coefficients would then require an IButterfly of size at least 2^{m+1} on a subspace $H' \supset H$ (or a truncated inverse FFT of length $2^m + b$, which still processes more points than an IButterfly of size 2^m), whereas in the LCH basis the IButterfly operates only on H and the blinding appends random coefficients. The subsequent Butterfly on the codeword domain L costs the same in both approaches, since the degree of the blinded polynomial does not depend on how the blinding is constructed.

The prover also requires a sumcheck masking polynomial (Phase 3 in Section 3.4) of the form $\hat{m} = \mathbb{Z}_H \cdot \hat{h}' + \hat{g}'$, where $\hat{h}'$ and $\hat{g}'$ are uniformly random polynomials with $\deg(\hat{h}') < |H| + b - 1$ and $\deg(\hat{g}') < |H|$, subject to $\sum_{a \in H} \hat{g}'(a) = 0$. The first term is a multiple of a vanishing polynomial and is sampled as above; the constraint on $\hat{g}'$ has a simple form in the LCH basis.

Proposition 4. *Let* $H = \langle \beta_0, \ldots, \beta_{\log_2 |H|-1} \rangle$ *and let* $g(x) = \sum_{i=0}^{|H|-1} g_i^{\bar{\mathbb{X}}} \bar{X}_i(x)$. *Then* $\sum_{a \in H} g(a) = 0$ *if and only if* $g_{|H|-1}^{\bar{\mathbb{X}}} = 0$.

Proof. By [BCR+19, Lemma 5.4], every polynomial of degree less than $|H| - 1$ sums to zero over H, so only the coefficient of $\bar{X}_{|H|-1}$ can affect the sum. Since $\bar{X}_{|H|-1}$ has leading coefficient $p_{|H|-1}^{-1}$, the same lemma gives $\sum_{a \in H} \bar{X}_{|H|-1}(a) = \varepsilon / p_{|H|-1} \neq 0$, where $\varepsilon \neq 0$ is the linear coefficient of $\mathbb{Z}_H$. □

The prover therefore samples $\hat{g}'$ uniformly from the admissible polynomials by choosing its lower LCH coefficients uniformly and setting its coefficient of $\bar{X}_{|H|-1}$ to zero.



(a) DivLCHAlt (Algorithm 2) (b) DivLCH (Algorithm 3)

Figure 1: Data flow of Algorithm 2 and Algorithm 3 for $d = 3$ and generic m; stages proceed top to bottom. Each wire carries a block $\vec{\mathbf{a}}_j$ of 2^m coefficients, and every addition ($\oplus$) and multiplication ($\otimes$) operates element-wise on vectors of size 2^m.

6 Avoiding Vanishing Polynomial Materialization

In other stages of the Aurora prover, vanishing-polynomial evaluation and sumcheck decomposition can also be performed without constructing monomial-basis representations of the vanishing polynomials. We compute domain evaluations directly from the subspace-polynomial recurrence and obtain the sumcheck decomposition from LCH coefficients.

6.1 Vanishing Polynomial Evaluation over Domains

The virtual oracle f_z (Phase 1), the sumcheck (Phase 3), and the rowcheck (Phase 4) require $\{\mathbb{Z}_m(a)\}_{a\in L}$ for $L = \theta + \langle \beta_0, \ldots, \beta_{\log_2|L|-1} \rangle$. The original `libiop` constructs the sparse monomial representation $\mathbb{Z}_m(x) = \sum_{j=0}^{m} c_j x^{2^j}$ to compute the image domain $\mathbb{Z}_m(\theta) + \langle \mathbb{Z}_m(\beta_0), \ldots, \mathbb{Z}_m(\beta_{\log_2|L|-1}) \rangle$, then enumerates its elements by subset sums. The coefficients of $\mathbb{Z}_m$ are not needed: starting from $\mathbb{Z}_0(x) = x$, the recurrence (1) evaluated at β_i and θ yields $\mathbb{Z}_m(\beta_i)$ and $\mathbb{Z}_m(\theta)$, and the same subset-sum enumeration applies by $\mathbb{F}_2$-linearity:

$$\mathbb{Z}_m\left(\theta + \sum_{i=0}^{\log_2|L|-1} \varepsilon_i \beta_i\right) = \mathbb{Z}_m(\theta) + \sum_{i=0}^{\log_2|L|-1} \varepsilon_i \mathbb{Z}_m(\beta_i), \qquad \varepsilon_i \in \mathbb{F}_2.$$

After computing $\mathbb{Z}_m(\beta_i)$ and $\mathbb{Z}_m(\theta)$, the domain evaluations require $|L| - 1$ additions, without a polynomial transform. Since these values depend only on the domain, they can be computed once per domain.

6.2 Sumcheck Decomposition in the LCH Basis

The sumcheck (Phase 3) requires the coefficient-form decomposition $\hat{f} = \mathbb{Z}_m \cdot \hat{h} + \hat{r}$ with $\deg(\hat{f}) < 2 \cdot 2^m + b - 1$. The original `libiop` constructs the monomial representation of the vanishing polynomial and applies schoolbook division. When $\hat{f}$ is already represented in LCH coefficients (from the preceding IButterfly), the remainder $\hat{r}$ is its lower block of LCH coefficients (positions $0, \ldots, 2^m - 1$), since every $\bar{X}_j$ with $j \geq 2^m$ is divisible by $\mathbb{Z}_m$ (Section 5). If $\deg(\hat{f}) < 2 \cdot 2^m$, as without zero knowledge, the quotient $\hat{h}$ is the upper block scaled by $p_{2^m}^{-1}$, which is the base case $d = 0$ of Algorithm 3. With zero-knowledge masking, $\hat{f}$ has coefficients beyond position $2 \cdot 2^m$, and $\hat{h}$ is computed by Algorithm 3 with $d = 1$ for our parameters. No vanishing-polynomial construction and no iterative long division are required.

7 FRI Folding in the LCH Polynomial Basis

We compute the classical FRI fold (9) by fusing LCH interpolation with evaluation at the challenge. On each coset S_j, the required output is $P_j(r)$, where P_j is the degree-$< \kappa$ polynomial interpolating the input values on S_j. An LCH IButterfly would recover its coefficients, which could then be evaluated at r. Instead, we combine these computations without explicitly constructing the full coefficient vector. The resulting fold eliminates per-coset inversions; inverses of domain-dependent constants are precomputed.

Throughout this section, we fix a round i and omit the superscript (i), writing $L = \theta + \langle \beta_0, \ldots, \beta_{m-1} \rangle$, $r = r^{(i)}$, $n = 2^m$, and $\kappa = 2^\eta$. We denote by $\mathbb{Z}_s(x)$ the vanishing polynomial of $\langle \beta_0, \ldots, \beta_{s-1} \rangle$ and by $\bar{\mathbb{Z}}_s(x) = \mathbb{Z}_s(x)/p_{2^s}$ its normalization, where $p_{2^s} = \mathbb{Z}_s(\beta_s)$ as in (2). With the enumeration of Section 3, the coset $S_j = a_j + \langle \beta_0, \ldots, \beta_{\eta-1} \rangle$, with $a_j = \theta + \alpha_j$ as in Section 3.3, occupies the positions $j\kappa, \ldots, j\kappa + \kappa - 1$ of L, and $y_j = \mathbb{Z}_\eta(a_j)$ is the j-th element of $L^{(i+1)}$.

