Accelerating Post-quantum Secure zkSNARKs by Optimizing Additive FFT

Mohammadtaghi Badakhshan, Susanta Samanta, and Guang Gong

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, ON, Canada
{mbadakhshan,ssamanta,ggong}@uwaterloo.ca

Abstract. Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zkSNARKs) are increasingly utilized across diverse applications. While significant advances have been made in the development of post-quantum secure zkSNARKs, these schemes face challenges, including substantial computational complexity. In this paper, we propose leveraging the Cantor special basis in post-quantum secure zkSNARKs operating over binary extension fields. This approach enables the optimization of the additive Fast Fourier Transform (FFT) algorithm in Aurora, a post-quantum secure zkSNARK, by replacing the previously used Gao–Mateer FFT with the Cantor and LCH FFTs. Our implementation demonstrates a significant reduction in computation time for Aurora, with the potential to accelerate other zkSNARKs utilizing additive FFTs. Additionally, we present a detailed theoretical analysis of the computational costs of the Cantor FFT algorithm, providing exact counts of additions, multiplications, and precomputation overhead. Furthermore, we analyze the FFT call complexity within the encoding of the Rank-1 Constraint System in the Aurora zkSNARK.

Keywords: Post-quantum secure zkSNARK · R1CS · Aurora · Additive FFT · Cantor Algorithm · Gao–Mateer Algorithm

1 Introduction

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) protocols are cryptographic schemes involving a prover and a verifier. The prover generates a publicly verifiable and succinct proof that demonstrates knowledge of a witness vector (secret inputs) satisfying a given constraint system. The proof does not reveal any information about the witness itself. These protocols have a wide range of applications, from post-quantum secure digital signature algorithms [2,17,18,32] to privacy preserving applications over blockchains [8,14,43,47], and blockchain scalability solutions using rollups [16,44]. The efficiency of the prover and verifier algorithms, along with the proof size of the zkSNARK protocol, are crucial factors influencing the cost-effectiveness and overall practicality of the aforementioned applications. Additionally, two other important factors are: whether the zkSNARK protocol relies on a trusted setup

or employs a transparent setup, and, whether it offers plausible post-quantum security against a malicious prover with quantum capabilities.

Optimizing or accelerating the implementation of algorithms in zkSNARKs is an active and popular area of research [5,23,40,6,31]. In parallel, optimizing the *Fast Fourier Transform (FFT)* over additive groups has been a significant focus of study [15,27,26,37,36]. The FFT over additive groups can be utilized in various zkSNARK protocols operating over binary extension fields. Examples of such protocols include Ligero [1], STARK [4], Aurora [10], Fractal [21], and Polaris [25].

The FFT over additive groups, known as the *additive FFT*, over finite fields was developed in the late 1980s. Wang and Zhu [46] first introduced this concept, followed independently by Cantor [15]. These algorithms evaluate polynomials of degree less than $n = 2^m$ over m-dimensional *affine subspaces* of $\mathbb{F}_{2^k}$, where $k = 2^\ell$. This was later generalized to arbitrary k by von zur Gathen and Gerhard [27], but this incurred a higher computational cost. Subsequently, Gao and Mateer [26] proposed two algorithms based on Taylor expansions: one applicable to arbitrary k, with lower complexity than the method of von zur Gathen and Gerhard, and another optimized for additive *FFTs of length* 2^m, where m is a power of two, reducing the number of additions while matching Cantor's multiplication count.

Lin et al. [38] later introduced the LCH basis, constructed from the *vanishing polynomials*, enabling FFTs with $O(n \log n)$ additions and multiplications. They also developed conversion algorithms [37] between the LCH and monomial bases, requiring $O(n \log n \log \log n)$ additions and no multiplications under the *Cantor special basis*, with further refinements in [22]. These methods have been applied to fast binary polynomial multiplication [19,20,35]. Since additive FFTs require input lengths that are powers of two, a polynomial with $(t + 1)$ coefficients (i.e., of degree t) must be zero-padded to length 2^m if $t + 1 < 2^m$. Bernstein et al. [13] addressed this by modifying the Gao–Mateer algorithm to skip operations on known-zero coefficients. Further improvements were made by Bernstein and Chou [12], incorporating the Cantor special basis and tower field constructions, particularly for FFTs of length up to 64.

For this study, we select Aurora [10] to demonstrate the performance improvements achieved by optimizing the FFT algorithm using the Cantor special basis. While our optimization is applicable to all the aforementioned zkSNARKs, Aurora was chosen due to its small proof size, which makes it a strong candidate for post-quantum secure digital signature schemes. Accordingly, Aurora serves as the foundation of Preon [18], a post-quantum digital signature scheme that was a first-round candidate in NIST's PQC standardization process [42].

Contributions Our main contributions are summarized as follows:

- In the Cantor FFT algorithm, we present a theoretical analysis of the vanishing polynomials, providing a precise count of their terms based on Hamming weight. To the best of our knowledge, prior works have only reported upper bounds. We also efficiently compute the vanishing polynomials and multiplication factors, improving efficiency even without precomputation.

Table 1: Comparison of the number of finite field additions (#A) and multiplications (#M) in the Gao–Mateer (GM), LCH, and Cantor additive FFT algorithms of length n over general and Cantor special basis.

FFT	Basis	Basis Conversion	Evaluation
GM	General (Section 4.3)	#A: $\frac{1}{4}n(\log_2 n)^2 - \frac{1}{4}n\log_2 n$ #M: $n\log_2 n - n + 1$	#A: $n\log_2 n$ #M: $\frac{1}{2}n\log_2 n$
	Cantor (Section 4.4)	#A: $\frac{1}{4}n(\log_2 n)^2 - \frac{1}{4}n\log_2 n$ #M: 0	#A: $n\log_2 n$ #M: $\frac{1}{2}n\log_2 n$
LCH	General [37]	#A: $O(n(\log_2 n)^2)$ #M: $O(n\log_2 n)$	#A: $n\log_2 n$ #M: $\frac{1}{2}n\log_2 n$
	Cantor [37]	#A: $O(n\log_2 n\log_2\log_2 n)$ #M: 0	#A: $n\log_2 n$ #M: $\frac{1}{2}n\log_2 n$
Cantor	Cantor (Section 3.4)	N/A	#A: $\frac{1}{2}n\log_2 n + \frac{1}{2}n\sum_{r=0}^{\log_2(n)-1} 2^{\mathrm{wt}(r)}$ #M: $\frac{1}{2}n\log_2 n$

- We propose Cantor FFT building blocks and demonstrate notable performance gains in the current Aurora implementation [9] over the Gao–Mateer algorithm across all input sizes and the LCH algorithm for smaller circuits, which are prevalent in many zkSNARK applications [18,29]. Table 2 shows how our FFT optimizations using the Cantor special basis can accelerate both prover and verifier algorithms in Aurora.
- We provide a detailed breakdown of the Gao–Mateer algorithm's core components, Expand and Aggregate, and enhance their computational and space efficiency using the Cantor special basis. We also introduce precomputation techniques that substantially reduce overhead in the Cantor algorithm, along with two levels of precomputation for Gao–Mateer.
- We analyze the FFT call complexity in the Aurora zkSNARK, evaluating the number and size of each FFT/IFFT call based on R1CS constraints, variables, and the target security parameter. We also show how selecting the shift element in Aurora's affine subspaces reduces precomputation space complexity significantly in the Cantor FFT by leveraging the Cantor special basis.
- We provide a C++ implementation of the Cantor algorithm, along with the Gao–Mateer and LCH algorithms using the Cantor special basis, as well as our precomputation techniques. A comprehensive comparison of these algorithms is presented in Figure 1.

The paper is structured as follows. Section 2 covers mathematical preliminaries, including the Cantor and Gao–Mateer algorithms and R1CS encoding in Aurora. Section 3 presents Cantor algorithm optimizations, while Section 4 details the Gao–Mateer algorithm and its precomputations. Section 5 analyzes FFT

Table 2: Runtime (in seconds) of the Aurora [10] algorithms over $\mathbb{F}_{2^{256}}$ on the platform described in Section 6, based on the number of constraints N and the size of the codeword domain $|L|$.

		Aurora Prover			Aurora Verifier		
$\log_2(N)$	$\log_2(\|L\|)$	GM* libiop [9]	Cantor (this work)	LCH (this work)†	GM* libiop [9]	Cantor (this work)	LCH (this work)†
9	16	0.44	0.33	0.35	0.04	0.04	0.04
10	17	0.90	0.67	0.71	0.05	0.05	0.05
11	18	1.87	1.36	1.44	0.07	0.06	0.07
12	19	3.99	2.91	2.93	0.10	0.09	0.10
13	20	8.53	6.02	5.95	0.17	0.15	0.16
14	21	19.47	12.01	12.44	0.29	0.26	0.28
15	22	41.05	25.27	25.41	0.54	0.48	0.52
16	23	84.26	50.83	50.63	1.02	0.93	1.00
17	24	176.67	104.26	102.95	1.98	1.79	1.93
18	25	373.83	216.00	213.61	3.88	3.51	3.78
19	26	771.42	443.88	441.51	7.78	6.91	7.44

* Gao–Mateer FFT using standard basis.

† The LCH FFT using Cantor special basis from [19] is reimplemented in C++ to become compatible with `libiop` [9] and polymorphism over finite fields.

call complexity in Aurora, and Section 6 benchmarks the FFT algorithms and Aurora. Finally, Section 7 summarizes the findings and future directions.

2 Preliminaries

This section presents key definitions and preliminaries essential to our study.

2.1 Algebraic Foundations

Finite Field Let $\mathbb{F}_{2^k}$ be the finite field of order 2^k. We know that there exists a vector space isomorphism from $\mathbb{F}_{2^k}$ to $\mathbb{F}_2^k$ defined by $\mathbf{x} = (x_0\beta_0 + x_1\beta_1 + \cdots + x_{k-1}\beta_{k-1}) \mapsto (x_0, x_1, \ldots, x_{k-1})$, where $\{\beta_0, \beta_1, \ldots, \beta_{k-1}\}$ is a basis of $\mathbb{F}_{2^k}$. The polynomial ring over $\mathbb{F}_{2^k}$ in the variable x is denoted by $\mathbb{F}_{2^k}[x]$.

Affine Subspace Let us define the subspace W_m of $\mathbb{F}_{2^k}$ as the linear combinations of $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$. We order the elements of the subspace W_m by $\{\eta_0 = 0, \eta_1, \eta_2, \ldots, \eta_{2^m-1}\}$ where $\eta_j = \sum_{i=0}^{m-1} x_i\beta_i \quad \text{and} \quad j = \sum_{i=0}^{m-1} x_i 2^i, x_i \in \{0, 1\}$.

Note that for any $0 \leq m < k$, we can decompose the elements of W_{m+1} into two disjoint sets: the subspace W_m and the *affine subspace* $\beta_m + W_m$, where $\beta_m + W_m$ is the set obtained by translating (or shifting) the subspace W_m by the vector β_m.

Vanishing Polynomial The polynomial $\mathbb{Z}_{W_m}(x) = \prod_{a \in W_m}(x - a)$ which is a linearized polynomial, is given by $\mathbb{Z}_{W_m}(x) = \sum_{i=0}^{m} c_i x^{2^i}, c_i \in \mathbb{F}_{2^k}$. This polynomial is called the *vanishing polynomial* for the subspace W_m. Since $W_{m+1} = W_m \cup (\beta_m + W_m)$, we have $\mathbb{Z}_{W_{m+1}}(x) = (\mathbb{Z}_{W_m}(x))^2 - \mathbb{Z}_{W_m}(\beta_m) \cdot \mathbb{Z}_{W_m}(x)$.

Univariate Polynomials Vectorial Representation For any univariate polynomial $f(x) = \sum_{i=0}^{n-1} c_i x^i, c_i \in \mathbb{F}_{2^k}$, where $\deg(f) < n = 2^m$, the coefficients are represented as a vector of n elements, ordered from the constant term to the highest degree term. Namely, $\mathbf{f} = (c_0, \ldots, c_{n-1}), c_i \in \mathbb{F}_{2^k}$ represents the polynomial $f(x)$, where $\deg(f) < n$.

Additive Discrete Fourier Transform Now we will discuss the evaluation of a univariate polynomial $f(x)$ over the subspace W_m. The evaluation of $f(x)$ at the points $\eta_0, \eta_1, \ldots, \eta_{2^m-1}$ is given by $\hat{\mathbf{f}} = (f(\eta_0), f(\eta_1), \ldots, f(\eta_{2^m-1}))$.

This set of evaluations is referred to as the additive discrete Fourier transform (DFT) of $f(x)$ over the subspace W_m. We sometimes refer to the vector $\hat{\mathbf{f}}$ as the *discrete Fourier transform of length* $n = 2^m$ for the function $f(x)$, denoted by $\mathrm{DFT}(f, W_m)$. *The additive FFT* is an efficient method for computing $\mathrm{DFT}(f, W_m)$, which we will denote as $\mathrm{FFT}(f, W_m)$.