7.1 Binary Fold

We first consider $\eta = 1$, so that $S_j = \{a_j, a_j + \beta_0\}$. Since $\mathbb{Z}_0(x) = x$, the LCH basis polynomials of degree less than two are 1 and $\bar{\mathbb{Z}}_0(x) = x/\beta_0$. Writing $P_j(x) = A_j + B_j\, \bar{\mathbb{Z}}_0(x)$ and evaluating at the two points of S_j, we obtain

$$B_j = f^{(i)}(a_j) + f^{(i)}(a_j + \beta_0), \qquad A_j = f^{(i)}(a_j) + \bar{\mathbb{Z}}_0(a_j)\, B_j, \tag{14}$$

which is a single butterfly of the LCH IButterfly (5) with twiddle factor $\bar{\mathbb{Z}}_0(a_j)$. Evaluating P_j at the challenge then gives the fold

$$f^{(i+1)}(y_j) = A_j + \bar{\mathbb{Z}}_0(r)\, B_j = f^{(i)}(a_j) + \big(\bar{\mathbb{Z}}_0(r) + \bar{\mathbb{Z}}_0(a_j)\big)\big(f^{(i)}(a_j) + f^{(i)}(a_j + \beta_0)\big). \tag{15}$$

The value $\bar{\mathbb{Z}}_0(r)$ is computed once per round. Since $\bar{\mathbb{Z}}_0$ is $\mathbb{F}_2$-linear, the twiddle factors $\bar{\mathbb{Z}}_0(a_j)$ are subset sums of $\bar{\mathbb{Z}}_0(\theta)$ and $\bar{\mathbb{Z}}_0(\beta_s)$; they depend only on the domain and can be computed with additions alone. Each pair therefore costs a single multiplication, and the only inverted quantity is the domain constant β_0.

7.2 Higher-Arity Fold

For $\eta > 1$, each coset S_j contains $\kappa = 2^\eta$ points, and P_j has degree less than κ. We still use only binary butterflies, applied in η stages in the same order as the IButterfly on S_j. The first stage pairs the points of S_j that differ by β_0, and each later stage pairs the points of a smaller coset. Two observations make this possible.

Stage cosets. The vanishing polynomials of the LCH basis are compositions of quadratic maps. By (1),

$$\mathbb{Z}_{s+1}(x) = q_s\big(\mathbb{Z}_s(x)\big), \qquad q_s(x) = x^2 + p_{2^s}\, x, \tag{16}$$

where q_s is the vanishing polynomial of $\{0, p_{2^s}\}$; it maps both points of a pair $\{u,\, u + p_{2^s}\}$ to the same value. Since $\mathbb{Z}_s$ is $\mathbb{F}_2$-linear and vanishes exactly on $\langle \beta_0, \ldots, \beta_{s-1}\rangle$, it maps S_j onto the *stage-s coset* $S_j^{(s)} = \mathbb{Z}_s(S_j)$, of size $2^{\eta-s}$. Because $\mathbb{Z}_s(\beta_s) = p_{2^s}$, the points of $S_j^{(s)}$ form pairs $\{u,\, u + p_{2^s}\}$, and q_s maps $S_j^{(s)}$ onto $S_j^{(s+1)}$. The stage cosets shrink from $S_j^{(0)} = S_j$ to $S_j^{(\eta)} = \{y_j\}$.

One stage halves the degree. Let $P(x)$ be a polynomial of degree less than $2^{\eta-s}$. Dividing P repeatedly by the quadratic q_s, we can write

$$P(x) = A\big(q_s(x)\big) + \frac{x}{p_{2^s}}\, B\big(q_s(x)\big), \tag{17}$$

where A and B have degree less than $2^{\eta-s-1}$. Both points of a pair $\{u,\, u + p_{2^s}\}$ have the same image $z = q_s(u)$, so on the pair, (17) is the linear polynomial $A(z) + B(z)\, x/p_{2^s}$, written in the LCH basis $\{1,\ x/p_{2^s}\}$. A butterfly as in (14) recovers its coefficients,

$$B(z) = P(u) + P(u + p_{2^s}), \qquad A(z) = P(u) + \frac{u}{p_{2^s}}\, B(z),$$

and a binary fold as in (15) with challenge ξ returns $A(z) + B(z)\, \xi/p_{2^s}$. Over all pairs, these results are the values on $S_j^{(s+1)}$ of the polynomial $A + B\, \xi/p_{2^s}$, which has half the degree of P. By (17), this polynomial takes the value $P(\xi)$ at the point $q_s(\xi)$.

The folding recurrence. Combining the two observations, the fold of S_j at r reduces to a fold of $S_j^{(1)}$ at $\mathbb{Z}_1(r) = q_0(r)$, which in turn reduces to a fold of $S_j^{(2)}$ at $\mathbb{Z}_2(r)$, and so on, until the single point of $S_j^{(\eta)}$ remains. We write the points of $S_j^{(s)}$ as $u = \mathbb{Z}_s(v)$ with $v \in S_j$, so that $u/p_{2^s} = \bar{\mathbb{Z}}_s(v)$ and $\mathbb{Z}_s(r)/p_{2^s} = \bar{\mathbb{Z}}_s(r)$. Starting from $g^{(0)} = f^{(i)}|_{S_j}$, stage s computes

$$g^{(s+1)}\big(\mathbb{Z}_{s+1}(v)\big) = g^{(s)}(u) + \big(\bar{\mathbb{Z}}_s(r) + \bar{\mathbb{Z}}_s(v)\big)\big(g^{(s)}(u) + g^{(s)}(u + p_{2^s})\big), \qquad u = \mathbb{Z}_s(v),\ v \in S_j, \tag{18}$$

and after η stages, $g^{(\eta)}(y_j) = P_j(r)$ is the fold.

Example. For $\eta = 2$, let $a = a_j$, so that $S_j = \{a,\, a + \beta_0,\, a + \beta_1,\, a + \beta_0 + \beta_1\}$. Stage 0 folds the pairs $\{a,\, a + \beta_0\}$ and $\{a + \beta_1,\, a + \beta_1 + \beta_0\}$ as in (15), with twiddle factors $\bar{\mathbb{Z}}_0(a)$ and $\bar{\mathbb{Z}}_0(a + \beta_1)$ and challenge value $\bar{\mathbb{Z}}_0(r)$. It produces two values, at $\mathbb{Z}_1(a)$ and $\mathbb{Z}_1(a + \beta_1) = \mathbb{Z}_1(a) + p_2$. Stage 1 folds this pair with twiddle factor $\bar{\mathbb{Z}}_1(a)$ and challenge value $\bar{\mathbb{Z}}_1(r)$, and returns $f^{(i+1)}(y_j) = P_j(r)$. The coset of size four is thus folded with three multiplications.

Relation to the IButterfly. The IButterfly on S_j keeps both outputs of every butterfly and returns all κ LCH coefficients of P_j. The fold keeps only the combination $A + \bar{\mathbb{Z}}_s(r)\, B$ at each stage, so it uses $\kappa - 1$ butterflies per coset instead of $\frac{\kappa}{2}\log_2 \kappa$. The challenge enters only through the η values $\bar{\mathbb{Z}}_s(r)$, which are obtained by iterating (16) on r.

Algorithm 4 evaluates (18) in place. At stage s, the entries g_{2t} and g_{2t+1} hold the values at the pair $\{u, u + p_{2^s}\}$ with $u = \mathbb{Z}_s(v_{j,s,t})$ and $v_{j,s,t} = a_j + \sum_e t_e\, \beta_{s+1+e}$. By linearity, the twiddle factor of this pair is

$$\bar{\mathbb{Z}}_s(v_{j,s,t}) = c_{j,s} + \delta_{s,t}, \qquad c_{j,s} = \bar{\mathbb{Z}}_s(a_j), \qquad \delta_{s,t} = \sum_e t_e\, \bar{\mathbb{Z}}_s(\beta_{s+1+e}),$$

where both terms are subset sums of domain constants. With $\lambda_s = \bar{\mathbb{Z}}_s(r)$, the algorithm adds $\lambda_s + c_{j,s}$ once per coset and stage.