2.2 Cantor Algorithm

The evaluation of a polynomial $f(x)$ of degree less than $n = 2^m$ over the subspace W_m using Cantor's algorithm is valid only when W_m is a subspace of the field (or subfield) $\mathbb{F}_{2^k}$, where $k = 2^\ell$. Cantor introduced a special basis to facilitate the efficient evaluation of $f(x)$.

The Cantor Special Basis Consider the function $S : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$ defined by $S(x) = x^2 + x$, and let the following sequence of functions be defined recursively: $S^0(x) = x$ and $S^m(x) = S(S^{m-1}(x))$. A nonrecursive formula for $S^m(x)$ is $S^m(x) = \sum_{i=0}^{m} \binom{m}{i} x^{2^i}$, where $\binom{m}{i}$ denotes the binomial coefficient reduced modulo 2. For $m = 2^t$, we have $S^{2^t} = x^{2^{2^t}} + x$. The *Cantor special basis* $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ for the subspace W_m is defined as $S(\beta_i) = \beta_{i-1}$ for $i = 1, \ldots, m-1$ with $\beta_0 = 1$. With this basis, we have $S^i(\beta_i) = 1$ for $i = 0, 1, \ldots, m-1$. Consequently, in the context of the Cantor special basis, the vanishing polynomial of the subspaces simplifies to $\mathbb{Z}_{W_i}(x) = S^i(x)$ for $i = 0, 1, \ldots, m$.

Polynomial Evaluation Let $f(x) \in \mathbb{F}_{2^k}[x]$ be a polynomial of degree less than $n = 2^m$ and we want to evaluate $f(x)$ over the affine subspace $\theta + W_m = \theta + \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$, where $\{\beta_0 = 1, \beta_1, \ldots, \beta_{m-1}\}$ is a Cantor special basis. The evaluation of $f(x)$ using the Cantor algorithm proceeds as follows: first, compute two polynomials $f_0(x)$ and $f_1(x)$ such that $f_0(x) = f(x)$ for all $x \in \theta + W_{m-1}$ and $f_1(x) = f(x)$ for all $x \in \theta + \beta_{m-1} + W_{m-1}$. The polynomials $f_0(x)$ and $f_1(x)$

can be obtained by taking the remainders of $f(x)$ when divided by the vanishing polynomials of the affine subspaces. Specifically, we have

$$f_0(x) = f(x) \mod S^{m-1}(x+\theta) \quad \text{and} \quad f_1(x) = f(x) \mod S^{m-1}(x+\theta+\beta_{m-1}).$$

We then proceed by recursively evaluating $f_0(x)$ and $f_1(x)$ over the affine subspaces $\theta + W_{m-1}$ and $\theta + \beta_{m-1} + W_{m-1}$, respectively. The recursion continues until all the resulting polynomials $f_0(x)$ and $f_1(x)$ are constants. This is summarized in Algorithm 6 in Appendix A.

2.3 Gao–Mateer Algorithm

Note that the evaluation of $f(x)$ over $\theta + W_m$ is equivalent to the evaluation of the function $g(x) = f(\beta_{m-1}x)$ over

$$\beta_{m-1}^{-1}(\theta + W_m) = \beta_{m-1}^{-1}\theta + \beta_{m-1}^{-1} \cdot W_m = (\theta_0 + G) \cup (1 + \theta_0 + G),$$

where $\theta_0 = \beta_{m-1}^{-1}\theta$ and G is a $m-1$ dimensional subspace given by $G = \langle \gamma_0, \ldots, \gamma_{m-2} \rangle$, where $\gamma_i = \beta_i \cdot \beta_{m-1}^{-1}$ for $i = 0, 1, \ldots, m-2$.

Now, consider the Taylor expansion of $g(x)$ at $x^2 + x$ to obtain $f_0(x)$ and $f_1(x)$. Specifically, express $g(x)$ as

$$g(x) = \sum_{i=0}^{\ell-1} (g_{i0} + g_{i1}x)(x^2 + x)^i, \text{ where } \ell = 2^{m-1} \text{ and } g_{ij} \in \mathbb{F}_{2^k}$$

and define $f_0(x) = \sum_{i=0}^{\ell-1} g_{i0}x^i$ and $f_1(x) = \sum_{i=0}^{\ell-1} g_{i1}x^i$. Then, the FFT of $g(x)$ over $\beta_{m-1}^{-1}(\theta + W_m)$ can be derived from the evaluation of $f_0(x)$ and $f_1(x)$ over $\theta_0^2 + \theta_0 + D$, where $D = \langle \delta_0, \delta_1, \ldots, \delta_{m-2} \rangle$ is a $m-1$ dimensional subspace with $\delta_i = \gamma_i^2 + \gamma_i$ for $i = 0, 1, \ldots, m-2$. By applying this reduction step again to $\mathrm{FFT}(f_0, \theta_0^2 + \theta_0 + D)$ and $\mathrm{FFT}(f_1, \theta_0^2 + \theta_0 + D)$, we continue until D has a dimension of 1. This is summarized in Algorithm 7 in Appendix A.

2.4 Aurora zkSNARK

Aurora [10] is a zkSNARK for R1CS relations. It encodes an R1CS instance into entries of Reed–Solomon (RS) codewords and to generate an argument regarding the R1CS instance, it employs the fast RS interactive oracle proof (IOP) of proximity (FRI) [3] to prove that the given codewords are close to a low-degree polynomial.

Definition 1 (Rank-1 Constraint System (R1CS)). *Let d_1, d_2, and d_3 denote the number of constraints, variables, and public inputs. The R1CS relation consists of $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}_{2^k}^{d_1 \times (d_2+1)}$ and public inputs $\mathbf{v} \in \mathbb{F}_{2^k}^{d_3}$. $\mathbf{w} \in \mathbb{F}_{2^k}^{d_2-d_3}$, denoting private (auxiliary) inputs, satisfies the system if $\mathbf{Az} \circ \mathbf{Bz} = \mathbf{Cz}$, where $\mathbf{z} := (1, \mathbf{v}, \mathbf{w}) \in \mathbb{F}_{2^k}^{d_2+1}$ and "$\circ$" denotes the Hadamard product.*

Definition 2 (Reed–Solomon (RS) Code). *Let $L \subseteq \mathbb{F}_{2^k}$ denote the codeword domain and $\rho \in [0,1]$ denote the rate parameter. $RS[L,\rho]$ is the set of all codewords $\hat{\mathbf{f}} : L \to \mathbb{F}_{2^k}$ that are the evaluation of polynomials over L with degree $< \rho|L|$.*

At the beginning of the Aurora protocol, the prover and verifier establish a finite field $\mathbb{F}_{2^k}$, a security parameter $\boldsymbol{\lambda}$, an RS rate ρ, an FRI localization parameter η, an R1CS instance, and two subspaces $H_1, H_2 \subset \mathbb{F}_{2^k}$, where $|H_1| = d_1$ and $|H_2| = d_2 + 1$ such that $H_1 \subseteq H_2$ or $H_2 \subseteq H_1$. We can write $H_1 \cup H_2 = \{h_0, \ldots, h_{t-1}\}$, where $t = |H_1 \cup H_2| = \max(d_1, d_2 + 1)$. Finally, the codeword domain L is determined such that $t \leq \rho|L|$ and $L \cap (H_1 \cup H_2) = \emptyset$. Section 5 describes how $|L|$ is determined. Given these parameters, the repetition counts λ_i for the *lincheck* protocol and λ'_i for the *FRI low-degree testing* (FRI-LDT) protocol, Aurora's two main sub-protocols, are determined.

The prover algorithm starts by interpolating a polynomial $f_{(1,\mathbf{v})}(x)$ of degree $d_3 + 1$ such that $f_{(1,\mathbf{v})}(h_0) = 1$ and $f_{(1,\mathbf{v})}(h_{i+1}) = v_i$ for $i = 0, \ldots, d_3 - 1$, where v_i denotes the i-th element in $\mathbf{v}$. Then, during each round $\ell = 1, \ldots, \lambda_i$ of the lincheck protocol, four public-coin random numbers $\alpha_\ell, s^A_\ell, s^B_\ell, s^C_\ell \in_R \mathbb{F}_{2^k}$ are sampled. Then, both prover and verifier interpolate

1. A polynomial $p_{\alpha_\ell}(x)$ of degree $t - 1$, such that $p_{\alpha_\ell}(h_i) = {\alpha_\ell}^{i+1}$ for $i = 0, \ldots, d_1 - 1$ and if $t > d_1$, then $p_{\alpha_\ell}(h_i) = 0$ for $i = d_1, \ldots, t - 1$.
2. Three polynomials $p^{\mathbf{A}}_{\alpha_\ell}(x)$, $p^{\mathbf{B}}_{\alpha_\ell}(x)$, and $p^{\mathbf{C}}_{\alpha_\ell}(x)$ of degree $t - 1$, such that $p^{\mathbf{M}}_{\alpha_\ell}(h_j) = \sum_{i=0}^{d_1-1} \mathbf{M}_{i,j} {\alpha_\ell}^{i+1}$ for $j = 0, \ldots, d_2$ and $M \in \{A, B, C\}$. If $t > d_2 + 1$, then $p^{\mathbf{M}}_{\alpha_\ell}(h_j) = 0$ for $j = d_2 + 1, \ldots, t - 1$.
3. A polynomial $p^{ABC}_{\alpha_\ell}(h_j) = \sum_{M \in \{A,B,C\}} s^{\mathbf{M}}_\ell p^{\mathbf{M}}_{\alpha_\ell}(h_j)$, which is a random combination of defined $p^{\mathbf{A}}_{\alpha_\ell}(x)$, $p^{\mathbf{B}}_{\alpha_\ell}(x)$, and $p^{\mathbf{C}}_{\alpha_\ell}(x)$.

Table 6 in Appendix C lists the primary codewords obtained by encoding the R1CS instance, together with the polynomials subsequently derived during the prover's computation. To ensure zero-knowledge against a $\mathtt{b}$-query bounded malicious verifier, every degree is raised by $\mathtt{b}$; hence any $\mathtt{b}$ evaluations from of the polynomials in L are independent and uniform in $\mathbb{F}_{2^k}$ [7]. Finally, the codewords' low degree is proven with the FRI protocol.

3 Cantor Algorithm Building Blocks

The Cantor FFT of length $n = 2^m$ consists of m iterative rounds to evaluate a polynomial $f(x) \in \mathbb{F}_{2^k}[x]$ of degree $< 2^m$ over the affine subspace $\theta + W_m$, where $\theta \in \mathbb{F}_{2^k}$ and W_m must be generated by the Cantor special basis as described in Section 2.2. Also, for the Cantor special basis, we know that $\mathbb{Z}_{W_i}(x) = S^i(x)$ for $i = 0, 1, \ldots, m$. Thus, the coefficients of the vanishing polynomials $\mathbb{Z}_{W_i}(x)$ are in $\mathbb{F}_2$. Additionally, we have $S^i(\beta_i) = 1$.

Let $0 \leq r \leq m - 1$ be the round number. In each round r, the algorithm processes 2^r polynomials of degree $< 2^{m-r}$, resulting in 2^{r+1} polynomials of degree $< 2^{m-r-1}$ at the end of the round. $f(x)$ is represented by $\mathbf{f}$, a vector of

size 2^m that stores the coefficients as described in Section 2. We use the same vector to store all intermediate polynomials during each round. For example, in round r, $\mathbf{f}$ stores the concatenation of the 2^{r+1} polynomials. Finally, in round $r = m - 1$, the algorithm outputs 2^m constant values stored in the vector $\mathbf{f}$, representing the evaluations of $f(x)$ over $\theta + W_m$.

Before presenting the details of our Cantor algorithm implementation, we first explain the selection process of the Cantor special basis of length m, denoted by $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$. From [26, Appendix], we know that $W_m = \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$ must be a subspace of the field (or subfield) $\mathbb{F}_{2^{2^\ell}}$. To determine the Cantor special basis for $\mathbb{F}_{2^{2^\ell}}$, we begin by defining a basis $\{\beta_0, \beta_1, \ldots, \beta_{2^\ell-1}\}$ such that $\mathrm{Tr}_{\mathbb{F}_{2^{2^\ell}}/\mathbb{F}_2}(\beta_{2^\ell-1}) = 1$. Then, we recursively determine the remaining basis elements by $\beta_{j-1} = \beta_j^2 + \beta_j$ for $1 \leq j \leq 2^\ell - 1$. We then select the first m elements from this basis to construct our Cantor special basis of dimension m. In our implementation, we randomly try different values of $\beta_{2^\ell-1}$ and compute their trace. With a probability of 0.5, the trace 1.