Algorithm 4 FRI fold in the LCH basis (round i)

Require: Word $f^{(i)}$ on L, with S_j at positions $j\kappa, \dots, j\kappa + \kappa - 1$; challenge r; $\kappa = 2^\eta$; precomputed $p_{2^s}^{-1}$, coset constants $c_{j,s}$, and offsets $\delta_{s,t}$ for $s < \eta$.
Ensure: $f^{(i+1)}$ on $L^{(i+1)}$ with $f^{(i+1)}(y_j) = P_j(r)$ for all j.

1: $\xi \leftarrow r$
2: **for** $s = 0, \dots, \eta - 1$ **do**
3: $\quad \lambda_s \leftarrow \xi \cdot p_{2^s}^{-1}$; $\quad \xi \leftarrow \xi^2 + p_{2^s}\,\xi$ ▷ $\lambda_s = \bar{\mathbb{Z}}_s(r)$, $\xi = \mathbb{Z}_{s+1}(r)$
4: **end for**
5: **for** $j = 0, \dots, n/\kappa - 1$ **do**
6: $\quad (g_t)_{t<\kappa} \leftarrow \left(f^{(i)}[j\kappa + t]\right)_{t<\kappa}$
7: $\quad$ **for** $s = 0, \dots, \eta - 1$ **do**
8: $\quad\quad \tau \leftarrow \lambda_s + c_{j,s}$
9: $\quad\quad$ **for** $t = 0, \dots, 2^{\eta-s-1} - 1$ **do**
10: $\quad\quad\quad g_t \leftarrow g_{2t} + (\tau + \delta_{s,t})\,(g_{2t} + g_{2t+1})$ ▷ inverse butterfly and evaluation at λ_s
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad f^{(i+1)}[j] \leftarrow g_0$
14: **end for**

The following theorem establishes the correctness of Algorithm 4. Its proof is deferred to Appendix A.4.

Theorem 3. *Let $L = \theta + \langle \beta_0, \dots, \beta_{m-1} \rangle \subseteq \mathbb{F}_{2^k}$, let $\kappa = 2^\eta$, and let $S_j = a_j + \langle \beta_0, \dots, \beta_{\eta-1} \rangle \subseteq L$ be its localizer cosets. For every word $f^{(i)} \colon L \to \mathbb{F}_{2^k}$ and every challenge $r \in \mathbb{F}_{2^k}$, Algorithm 4 returns the classical fold (9), that is, $f^{(i+1)}(y_j) = P_j(r)$ for every j and $y_j = \mathbb{Z}_\eta(a_j)$, where P_j is the degree $< \kappa$ interpolant of $f^{(i)}|_{S_j}$.*

The theorem does not assume that $f^{(i)}$ is a codeword. Algorithm 4 therefore computes the same next word as the classical fold for every input word, including words sent by a cheating prover, so the verifier and the soundness analysis remain unchanged.

7.3 Cost Analysis

Proposition 5. *Given its precomputed constants, Algorithm 4 performs $n(1 - 1/\kappa) + 2\eta$ field multiplications, η squarings, and $3n(1 - 1/\kappa) + \eta n/\kappa + \eta$ field additions, and no field inversions.*

Proof. Line 10 performs one multiplication and three additions. For each of the n/κ cosets, it is executed $\sum_{s=0}^{\eta-1} 2^{\eta-s-1} = \kappa - 1$ times, and line 8, with one addition, η times. Line 3 performs two multiplications, one squaring, and one addition, and is executed η times. □

The constants depend only on L. The values $\bar{\mathbb{Z}}_s(a_j)$ occupy $\eta n/\kappa$ field elements per round and the in-coset offsets fewer than κ; they are computed with η inversions, $O(\eta m)$ multiplications, and otherwise only additions. For a Cantor basis, $p_{2^s} = 1$ [GM10, Lemma 1], so $\bar{\mathbb{Z}}_s = \mathbb{Z}_s$ and the multiplications by $p_{2^s}^{-1}$ disappear.

For comparison, the Lagrange-based fold of `libiop` [BSCK+19] evaluates

$$P_j(r) = \sum_{v \in S_j} f^{(i)}(v)\, \frac{\mathbb{Z}_{S_j}(r)}{\varepsilon\,(r - v)},$$

where $\mathbb{Z}_{S_j}(x) = \mathbb{Z}_\eta(x) + \mathbb{Z}_\eta(a_j)$ and ε is the linear coefficient of $\mathbb{Z}_\eta$. The denominators $r - v$ depend on the challenge and cannot be precomputed, so `libiop` inverts them by one batch

inversion per coset, at the cost of one field inversion and about 3κ multiplications, together with about $\kappa + \eta$ further multiplications for $\mathbb{Z}_{S_j}(r)$ and the combination. For $\kappa = 2$, with the `libff` inversion over $\mathbb{F}_{2^{256}}$ (15 multiplications and 255 squarings), this amounts to about 24 multiplications and 257 squarings per pair, compared with one multiplication in Algorithm 4. Even a single batch inversion over the whole round would require about $4n$ multiplications.

8 Performance Evaluation

Implementation. We implement our algorithms in the `libiop`-based Aurora prover of [BSG26], which extends `libiop` [BSCK+19] with the Cantor basis and the LCH additive FFT. Our implementation, including the benchmark scripts, is publicly available.[1] All configurations multiply in $\mathbb{F}_{2^{256}}$ with carry-less multiplication instructions. On the standard basis, the Butterfly and IButterfly form their twiddle factors from a single table of the normalized values $\bar{\mathbb{Z}}_r(\beta_j)$, which is computed once and reused by all transforms: every domain of the prover is spanned by a prefix of the same ordered basis, and the shift of L is the next basis element, so every twiddle factor is a subset sum of table entries and is obtained from a previous one with a single addition. On the Cantor basis, we keep the twiddle-factor table used in [BSG26]. The dividends in Aurora are slightly longer than a power of two; for example, the sumcheck polynomial has degree less than $2t + b - 1$. Instead of padding a dividend to the next power of two, which would double the work, our implementation of Algorithm 3 pads it only to a multiple of 2^m and skips the merge steps whose upper half lies entirely in the padding, since these merges add zero. After computing the normalized domain constants once per round, the FRI fold (Algorithm 4) generates its remaining twiddle factors with additions only, since they do not depend on the proof. All configurations in Tables 1 and 3 are built from the same source tree and share the R1CS matrix-vector products and the Merkle-tree, hashing, and Fiat–Shamir code, so they differ only in the evaluation-domain basis, the polynomial representation, the additive FFT, and the FRI fold:

- *Gao–Mateer AFFT* [BSCK+19]: standard basis, monomial coefficients, the Gao–Mateer additive FFT, and the Lagrange-based FRI fold of `libiop`.
- *LCH AFFT* [BSG26]: Cantor basis, monomial coefficients, the LCH additive FFT including its basis-conversion stage, and the Lagrange-based FRI fold of `libiop`.
- *LCH-basis arithmetic + LCH fold (this paper)*: LCH coefficients throughout, the Butterfly and IButterfly without basis conversion, Algorithm 3 for division, the techniques of Sections 5 and 6, and Algorithm 4 for FRI folding, on the standard basis and on the Cantor basis.
- *Baselines + LCH fold (this paper)*: the Gao–Mateer AFFT [BSCK+19] and LCH AFFT [BSG26] configurations with the LCH fold (Algorithm 4) instead of the Lagrange-based fold, which isolate the effect of the LCH-basis arithmetic.

The general-basis columns are evaluated on the standard basis; other bases are supported by the algorithms. The measured times include the computation of the twiddle factors.

Parameters. All experiments use $\mathbb{F}_{2^{256}}$, zero knowledge, RS rate $\rho = 1/32$, BLAKE2b, and the proven soundness setting of `libiop` with a target security parameter of 128 bits. The R1CS instances have a power-of-two number of constraints, from 2^9 to 2^{19}, one fewer variable than constraints, and 31 public inputs. For each size, one random satisfiable

[1] https://github.com/mtbadakhshan/cantor-libiop/tree/lch-computation.

instance and witness are generated and shared by all configurations. Tables 1 and 3 report results for binary folds ($\eta = 1$) in every FRI round.

Measurement. Each sample is the wall-clock time of one complete proof, from protocol initialization through transcript extraction, including all commitments. Instance generation, parameter construction, and verification are excluded. For each size and configuration, we run one warm-up proof followed by 20 measured proofs, rotate the order of the configurations between repetitions, and report the median. The prover runs on a single thread. Stage times are taken from the instrumentation of `libiop` and are inclusive. Experiments run on an AMD Ryzen 9 9950X processor (16 cores) with 64 GB of RAM, running Debian GNU/Linux (trixie/sid) with Linux kernel `6.12.12+bpo-amd64`. We compile with GCC 14.2.0 in GNU C++14 mode, using `-O2`, `-march=native`, `-mtune=native`, `-mpclmul`, and `-msse4.1`.

Proof size and verification. Our algorithms change only how the prover computes the oracle evaluations. The final FRI polynomial is sent in the monomial basis, so for a fixed evaluation basis the proof has the same format and size as in the monomial-basis configuration and is checked by the same verification algorithm. Every benchmarked proof is verified, and our tests also check that proofs generated with LCH-basis arithmetic are accepted by the verifier of the monomial-basis configuration.

8.1 Prover Pipeline Breakdown

Table 3 reports stage times at 2^{19} constraints. Witness oracle construction corresponds to Phase 1 of Section 3.4, and the encoded protocol proof to Phases 2 and 3. The LDT masking polynomial is the masking codeword of Phase 5, committed in the first round. The rowcheck oracle of Phase 4 is virtual and is evaluated within the LDT proof, together with the combined low-degree test of Phase 5. The share of total prover time spent in the additive FFT falls from 57.9% to 31.4% on the general basis, while the LDT proof rises from 36.2% to 50.4% and Merkle tree construction from 4.4% to 16.8%. On the Cantor basis, the LDT proof accounts for 63.6% of the prover time of [BSG26] and 50.1% of ours.

8.2 Application to Fractal

Fractal [COS20] is a preprocessing variant of Aurora: it replaces the lincheck by a holographic lincheck over index polynomials, which an indexer computes once per circuit, and otherwise reuses Aurora's encoded protocol. Our modified `libiop` therefore applies the same changes to the components that Fractal shares with Aurora: oracle construction with division by $\mathbb{Z}_S$ and blinding (Sections 4 and 5), the sumcheck over H (Section 6), the rowcheck, and the FRI fold (Section 7). Apart from its sumcheck over H, the holographic lincheck involves no division or multiplication by vanishing polynomials, so it benefits only from transforms without basis conversion. The one-time preprocessing of the indexer is excluded from the prover time. At 2^{18} constraints, our prover is 2.88× faster than `libiop`, and 2.30× faster when `libiop` also uses our FRI fold. These results are on the general basis, since the Fractal prover of `libiop` supports only standard-basis domains.

8.3 Discussion

On the general basis, the LCH-basis arithmetic gives the largest reduction in oracle construction and in the lincheck and sumcheck (Phases 1–3); on the Cantor basis, where the baseline transforms are already faster, the largest reduction is in FRI folding. These phases consist of interpolations, evaluations, divisions, and multiplications by vanishing polynomials, and our algorithms remove the basis conversions from all of them. On the general basis, removing the basis conversion reduces the field multiplications of a transform

Table 3: Aurora prover subcomponent times in seconds at 2^{19} constraints and $\eta = 1$. Entries are medians. Subcomponent timings are inclusive: FRI arithmetic is included in the LDT stage, and cross-cutting totals overlap protocol stages. For Algorithm 4, the folding loop includes the computation of its twiddle factors. Commitments for rounds 1–2 include transcript updates and are reconstructed from per-proof timers; subsequent commitments are included in the LDT stage. A dash denotes an inapplicable component.

Component	General basis		Cantor basis	
	Gao–Mateer AFFT [BSCK+19]	LCH-basis arithmetic + LCH fold (this paper)	LCH AFFT [BSG26]	LCH-basis arithmetic + LCH fold (this paper)
Protocol stages				
Initialization	0.021	0.026	0.021	0.028
Witness oracles	224.112	35.473	59.597	36.267
LDT masking	44.680	7.749	11.986	7.839
Encoded protocol	185.919	29.067	53.385	29.117
Round 1–2 commitments	26.235	26.778	26.601	26.915
LDT proof (incl. FRI)	280.091	107.449	279.470	107.264
Query responses	7.190	5.100	7.187	5.094
FRI arithmetic (included in the LDT proof)				
Folding loop	174.014	1.297	173.893	1.292
Cross-cutting totals				
Merkle trees	34.046	35.713	34.513	35.771
Additive FFT	447.795	66.901†	118.489	67.509†
Additive IFFT	1.921	0.461†	0.577	0.467†
Total prover	**773.369**	**213.070**	**439.412**	**213.945**

†Our LCH-basis arithmetic eliminates the monomial-basis conversion stage of the additive FFT/IFFT, leaving only Butterfly/IButterfly.

on L by a factor of about 3 and its field additions by a factor of about 7 at $|L| = 2^{26}$ (Section 2); the measured reduction of the additive FFT, 6.7×, lies between the two. On the Cantor basis, the conversion consists of additions only, and removing it reduces the additive FFT time by 1.8×. FRI folding gains from Algorithm 4 relative to the Lagrange-based fold of `libiop`, which inverts challenge-dependent values on every coset: at 2^{19} constraints, the folding loop takes 1.3 s instead of 174.0 s.

With the LCH-basis arithmetic, the prover takes nearly the same time on both bases: the two columns of Table 1 differ by less than 2% at every size, within the run-to-run variation. In the baselines, a Cantor basis is faster because the basis-conversion stage of the additive FFT needs no multiplications on that basis [BSG26]; our transforms omit this stage on every basis, and the remaining butterfly performs $(n/2)\log_2 n$ multiplications for any basis. On a Cantor basis, division by $\mathbb{Z}_m$ also needs no multiplications (Table 2), but this saves at most one multiplication per coefficient, and the two divisions of a proof process about $|H|$ and $2|H|$ coefficients, whereas each transform on L runs over $|L| = 128|H|$ points ($|L| = 2^{26}$ at 2^{19} constraints). Blinding needs no multiplications on either basis, since the scaling is absorbed into coefficient sampling (Section 5). Conversely, the Cantor-basis butterfly that we reuse from [BSG26] forms each twiddle factor from a precomputed table of the spans of four blocks of eight basis elements, using up to three field additions; in our implementation, the general-basis butterfly derives each twiddle factor from a previously computed one with a single addition.

After these changes, the LDT proof and Merkle hashing dominate the prover time; Merkle hashing is identical across configurations. With our fold, the LDT proof takes about 107 s on both bases, of which folding takes 1.3 s and the FRI commitments about 9 s. Most of the remainder forms the first FRI codeword: the virtual oracles are evaluated

on L, and `libiop` computes the degree adjustments of the batch by raising every point of L to the required powers.