3.1 Vanishing Polynomials

In the Cantor additive FFT of length 2^m, the vanishing polynomials $\mathbb{Z}_{W_0}, \mathbb{Z}_{W_1}, \ldots, \mathbb{Z}_{W_{m-1}}$ must be computed to perform the division algorithm. The coefficients in each $\mathbb{Z}_{W_i}(x)$ are derived from

$$\mathbb{Z}_{W_i}(x) = \sum_{j=0}^{i} \left[\binom{i}{j} \mod 2 \right] x^{2^j}.$$

Employing Lucas's theorem [39], we efficiently compute $\binom{i}{j} \equiv \prod_{k=0}^{t-1} \binom{i_k}{j_k} \bmod 2$, where

$$i = i_0 + i_1 2 + i_2 2^2 + \ldots + i_{t-1} 2^{t-1} \quad (i_k \in \{0, 1\}),$$
$$j = j_0 + j_1 2 + j_2 2^2 + \ldots + j_{t-1} 2^{t-1} \quad (j_k \in \{0, 1\}).$$

Theorem 1. *[24, Theorem 2] The number of integers j not exceeding i for which $\binom{i}{j} \not\equiv 0 \pmod 2$ is $\prod_{k=0}^{t}(i_k + 1)$.*

Thus, by the above theorem we conclude that the number of non-zero coefficients in $\mathbb{Z}_{W_i}(x)$ equals $2^{\mathrm{wt}(i)}$, where $\mathrm{wt}(i)$ denotes the Hamming weight of i, i.e., the number of bits equal to 1. Since the number of non-zero coefficients in most vanishing polynomials is considerably less than its degree (i.e., 2^i), we avoid using vectorial representation of univariate polynomials described in Section 2 for these polynomials. Otherwise, the polynomial division algorithm would require excessive addition operation on zero coefficients.

To efficiently represent the vanishing polynomials, we store the index number of the coefficients in the reverse order. Specifically, let $\mathbf{z}_i$ represent $\mathbb{Z}_{W_i}(x) = \sum_{j=0}^{i} c_j x^{2^j}$ in our implementation. We define

$$\mathbf{z}_i = (\zeta_0, \zeta_1, \ldots, \zeta_{2^{\mathrm{wt}(i)}-1}) = (2^i - 2^j | c_j = 1 \text{ in } \mathbb{Z}_{W_i}(x)),$$

Algorithm 1: Vanishing Polynomial (i, θ)

Input: $i \in \mathbb{N}$ and $\theta \in \mathbb{F}_{2^k}$
Output: $\mathbf{z}_i$, eval.

$\ell \leftarrow 0$
eval $\leftarrow \theta$
for $j = 0$ **to** $i - 1$ **do**
  $t \leftarrow \lceil \log_2(j+1) \rceil$
  $k \leftarrow 0$
  while $(i_k \vee \neg j_k) \wedge (k < t)$ **do**
    $k \leftarrow k + 1$
  end
  if $k = t$ **then**
    $\zeta_\ell \leftarrow 2^i - 2^j$
    eval $\leftarrow$ eval $+ \theta^{2^j}$
    $\ell \leftarrow \ell + 1$
  end
end
eval $\leftarrow$ eval $+ \theta^{2^i}$ // As the highest degree term is not in $\mathbf{z}_i$
assert $\ell = 2^{\mathrm{wt}(i)} - 2$
return $\mathbf{z}_i \leftarrow (\zeta_0, \zeta_1, \ldots, \zeta_{2^{\mathrm{wt}(i)}-2})$, eval.

where Algorithm 1 describes how $\mathbf{z}_i$ is computed.

Algorithm 1 also computes the evaluation of $\mathbb{Z}_{W_i}(\theta)$ which is used later in Section 3.3. Reversing the order (i.e., measuring the distance from 2^i rather than from 0) eliminates the need for degree shifting in $\mathbb{Z}_{W_i}(x)$ during the division rounds. Additionally, since $\binom{i}{i} = 1$, c_i is always one, and $\zeta_{2^{\mathrm{wt}(i)}-1}$ is zero which can be omitted from $\mathbf{z}_i$. This omission excludes this coefficient from the polynomial division algorithm, saving one addition per division round while keeping the quotient in the same vector as the dividend.

3.2 Polynomial Division

In round r of the Cantor algorithm, the 2^r polynomials $f_{i,r}(x)$, each of degree $< 2^p$, are divided by $\mathbb{Z}_{Wp-1}(x+\theta_{i,r})$ and $\mathbb{Z}_{Wp-1}(x+\theta_{i,r}+\beta_{p-1})$, where $p = m-r$ and $0 \leq i \leq 2^r - 1$. The corresponding remainders of these divisions for each $f_{i,r}(x)$ are outputted to be processed in the next round. Since the vanishing polynomials are linearized polynomials, we have

$$\mathbb{Z}_{Wp-1}(x + \theta_{i,r}) = \mathbb{Z}_{Wp-1}(x) + \mathbb{Z}_{Wp-1}(\theta_{i,r}), \text{ and}$$
$$\mathbb{Z}_{Wp-1}(x + \theta_{i,r} + \beta_{p-1}) = \mathbb{Z}_{Wp-1}(x) + \mathbb{Z}_{Wp-1}(\theta_{i,r}) + \mathbb{Z}_{Wp-1}(\beta_{p-1}),$$

where $\mathbb{Z}_{Wp-1}(\beta_{p-1}) = 1$. Since $\deg(f_i(x)) < 2\deg(\mathbb{Z}_{Wp-1}(x))$, we reduce the two divisions required for each $f_i(x)$ to a single division by $\mathbb{Z}_{Wp-1}(x)$, and then compute the remainders of the original divisions accordingly.

Since the coefficients in $\mathbb{Z}_{W_{p-1}}(x)$ are in $\mathbb{F}_2$, division by $\mathbb{Z}_{W_{p-1}}(x)$ requires only additions. In the division algorithm, the dividend polynomial, denoted by $f_{i,r}(x)$, is added to scaled degree-shifts of $\mathbb{Z}_{W_{p-1}}(x)$. However, we eliminate the need for degree shifts by using the reversed index order in $\mathbf{z}_i$, which represents the distance of each non-zero coefficient from the highest degree (i.e., 2^p).

Dividing $f_{i,r}(x)$, a polynomial of degree $< 2^p$, by $\mathbb{Z}_{W_{p-1}}(x)$ yields the quotient $q_{i,r}(x)$ and the remainder $r_{i,r}(x)$, each with degree $< 2^{p-1}$. Let $\mathbf{f}_{i,r}$ represent the $(i+1)$-th sub-vector of 2^p elements in the vector $\mathbf{f}$ at the beginning of round r. This sub-vector stores the coefficients of $f_{i,r}(x)$, ordered from the constant term to the highest degree term. Our polynomial division algorithm begins with the coefficient of the highest degree term in $\mathbf{f}_{i,r}$ and subtracts that coefficient from the lower-degree coefficients, spaced by distances determined by $\mathbf{z}_i$. Since $\mathbf{z}_i$ does not include zero, the coefficient of the highest degree term remains unaffected. In the next round of the polynomial division, the algorithm repeats this process with the second highest degree term. After 2^{p-1} rounds, the higher-degree (right) half of $\mathbf{f}_{i,r}$ stores the coefficients of $q_{i,r}(x)$, while the lower-degree (left) half stores the coefficients of $r_{i,r}(x)$. Then, the algorithm processes inputs to the next round

$$f_{2i,r+1}(x) = r_{i,r}(x) - \mathbb{Z}_{W_{p-1}}(\theta_{i,r})\, q_{i,r}(x), \text{ and}$$
$$f_{2i+1,r+1}(x) = f_{2i,r+1}(x) - q_{i,r}(x),$$

where $\mathbb{Z}_{W_{p-1}}(\theta_{i,r})$ denotes the evaluation of the vanishing polynomial at $\theta_{i,r}$. Figure 2 in Appendix D illustrates the computation of the polynomials for the next round from the quotient and remainder in each round. Our polynomial division algorithm implementation integrates the computation of the quotient, remainder, and the polynomials for the next round.