9 Conclusions and Future Work

We presented a divide-and-conquer algorithm for polynomial division by subspace vanishing polynomials that operates directly in the LCH polynomial basis, for arbitrary ordered $\mathbb{F}_2$-bases. In the LCH basis, multiplying a zero-knowledge blinding polynomial by a vanishing polynomial reduces to sampling LCH coefficients. With these primitives, the polynomials of the Aurora prover remain in the LCH basis, and every additive FFT and IFFT reduces to the $O(n \log n)$ butterfly, without the basis-conversion stage. We also expressed the FRI fold in the LCH basis: on each coset, it combines an LCH inverse butterfly with an evaluation at the verifier challenge, requires no field inversions, and computes the classical fold on every input word. Only the final FRI polynomial is sent in the monomial basis, so the proof and the verifier are unchanged.

In Aurora over $\mathbb{F}_{2^{256}}$ at 2^{19} constraints with binary folds, and when all configurations use our FRI fold, the LCH-basis arithmetic speeds up the prover by 2.80× over `libiop` on a general basis and by 1.24× over the Cantor-basis configuration of [BSG26]. Against these baselines with their Lagrange-based fold, the speedups are 3.63× and 2.05×. With our arithmetic, the prover is equally fast on general and Cantor bases, so the evaluation basis can be chosen freely without a performance penalty. The algorithms apply wherever univariate-polynomial IOP provers over binary extension fields use classical additive FRI folds and division or multiplication by subspace vanishing polynomials, including STARKs, Fractal, and Polaris. Our modified `libiop` also runs Fractal with these changes, with a speedup of 2.88× over `libiop` on a general basis.

Three directions follow from this work. First, SLH-DSA signatures rely on SHA-2 and SHAKE, which favor binary-field arithmetizations; implementing SNARK-based aggregation of these signatures in Aurora or Fractal would measure the benefit of our arithmetic for this application. Second, eliminating the basis-conversion phase leaves a butterfly structure similar to that of the Cooley–Tukey FFT [CT65], which may allow the GPU parallelization techniques of multiplicative-FFT zkSNARK provers [LWY+23, MXS+23] to be applied to the additive setting. Third, the holographic lincheck of Fractal could be redesigned around the LCH basis, and Fractal could be extended to Cantor-basis domains.

Acknowledgments

The authors used Claude Opus 5.5 (Anthropic) and GPT-6.1 Sol (OpenAI) to edit the text and to assist with parts of the implementation and benchmark scripts. The authors reviewed all generated content and take full responsibility for it.

A Proofs

A.1 Proof of Correctness for Algorithm 1

Proof. Fix $m \geq 0$. We proceed by induction on the recursion depth $d \geq 0$. Consider a polynomial

$$A(x) = \sum_{i=0}^{2^{d+m+1}-1} a_i^{\mathbb{X}} X_i(x) \in \mathbb{F}_{2^k}[x],$$

with $\deg(A) < 2^{d+m+1}$, expressed in the LCH alternative basis. We must show that the Algorithm 1 returns polynomials $Q(x)$ and $R(x)$ such that

$$A(x) = \mathbb{Z}_m(x) \cdot Q(x) + R(x), \qquad \deg(R) < 2^m, \quad \deg(Q) < 2^{d+m+1} - 2^m.$$

Base case $d = 0$. When $d = 0$, the algorithm sets

$$R(x) = \sum_{i=0}^{2^m-1} a_i^{\mathbb{X}} X_i(x), \qquad Q(x) = \sum_{i=0}^{2^m-1} a_{i+2^m}^{\mathbb{X}} X_i(x),$$

and returns $(Q(x), R(x))$. By Lemma 1, this is exactly the quotient and remainder of dividing $A(x)$ by $\mathbb{Z}_m(x)$, and we have $A(x) = \mathbb{Z}_m(x)\, Q(x) + R(x)$ with $\deg(R) < 2^m$ and $\deg(Q) < 2^{m+1} - 2^m = 2^m$.

Induction step. Assume that the claim holds for some integer $d - 1 \geq 0$. We will show that it also holds for d. The algorithm decomposes $A(x)$ into two halves:

$$A_0(x) = \sum_{i=0}^{2^{m+d}-1} a_i^{\mathbb{X}} X_i(x), \qquad \text{and} \qquad A_1(x) = \sum_{i=0}^{2^{m+d}-1} a_{i+2^{m+d}}^{\mathbb{X}} X_i(x).$$

By the structure of the LCH alternative basis, this decomposition satisfies

$$A(x) = A_0(x) + A_1(x)\, X_{2^{m+d}}(x).$$

By the induction hypothesis, the two recursive calls correctly divide $A_0(x)$ and $A_1(x)$ by $\mathbb{Z}_m(x)$, yielding:

$$A_0(x) = \mathbb{Z}_m(x)\, Q_0(x) + R_0(x), \quad \text{where} \quad \deg(R_0) < 2^m,\ \deg(Q_0) < 2^{d+m} - 2^m$$

$$A_1(x) = \mathbb{Z}_m(x)\, Q_1(x) + R_1(x), \quad \text{where} \quad \deg(R_1) < 2^m,\ \deg(Q_1) < 2^{d+m} - 2^m.$$

The algorithm sets $R(x) = R_0(x)$, so the required bound $\deg(R) < 2^m$ is immediate.

Next, define the expansion polynomial as $D_{d,m}(x) := \sum_{i=0}^{2^d-1} p_{2^m(2^d-i-1)} X_{2^m i}(x)$, where p_j are defined in (2). Since $W_m \subseteq W_{m+d}$, the vanishing polynomial $\mathbb{Z}_m(x)$ divides $\mathbb{Z}_{m+d}(x) = X_{2^{m+d}}(x)$. The explicit quotient is determined iteratively by the recurrence relation $\mathbb{Z}_{j+1}(x) = \mathbb{Z}_j(x)^2 + p_{2^j}\mathbb{Z}_j(x)$ for $j = m, \ldots, m+d-1$ and in the LCH alternative basis the quotient is precisely $D_{d,m}(x)$. Thus,

$$X_{2^{m+d}}(x) = \mathbb{Z}_m(x)\, D_{d,m}(x). \tag{19}$$

For the quotient, the algorithm defines

$$Q(x) = Q_0(x) + R_1(x)\, D_{d,m}(x) + Q_1(x)\, X_{2^{m+d}}(x). \tag{20}$$

We have $\deg(D_{d,m}) = 2^{m+d} - 2^m$ because $\deg(X_{2^{m+d}}) = 2^{m+d}$ and $\deg(\mathbb{Z}_m) = 2^m$. Therefore, we have

$$\deg\big(R_1 D_{d,m}\big) \leq (2^m - 1) + (2^{m+d} - 2^m) = 2^{m+d} - 1,$$

$$\deg\big(Q_1 X_{2^{m+d}}\big) < (2^{m+d} - 2^m) + 2^{m+d} = 2^{m+d+1} - 2^m,$$

which implies that $\deg(Q) < 2^{d+m+1} - 2^m$.

It remains to verify the division identity. Using the decompositions of A_0 and A_1, we obtain

$$\begin{aligned}
A(x) &= A_0(x) + A_1(x)\, X_{2^{m+d}}(x) \\
&= \big(\mathbb{Z}_m(x)\, Q_0(x) + R_0(x)\big) + \big(\mathbb{Z}_m(x)\, Q_1(x) + R_1(x)\big) X_{2^{m+d}}(x) \\
&= R_0(x) + \mathbb{Z}_m(x)\, Q_0(x) + \mathbb{Z}_m(x)\, Q_1(x) X_{2^{m+d}}(x) + R_1(x) X_{2^{m+d}}(x) \\
&= R(x) + \mathbb{Z}_m(x)\, Q_0(x) + \mathbb{Z}_m(x)\, Q_1(x) X_{2^{m+d}}(x) + R_1(x)\, \mathbb{Z}_m(x)\, D_{d,m}(x) \quad [\text{by (19)}] \\
&= R(x) + \mathbb{Z}_m(x)\, [Q_0(x) + Q_1(x) X_{2^{m+d}}(x) + R_1(x) D_{d,m}(x)] \\
&= R(x) + \mathbb{Z}_m(x)\, Q(x) \quad [\text{by (20)}].
\end{aligned}$$

Thus, for every $d \geq 0$, Algorithm 1 returns polynomials $Q(x)$ and $R(x)$ such that $A(x) = \mathbb{Z}_m(x)Q(x) + R(x)$ with $\deg(R) < 2^m$ and $\deg(Q) < 2^{d+m+1} - 2^m$. □

A.2 Equivalence of Algorithms 1 and 2

Proof. We prove the claim by induction on the outer-loop stage $s = 1, \ldots, d$ of Algorithm 2. Let $\vec{\mathbf{q}}^{(s)}$ denote the state of $\vec{\mathbf{q}}$ after stage s, and $\vec{\mathbf{q}}^{(0)}$ its initial state.

Invariant $\mathcal{I}(s)$. After stage s, each block group indexed by $v \in \{0, \ldots, 2^{d-s} - 1\}$ stores, in $\vec{\mathbf{q}}^{(s)}$, exactly the quotient coefficients obtained by applying s recursive merge levels of Algorithm 1 to the corresponding recursion subtree.

Base case ($s = 0$). Before the loop, $\vec{\mathbf{q}}^{(0)}$ is the direct coefficient layout used by the recursive algorithm before any merge step; therefore $\mathcal{I}(0)$ holds.

Induction step. Assume $\mathcal{I}(s-1)$ for some $1 \leq s \leq d$. Fix a group v. In the recursive combine step at depth s, one adds the term $R_1(x)D_{s,m}(x)$ to $Q_0(x)$, where

$$D_{s,m}(x) = \sum_{u=0}^{2^s-1} p_{2^m(2^s-u-1)} X_{2^m u}(x).$$

In Algorithm 2, for each $u = 0, \ldots, 2^s - 2$ and local coefficient index i, the update

$$\vec{\mathbf{q}}[w2^m + i] \leftarrow \vec{\mathbf{q}}[w2^m + i] + \vec{\mathbf{q}}[t2^m + i] \cdot p_{2^m(2^s-u-1)},$$

with $t = v2^{s+1} + 2^s - 1$ and $w = v2^{s+1} + u$ adds to target block w exactly the contribution of the source block t scaled by the corresponding coefficient of $D_{s,m}(x)$. Thus the iterative updates reproduce the same linear combination as the recursive merge formula

$$Q = Q_0 + R_1 D_{s,m} + Q_1 X_{2^{m+s}}.$$

The source block t still holds R_1, since earlier stages write only the lower blocks of their own groups. The last term of $D_{s,m}$, $u = 2^s - 1$, has coefficient $p_0 = 1$ and falls on block t itself, where R_1 is already stored, so the loop ends at $u = 2^s - 2$. The term $Q_1 X_{2^{m+s}}$ consists of the blocks of Q_1, which are already in place. Hence, each group after the stage s matches the recursive result after one more merge level, so $\mathcal{I}(s)$ holds.

By induction, $\mathcal{I}(d)$ holds. Therefore, after the last stage, the vector $\vec{\mathbf{q}}$ equals the quotient produced by Algorithm 1. Also, $\vec{\mathbf{r}}$ is never modified, so it equals the same remainder block used in the recursive algorithm. Hence, both algorithms return identical $(Q(x), R(x))$. □

A.3 Proof of Correctness of Algorithm 3

We first establish two identities specific to the LCH basis, then prove the algorithm correct by induction.

Lemma 3 (LCH-basis quotient by $\mathbb{Z}_m$)**.** *Let $A(x) = \sum_{i=0}^{2^{m+1}-1} a_i^{\bar{\mathbb{X}}} \bar{X}_i(x)$ be a polynomial in LCH basis with $\deg(A) < 2^{m+1}$. Then*

$$A(x) = \mathbb{Z}_m(x) \cdot Q(x) + R(x),$$

where $R(x) = \sum_{i=0}^{2^m-1} a_i^{\bar{\mathbb{X}}} \bar{X}_i(x)$ and $Q(x) = \frac{1}{p_{2^m}} \sum_{i=0}^{2^m-1} a_{i+2^m}^{\bar{\mathbb{X}}} \bar{X}_i(x)$.

Proof. For $0 \le i < 2^m$, the indices i and 2^m have disjoint bit sets, so $\bar{X}_{2^m+i} = \bar{X}_{2^m} \cdot \bar{X}_i$. Since $\bar{X}_{2^m} = X_{2^m}/p_{2^m} = \mathbb{Z}_m(x)/p_{2^m}$, we obtain

$$\sum_{i=0}^{2^m-1} a_{i+2^m}^{\bar{\mathbb{X}}} \bar{X}_{2^m+i} = \frac{\mathbb{Z}_m(x)}{p_{2^m}} \sum_{i=0}^{2^m-1} a_{i+2^m}^{\bar{\mathbb{X}}} \bar{X}_i(x) = \mathbb{Z}_m(x)\, Q(x). \qquad \square$$

Lemma 4 (Uniform LCH coefficients of $D_{s,m}$)**.** *For $D_{s,m}(x) = \sum_{j=0}^{2^s-1} p_{2^m(2^s-j-1)}\, X_{2^m j}(x)$, we have*

$$D_{s,m}(x) = p_{2^m(2^s-1)} \sum_{j=0}^{2^s-1} \bar{X}_{2^m j}(x). \tag{21}$$

That is, all LCH-basis coefficients of $D_{s,m}$ at the block level are the same constant $p_{2^m(2^s-1)}$.

Proof. Rewriting $X_{2^m j} = p_{2^m j}\, \bar{X}_{2^m j}$ gives $D_{s,m} = \sum_j p_{2^m(2^s-j-1)}\, p_{2^m j}\, \bar{X}_{2^m j}$. The indices $2^m(2^s - j - 1)$ and $2^m j$ occupy positions $m, \dots, m+s-1$ with complementary bits (since $(2^s - j - 1) + j = 2^s - 1$), so their bit sets are disjoint and $p_{2^m(2^s-j-1)} \cdot p_{2^m j} = p_{2^m(2^s-1)}$ for every j. The constant factors out, giving (21). $\square$