Algorithm 2: Polynomial Division ($\mathbf{f}_{\text{in}}$, $\mathbf{z}_{p-1}$, $\mathbb{Z}_{W_{p-1}}(\theta_{i,r})$, p, i)

Input: $\mathbf{f}_{\text{in}} = (c_0, c_1, \ldots, c_{n-1})$, $\mathbf{z}_{p-1} = (\zeta_0, \zeta_1, \ldots, \zeta_{2^{\text{wt}(p-1)}-2})$, $\mathbb{Z}_{W_{p-1}}(\theta_{i,r}) \in \mathbb{F}_{2^k}$, $p = m - r$, and i determines the polynomial $f_{i,r}(x)$, of degree $< 2^p$, in $\mathbf{f}_{\text{in}}$.
Output: $\mathbf{f}_{\text{out}}$.

```
offset ← i × 2^p // The offset at which the coefficients of f_{i,r}(x) are in f_in
for k = 2^p + offset − 1 to 2^{p−1} + offset do
    // Iterates over the higher-degree half in decreasing order
    for ℓ = 0 to 2^{wt(p−1)} − 2 do
        c_{(k−ζ_ℓ)} ← c_{(k−ζ_ℓ)} + c_k
    end
    c_{(k−2^{p−1})} ← c_{(k−2^{p−1})} + c_k × Z_{W_{p−1}}(θ_{i,r}) // Computes f_{2i,r+1}(x)
end
for k = 2^{p−1} + offset to 2^p + offset − 1 do
    c_k ← c_k + c_{(k−2^{p−1})} // Computes f_{2i+1,r+1}(x)
end
return f_out ← (c_0, c_1, ..., c_{n−1}).
```

The Canopy module, described in the next section, is responsible for providing all of the inputs of the polynomial division algorithm.

3.3 Canopy Module

The Cantor algorithm is implemented by Canopy modules of varying input sizes 2^p and indices i, denoted by $\mathsf{Canopy}_{p,i}$. The index i indicates the module number

in each round and determines the offset from the start of the $\mathbf{f}$ vector, where the coefficients of the input polynomial begin with $\mathsf{offset} = i2^p$.

The $\mathsf{Canopy}_{p,i}$ determines $\theta_{i,r}$ and then evaluate $\mathbb{Z}_{W_{p-1}}(\theta_{i,r})$, which is necessary for running Algorithm 2. Let the Cantor algorithm evaluate $f(x)$ of degree $< 2^m$ over $\theta + \langle \beta_0 = 1, \beta_1, \ldots, \beta_{m-1} \rangle$. Let $\theta_{0,0} = \theta$, and for $1 \leq r \leq m-1$ and $0 \leq i \leq 2^r - 1$, $\theta_{i,r}$ is determined recursively according to the following rules:

$$\theta_{i,r} = \begin{cases} \theta_{i/2,r-1} & \text{if } i \bmod 2 = 0, \\ \theta_{(i-1)/2,r-1} + \beta_p & \text{if } i \bmod 2 = 1, \end{cases}$$

where $p = m - r$. This equation can be simplified to $\theta_{i,r} = \theta_{\lfloor i/2 \rfloor, r-1} + (i \bmod 2)\beta_p$, and can be written as

$$\theta_{i,r} = \theta + \sum_{j=0}^{r-1} \left(\lfloor \frac{i}{2^j} \rfloor \bmod 2 \right) \beta_{p+j} = \theta + \sum_{j=0}^{r-1} i_j \, \beta_{p+j},$$

where $i = i_0 + i_1 2 + i_2 2^2 + \ldots + i_{r-1} 2^{r-1}$ $(i_j \in \mathbb{F}_2)$, denotes the binary representation of i. Then, to evaluate $\mathbb{Z}_{W_{p-1}}(\theta_{i,r})$, we employ the rule $S^i(\beta_{i+\ell}) = \beta_\ell$ provided earlier in this section to simplify the computation. Specifically, we write $\mathbb{Z}_{W_{p-1}}(\beta_{p-1+j}) = \beta_j$. Therefore, $\mathbb{Z}_{W_{p-1}}(\theta_{i,r})$ can be evaluated as

$$\mathbb{Z}_{W_{p-1}}(\theta_{i,r}) = \mathbb{Z}_{W_{p-1}}(\theta) + \sum_{j=0}^{r-1} i_j \, \beta_{j+1},$$

where $\mathbb{Z}_{W_{p-1}}(\theta)$ is evaluated at each round while constructing $\mathbf{z}_{p-1}$ from $\mathbb{Z}_{W_{p-1}}(x)$ during Algorithm 1. The computation of $\mathbb{Z}_{W_{p-1}}(\theta)$ is shared across all the Canopy modules in each row since the vanishing polynomial remains consistent.

Algorithm 3: Cantor Algorithm ($\mathbf{f}_{\text{in}}$, θ, $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$)

Input: $\mathbf{f}_{\text{in}}$ is a vector of size 2^m which represents the coefficients in $f(x)$ (where $\deg f < 2^m$), $\theta \in \mathbb{F}_{2^k}$ is the affine shift, and $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ is the Cantor special basis.
Output: $\mathbf{f}_{\text{out}}$ is the vector of the evaluations of $f(x)$ over $\theta + \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$.

```
for r = 0 to m − 1 do
    p ← m − r
    z_{p−1}, eval ← Vanishing Polynomial (p − 1 , θ) // Algorithm 1
    for i = 0 to 2^r − 1 do
        Canopy_{p,i} (f_in, {β_0, β_1, ..., β_{m−1}}, z_{p−1}, eval, p, i) // Algorithm 8
    end
end
return f_out.
```

Algorithm 8 in Appendix D details the steps within the $\mathsf{Canopy}_{p,i}$ module, and Algorithm 3 describes the implementation of the Cantor algorithm based on those modules.