A.3.1 Proof of Theorem 2

Proof. Write $\vec{\mathbf{a}}_j$ for the j-th block of 2^m input coefficients, so that block t of $\vec{\mathbf{q}}$ initially holds $\vec{\mathbf{a}}_{t+1}$. For an integer x and $0 \le s \le d$, let $Z_s(x) \subseteq \{0, \dots, s\}$ be the set of positions of the zero bits of x among its bits $0, \dots, s$. We first describe the quotient produced by the recursion of Algorithm 1 in the LCH basis, and then show that Algorithm 3 computes it.
Merge factors. At each stage $s \ge 1$, the recursive combine formula in the LCH basis is

$$Q = Q_0 + \frac{D_{s,m}}{p_{2^{m+s}}}\, R_1 + \bar{X}_{2^{m+s}}\, Q_1,$$

where $\bar{X}_{2^{m+s}} = X_{2^{m+s}}/p_{2^{m+s}}$ and $X_{2^{m+s}} = \mathbb{Z}_m(x)\, D_{s,m}(x)$ by (19). By Lemma 4, the factor multiplying R_1 is

$$\frac{D_{s,m}(x)}{p_{2^{m+s}}} = \frac{p_{2^m(2^s-1)}}{p_{2^{m+s}}} \sum_{j=0}^{2^s-1} \bar{X}_{2^m j}(x),$$

so its contribution to every target block is the *same* scalar $\mu_s := p_{2^m(2^s-1)}/p_{2^{m+s}}$ times R_1, which equals the scalar γ of stage s in Algorithm 3. For $s = 0$, Lemma 3 gives the base-case quotient as the upper block scaled by $\mu_0 = 1/p_{2^m}$.
Quotient blocks. We show by induction on s that, for every group of 2^{s+1} consecutive input blocks starting at a multiple of 2^{s+1}, block w of the quotient of the group by $\mathbb{Z}_m(x)$, for $0 \le w \le 2^{s+1} - 2$ relative to the group, is

$$\sum_{s' \in Z_s(w)} \mu_{s'}\, \vec{\mathbf{a}}_{(w \,\mathrm{OR}\, (2^{s'}-1))+1}, \tag{22}$$

with input indices relative to the group. For $s = 0$, the quotient is $\mu_0 \vec{\mathbf{a}}_1$ and $Z_0(0) = \{0\}$. For $s \ge 1$, the remainder R_1 of the upper half is its lowest block $\vec{\mathbf{a}}_{2^s}$, the merge adds $\mu_s \vec{\mathbf{a}}_{2^s}$ to blocks $0, \dots, 2^s - 1$, and $\bar{X}_{2^{m+s}} Q_1$ places Q_1 at blocks $2^s, \dots, 2^{s+1} - 2$. If $w < 2^s - 1$, block w is block w of Q_0 plus $\mu_s \vec{\mathbf{a}}_{2^s}$; bit s of w is zero, $Z_s(w) = Z_{s-1}(w) \cup \{s\}$, and $(w \,\mathrm{OR}\, (2^s - 1)) + 1 = 2^s$. If $w = 2^s - 1$, block w is $\mu_s \vec{\mathbf{a}}_{2^s}$ and $Z_s(w) = \{s\}$. If $w \ge 2^s$, block w is block $w - 2^s$ of Q_1; bit s of w is one, $Z_s(w) = Z_{s-1}(w - 2^s)$, and the indices of the upper half are shifted by 2^s. In all three cases, (22) follows from the induction hypothesis.

Algorithm 3. Each block t is processed exactly once, at the stage s equal to the number of trailing ones of t, where $t = v \cdot 2^{s+1} + 2^s - 1$; it is not modified at any other stage. Let $w = t + 2^s$. The smallest element of $Z_d(t)$ is s, and its term in (22) is $\mu_s \vec{\mathbf{a}}_{t+1}$. Since w is t with bit s set, $Z_d(w) = Z_d(t) \setminus \{s\}$, and every $s' \in Z_d(w)$ satisfies $s' > s$ and $w \,\mathrm{OR}\, (2^{s'} - 1) = t \,\mathrm{OR}\, (2^{s'} - 1)$. Hence block t of the quotient equals $\mu_s \vec{\mathbf{a}}_{t+1}$ plus block w of the quotient, where block w is zero if $w = 2^{d+1} - 1$. At stage s, the algorithm scales block t by $\gamma = \mu_s$ and adds block w. Block w is processed at a stage greater than s, since its number of trailing ones exceeds s; because the stages run in decreasing order, block w already holds its final value. By induction over the stages in decreasing order, every block holds its quotient block after its stage.

Remainder. The vector $\vec{\mathbf{r}}$ is never modified. After the last stage, $A(x) = \mathbb{Z}_m(x)\, Q(x) + R(x)$, where $R(x) = \sum_{i<2^m} a_i^{\mathbb{X}} \bar{X}_i(x)$ has degree less than $2^m = \deg \mathbb{Z}_m$. Since the quotient and remainder of a division are unique, $Q(x)$ and $R(x)$ are the quotient and remainder of $A(x)$ by $\mathbb{Z}_m(x)$, with the stated degree bounds. □

A.4 Proof of Correctness of Algorithm 4

Proof of Theorem 3. Fix a coset S_j, and let $g^{(s)}$ be defined by (18) with $g^{(0)} = f^{(i)}|_{S_j}$. At stage s, line 10 of Algorithm 4 reads $g^{(s)}$ at the pair $\{u, u + p_{2^s}\}$ with $u = \mathbb{Z}_s(v_{j,s,t})$, using $\lambda_s + c_{j,s} + \delta_{s,t} = \bar{\mathbb{Z}}_s(r) + \bar{\mathbb{Z}}_s(v_{j,s,t})$, and stores $g^{(s+1)}$ at $\mathbb{Z}_{s+1}(v_{j,s,t})$. It therefore suffices to show that $g^{(\eta)}(y_j) = P_j(r)$.

We prove by induction on s that $g^{(s)}$ is the restriction to $S_j^{(s)}$ of a polynomial $P^{(s)}$ of degree less than $2^{\eta-s}$ with $P^{(s)}\big(\mathbb{Z}_s(r)\big) = P_j(r)$. For $s = 0$, we take $P^{(0)} = P_j$.

For the induction step, let $h = 2^{\eta-s-1}$ and $\xi = \mathbb{Z}_s(r)$. Since q_s is monic of degree two, repeated division of $P^{(s)}$ by q_s gives the decomposition (17) of $P^{(s)}$, with $\deg(A), \deg(B) < h$. Let $v \in S_j$, $u = \mathbb{Z}_s(v)$, and $z = q_s(u) = \mathbb{Z}_{s+1}(v)$. Since $q_s(u + p_{2^s}) = z$ and $u/p_{2^s} = \bar{\mathbb{Z}}_s(v)$,

$$g^{(s)}(u) = A(z) + \bar{\mathbb{Z}}_s(v)\, B(z), \qquad g^{(s)}(u) + g^{(s)}(u + p_{2^s}) = B(z).$$

Substituting into (18) gives $g^{(s+1)}(z) = A(z) + \bar{\mathbb{Z}}_s(r)\, B(z)$. Hence $g^{(s+1)}$ is the restriction of $P^{(s+1)} = A + \bar{\mathbb{Z}}_s(r)\, B$, which has degree less than h, and

$$P^{(s+1)}\big(\mathbb{Z}_{s+1}(r)\big) = A\big(q_s(\xi)\big) + \frac{\xi}{p_{2^s}}\, B\big(q_s(\xi)\big) = P^{(s)}(\xi) = P_j(r).$$

For $s = \eta$, the polynomial $P^{(\eta)}$ is a constant. Hence $g^{(\eta)}(y_j) = P^{(\eta)}\big(\mathbb{Z}_\eta(r)\big) = P_j(r)$. The argument uses only the values of $f^{(i)}$ on S_j, so it holds for every word. □

References

[ACFY24] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. STIR: Reed–Solomon proximity testing with fewer queries. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 380–413, Cham, 2024. Springer Nature Switzerland.

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. WHIR: Reed–Solomon proximity testing with super-fast verification. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025*, pages 214–243, Cham, 2025. Springer Nature Switzerland.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup.

In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[ark21] arkworks contributors. `arkworks` zkSNARK ecosystem. https://arkworks.rs, 2021. Source code available at https://github.com/arkworks-rs. Accessed: 2026-04-10.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, New York, NY, USA, 2012. Association for Computing Machinery.

[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 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 103–128. Springer, 2019.

[BG18] Sean Bowe and Jack Grigg. bellman: a Rust library for zk-SNARKs. https://github.com/zkcrypto/bellman, 2018. Full list of contributors available at https://github.com/zkcrypto/bellman/graphs/contributors. Accessed: 2026-04-10.

[BG25] Mohammadtaghi Badakhshan and Guang Gong. Evaluating the integration of Aurora zkSNARK in the Zupply framework. In *2025 IEEE/ACM 6th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pages 23–30, 2025.

[BGP20] Gautam Botrel, Gus Gutoski, and Thomas Piellard. gnark: a fast zk-SNARK library in Go. https://github.com/Consensys/gnark, 2020. Full list of contributors available at https://github.com/Consensys/gnark/graphs/contributors. Accessed: 2026-04-10.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ signature framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery.

[Bin] Binius64 contributors. Binius64. https://github.com/IrreducibleOSS/binius64. FRI evaluation-domain construction in `crates/math/src/ntt/domain_context.rs`; accessed 2026-10-05.

[BLGH21] Sean Bowe, Ying Tong Lai, Jack Grigg, and Daira Hopwood. halo2: a Rust implementation of the Halo2 zero-knowledge proving system. https://github.com/zcash/halo2, 2021. Full list of contributors available at https://github.com/zcash/halo2/graphs/contributors. Accessed: 2026-04-10.

[BP18] Jordi Baylina and Xavi Pinsach. snarkjs: a JavaScript and WebAssembly implementation of zkSNARK and PLONK schemes. https://github.com/iden3/snarkjs, 2018. Full list of contributors available at https://github.com/iden3/snarkjs/graphs/contributors. Accessed: 2026-04-10.

[BSBHR18a] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed–Solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BSBHR18b] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Shaul Kfir, Eran Tromer, Madars Virza, and Howard Wu. libsnark: a C++ library for zkSNARKs. https://github.com/scipr-lab/libsnark, 2014. Full list of contributors available at https://github.com/scipr-lab/libsnark/blob/master/AUTHORS. Accessed: 2026-04-10.

[BSCK+19] Eli Ben-Sasson, Alessandro Chiesa, Alex Kazorian, Dev Ojha, Aleksejs Popovs, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas Ward. libiop: A C++ library for zero-knowledge proofs. https://github.com/scipr-lab/libiop, 2019. Accessed: 2025-01-10.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings, Part II, of the 14th International Conference on Theory of Cryptography - Volume 9986*, pages 31–60, Berlin, Heidelberg, 2016. Springer-Verlag.

[BSG26] Mohammadtaghi Badakhshan, Susanta Samanta, and Guang Gong. Accelerating post-quantum secure zkSNARKs by optimizing additive FFT. In Christina Boura, Atefeh Mashatan, and Ali Miri, editors, *Selected Areas in Cryptography – SAC 2025*, pages 339–368, Cham, 2026. Springer Nature Switzerland.

[BWL22] William Borgeaud, Nicholas Ward, and Daniel Lubarov. Plonky2: a fast recursive SNARK based on PLONK and FRI. https://github.com/0xPolygonZero/plonky2, 2022. Full list of contributors available at https://github.com/0xPolygonZero/plonky2/graphs/contributors. Accessed: 2026-04-10.

[Can89] David G. Cantor. On arithmetical algorithms over finite fields. *J. Comb. Theory A*, 50(2):285–300, 1989.

[CCC+23] Ming-Shing Chen, Yu-Shian Chen, Chen-Mou Cheng, Shiuan Fu, Wei-Chih Hong, Jen-Hsuan Hsiang, Sheng-Te Hu, Po-Chun Kuo, Wei-Bin Lee, Feng-Hao Liu, and Justin Thaler. Preon: zk-SNARK based signature scheme. NIST PQC Additional Digital Signatures, Round 1 specification, 2023.

[Che25] Ming-Shing Chen. polyeval: Polynomial evaluation with additive FFT. https://github.com/devillegna/polyeval, 2025. Accessed: 2026-10-09.

[CJL+24] Kostas Kryptos Chalkias, Charanjit Jutla, Jonas Lindstrom, Varun Madathil, and Arnab Roy. Improved polynomial division in cryptography. Cryptology ePrint Archive, Paper 2024/1279, 2024.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 769–793, Cham, 2020. Springer International Publishing.

[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.

[DP25] Benjamin E. Diamond and Jim Posen. Succinct Arguments over Towers of Binary Fields. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025*, pages 93–122, Cham, 2025. Springer Nature Switzerland.

[DP26] Benjamin E. Diamond and Jim Posen. Polylogarithmic Proofs for Multilinears over Binary Towers. In *Advances in Cryptology – EUROCRYPT 2026*, volume 16547 of *Lecture Notes in Computer Science*, pages 3–32. Springer Nature Switzerland, 2026.

[Fas26] Matteo Frigo and abhi shelat. Anonymous credentials from ECDSA. *IACR Communications in Cryptology*, 3(1), 2026.

[FG22] Shihui Fu and Guang Gong. Polaris: Transparent succinct zero-knowledge arguments for R1CS with efficient verifier. *Proceedings on Privacy Enhancing Technologies*, 2022(1):544–564, 2022.

[GHR+23] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. Horst meets Fluid-SPN: Griffin for zero-knowledge applications. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part III*, pages 573–606, Berlin, Heidelberg, 2023. Springer-Verlag.

[GM10] Shuhong Gao and Todd D. Mateer. Additive fast Fourier transforms over finite fields. *IEEE Trans. Inf. Theory*, 56(12):6265–6272, 2010.

[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*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[HLP24] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, Paper 2024/278, 2024.

[LAH16] Sian-Jheng Lin, Tareq Y. Al-Naffouri, and Yunghsiang S. Han. FFT algorithm for binary extension finite fields and its application to Reed–Solomon codes. *IEEE Trans. Inf. Theory*, 62(10):5343–5358, 2016.

[LAHC16] Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yunghsiang S. Han, and Wei-Ho Chung. Novel polynomial basis with fast Fourier transform and its application to Reed–Solomon erasure codes. *IEEE Transactions on Information Theory*, 62(11):6284–6299, 2016.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to Reed–Solomon erasure codes. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 316–325. IEEE Computer Society, 2014.

[LWY+23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):194–220, 2023.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MXS+23] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. GZKP: A GPU accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 340–353. ACM, 2023.

[NIS24] NIST. FIPS 205: Stateless hash-based digital signature standard. Federal Information Processing Standards Publication (FIPS) 205, National Institute of Standards and Technology, August 2024.

[Plo] Plonky3 contributors. Plonky3: Polynomial IOPs for STARKs. https://github.com/Plonky3/Plonky3. FRI implementation, `fri/src/two_adic_pcs.rs`, commit `4e3eaf450039`; accessed 2026-10-05.

[RS60] Irving Stoy Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

[Sta24] StarkWare. Stwo: A STARK prover. https://github.com/starkware-libs/stwo, 2024. Accessed: 2026-10-03.

[ZCF24] Hadas Zeilberger, Binyi Chen, and Ben Fisch. BaseFold: Efficient field-agnostic polynomial commitment schemes from foldable codes. In *Advances in Cryptology – CRYPTO 2024*, pages 138–169. Springer, 2024.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *Symbolic and Algebraic Computation*, pages 216–226, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

[ZSE+24] Xinyu Zhang, Ron Steinfeld, Muhammed F. Esgin, Joseph K. Liu, Dongxi Liu, and Sushmita Ruj. Loquat: A SNARK-friendly post-quantum signature based on the Legendre PRF with applications in ring and aggregate signatures. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 3–38, Cham, 2024. Springer Nature Switzerland.