3.4 Detailed Cost Analysis

At the r-th iteration of the algorithm, we perform 2^r divisions by the polynomial $\mathbb{Z}_{W_{m-r-1}}(x)$. Moreover, as discussed in Section 3.1, each of these divisions results in the saving of one addition per division iteration. Consequently, the total number of additions resulting from polynomial division in the Cantor additive FFT is given by

$$\sum_{r=0}^{m-1} 2^r \cdot 2^{m-r-1}\left(2^{\mathrm{wt}(m-r-1)} - 1\right) = 2^{m-1}\sum_{r=0}^{m-1} 2^{\mathrm{wt}(r)} - m2^{m-1}$$

From Steps 6 and 9 of Algorithm 2, we know that for the input polynomial in the r-th iteration, we require 2×2^r polynomial additions, each of degree less than 2^{m-r-1}. This leads to a total of $\sum_{r=0}^{m-1} 2 \cdot 2^r \cdot 2^{m-r-1} = 2^m m$ additions. Therefore, the total number of additions in the Cantor additive FFT is given by

$$2^m m + 2^{m-1}\sum_{r=0}^{m-1} 2^{\mathrm{wt}(r)} - m2^{m-1} = \frac{1}{2}n\log_2 n + \frac{1}{2}n \sum_{r=0}^{\log_2(n)-1} 2^{\mathrm{wt}(r)}. \quad (1)$$

On the other hand, from Step 6 of Algorithm 2, we know that for the input polynomial in the r-th iteration, we require $2^r \times 2^{m-r-1} = 2^{m-1}$ multiplications. Thus, the number of multiplications in the Cantor additive FFT is given by $\sum_{r=0}^{m-1} 2^{m-1} = \frac{1}{2}n\log_2 n$.

If the Cantor additive FFT is performed over a subspace W_m, due to Step 6 of the Algorithm 2, we must account for a reduction of $\sum_{r=0}^{m-1} 2^{m-r-1} = 2^m - 1 = n - 1$ in both additions and multiplications. Thus, the costs for additions and multiplications are changed to $\frac{1}{2}n\log_2 n + \frac{1}{2}n\sum_{r=0}^{\log_2(n)-1} 2^{\mathrm{wt}(r)} - n + 1$ and $\frac{1}{2}n\log_2 n - n + 1$, respectively.

Remark 1. Equation 1 provides an exact count of the additions required in the Cantor additive FFT, while, to our knowledge, previous studies have only established upper bounds, such as $O(n(\log_2 n)^{1.58})$ [22] or, more precisely, as shown in [41], $\frac{1}{2}n(\log_2 n)^{1.58} + n\log_2 n$. Our exact count offers a significant improvement over these bounds on the number of additions. For example, when $n = 2^4$, the Cantor additive FFT requires only 104 additions, while [41] estimates 136 additions.

3.5 Precomputation

The multiplication factors $\mathbb{Z}_{W_{p-1}}(\theta_{i,r})$ depend only on the input size and the affine shift that defines the evaluation domain. Likewise, the vanishing polynomials are fixed and independent of input. The storage required to store the precomputed values for the Cantor algorithm of length $n = 2^m$ is $2^m - 1$ field elements to store multiplication factors and $\sum_{i=0}^{m-1}\left(2^{\mathrm{wt}(i)} - 1\right)$ integers to store vanishing polynomials which is negligible compared to the field elements. Over $\mathbb{F}_{2^{256}}$ this translates to 32KB, 1MB, and 32MB for $m = 10, 15, 20$, respectively.

In the *special case* where the affine shift θ lies in the Cantor special basis, the multiplication factors are simply linear combinations of the basis vectors in the subspace $W_m = \langle \beta_0, \ldots, \beta_{m-1} \rangle$. So, a single lookup table that lists these combinations therefore suffices. With a practical upper bound of $m = 32$ for the FFT input size, precomputing the entire span of W_m is infeasible. Instead, the basis is partitioned into four blocks, and the spans

$$\langle \beta_0, \ldots, \beta_7 \rangle, \langle \beta_8, \ldots, \beta_{15} \rangle, \langle \beta_{16}, \ldots, \beta_{23} \rangle, \langle \beta_{24}, \ldots, \beta_{31} \rangle$$

are precomputed. Each block contributes 2^8 field elements, so the complete table contains $4 \times 2^8 = 1024$ elements (compact enough to hard-code). For $\mathbb{F}_{2^{256}}$ this table occupies only 32 KB. Note that this special case applies in Aurora, where the domain L is an affine subspace whose affine shift is itself a basis element.

Table 3: Runtime speedup of the Cantor FFT general (gen.) precomputation method over $\mathbb{F}_{2^{256}}$ for an arbitrary θ versus the special (spec.) choice $\theta = \beta_{31}$ (see Section 6 for platform details).

m	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Gen.	1.38	1.29	1.21	1.13	1.09	1.07	1.07	1.37	1.47	1.49	1.46	1.45	1.51	1.49	1.47	1.43
Spec.	1.24	1.17	1.09	1.06	1.09	1.08	1.08	1.38	1.49	1.51	1.48	1.45	1.43	1.42	1.41	1.37

4 Gao–Mateer Algorithm Building Blocks

The Gao–Mateer FFT implementation of length $n = 2^m$ consists of $2m$ iterative rounds to evaluate a polynomial $f(x) \in \mathbb{F}_{2^k}[x]$ of degree $< 2^m$ over the affine subspace $\theta + W_m$, where $\theta \in \mathbb{F}_{2^k}$ and $W_m = \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$.

We describe the Gao–Mateer algorithm through two primary modules: the Expand module and the Aggregate module. Expand is an r-round algorithm where in each round $0 \leq r \leq m-1$, it expands 2^r polynomials of degree $< 2^{m-r}$ into 2^{r+1} smaller polynomials of degree $< 2^{m-r-1}$. Similar to our Cantor algorithm implementation, only one vector of length 2^m denoted by $\mathbf{f}$ is required to store all the polynomials in each round. Aggregate is an r-round algorithm that takes the output of Expand, and iteratively folds them over r rounds, ultimately producing the evaluations of $f(x)$ on $\theta + W_m$.

4.1 Expand Module

The Expand module involves multiple invocations of Taylor expansion, polynomial scaling (e.g., $f(\beta_m x)$), and computing basis vectors and shift elements for the Aggregate module. The Taylor expansion algorithm implemented in [9] is described in Algorithm 9 in Appendix D. To have the vector representation of

coefficients in the Gao–Mateer algorithm, it is required to have even-odd rearrangement of indices after each Taylor expansion, however, Algorithm 9 lets us omit those arrangements, instead add one bit-reversal rearrangement at the end of the Expand module. This also keeps the coefficients of terms with the same degree placed next to each other, thus allowing multiplying consecutive elements by the same scaling factor.

Algorithm 4: Expand $(\mathbf{f}_{\text{in}}, \theta, \{\beta_{0,0}, \ldots, \beta_{0,m-1}\})$

Input: $\mathbf{f}_{\text{in}} = (c_0, c_1, \ldots, c_{n-1})$, which represents $f(x) \in \mathbb{F}[x]$ of degree $< n = 2^m$, $\theta \in \mathbb{F}_{2^k}$ is the affine shift, and $\beta_{0,i} \in \mathbb{F}_{2^k}$ are the basis of W_m.
Output: $\mathbf{f}_{\text{out}}$, $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_{m-1})$, $\boldsymbol{\Gamma} = (\mathbf{G}_0 = \emptyset, \mathbf{G}_1, \ldots, \mathbf{G}_{m-1})$, where θ_r and $\mathbf{G}_r = \{\gamma_{r,0}, \ldots, \gamma_{r,r-1}\}$ denote the affine shift and basis corresponding to round r of the Aggregate module respectively.

```
for r = 0 to m − 1 do
    // Scaling polynomials:
    ψ ← 1 // ψ denotes the scaling factor of each term
    offset ← 2^r
    while offset ≤ 2^m − 1 do
        for i = 0 to 2^r − 1 do
            c_{offset+i} ← c_{offset+i} × ψ
        end
        ψ ← ψ × β_{r,m−r−1}
        offset ← offset + 2^r
    end
    (c_0, ..., c_{n−1}) ← Taylor Expansion ((c_0, ..., c_{n−1}), r) // Algorithm 9
    for i = 0 to m − r − 2 do
        γ_{m−r−1,i} ← β_{r,i} × β^{−1}_{r,m−r−1}
        β_{r+1,i} ← γ^2_{m−r−1,i} + γ_{m−r−1,i}
    end
    G_{m−r−1} ← (γ_{m−r−1,0}, ..., γ_{m−r−1,m−r−2})
    θ_{m−r−1} ← θ × β^{−1}_{r,m−r−1}
    θ ← θ^2_{m−r−1} + θ_{m−r−1}
end
(c_0, ..., c_{n−1}) ← Bit-reversal Rearrangement (c_0, ..., c_{n−1})
return f_out ← (c_0, c_1, ..., c_{n−1})
```

4.2 Aggregate Module

For $0 \le r \le m-1$, let $\theta_r + \mathbf{G}_r = \langle \gamma_{r,0}, \ldots, \gamma_{r,r-1} \rangle$ be provided in round $m-r-1$ of the Expand module, the round r of the Aggregate module spans $\theta_r + \mathbf{G}_r$ and combines 2^r adjacent elements in $\mathbf{f}$. Algorithm 5 describes the Aggregate module.

4.3 Detailed Cost Analysis

We now compute the number of multiplications and additions required by the algorithm. From Algorithm 4, we know that at the r-th iteration, we need to scale 2^r polynomials, each of degree 2^{m-r}. Thus, the multiplication for the scaling is given by $\sum_{r=0}^{m-1} 2^r \cdot (2^{m-r} - 1) = 2^m m - \sum_{r=0}^{m-1} 2^r = n \log_2 n - n + 1$.

Algorithm 5: Aggregate $(\mathbf{f}_{\text{in}}, \boldsymbol{\Gamma}, \boldsymbol{\theta})$

Input: $\mathbf{f}_{\text{in}} = (c_0, c_1, \ldots, c_{n-1})$ is a vector of length $n = 2^m$, $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_{m-1})$, $\boldsymbol{\Gamma} = (\mathbf{G}_0 = \emptyset, \mathbf{G}_1, \ldots, \mathbf{G}_{m-1})$, where θ_r and $\mathbf{G}_r = \{\gamma_{r,0}, \ldots, \gamma_{r,r-1}\}$ denote the affine shift and basis corresponding to round r.

Output: $\mathbf{f}_{\text{out}}$ is the vector of the evaluations of $f(x)$ over $\theta + W_m$

```
1  for r = 0 to m − 1 do
2      {η_0, η_1, ..., η_{2^r−1}} ← Span(G_r, θ_r)
3      for j = 0 to 2^{m−r−1} − 1 do
4          d ← j2^{r+1}
5          for i = 0 to 2^r − 1 do
6              c_{d+i} ← c_{d+i} + c_{d+2^r+i} × η_i
7              c_{d+2^r+i} ← c_{d+2^r+i} + c_{d+i}
8          end
9      end
10 end
11 return f_out ← (c_0, c_1, ..., c_{n−1})
```

From Algorithm 5, we know that at the r-th iteration, the number of required multiplications is $2^r \cdot 2^{m-r-1} = 2^{m-1}$. Thus, the total multiplication cost in Algorithm 5 is $\sum_{r=0}^{m-1} 2^{m-1} = \frac{1}{2} n \log_2 n$. Therefore, the total number of multiplications in the Gao–Mateer algorithm is given by $n \log_2 n - n + 1 + \frac{1}{2} n \log_2 n = \frac{3}{2} n \log_2 n - n + 1$.

From Algorithm 9, we know that in the r-th iteration, the number of additions due to the Taylor expansion is $2^{m-1}(m - r - 1)$. Thus, the total number of additions required for the Taylor expansions is $\sum_{r=0}^{m-2} 2^{m-1}(m - r - 1) = 2^{m-2} m(m-1)$. Also, in Algorithm 5, we know that at the r-th iteration, the number of required additions is $2 \cdot 2^r \cdot 2^{m-r-1} = 2^m$. Therefore, the total addition cost for the algorithm is given by $2^{m-2} m(m-1) + m \cdot 2^m = \frac{1}{4} n (\log_2 n)^2 + \frac{3}{4} n \log_2 n$.

When performing the FFT over a subspace, at the r-th iteration of Algorithm 5, we have $\eta_0 = 0$. Thus, for Steps 6 and 7, we need no multiplications, and only one addition is needed. Consequently, we must account for a reduction of $\sum_{r=0}^{m-1} 2^{m-r-1} = 2^m - 1 = n - 1$ in both additions and multiplications. Therefore, the costs for multiplications and additions are changed to $\frac{3}{2} n \log_2 n - 2n + 2$ and $\frac{1}{4} n (\log_2 n)^2 + \frac{3}{4} n \log_2 n - n + 1$, respectively.

4.4 Optimization for Cantor Special Basis

If we have a Cantor special basis of dimension m, we can avoid the scaling in every iteration. We know that a Cantor special basis satisfies the following

$$\beta_0 = 1 \quad \text{and } S(\beta_i) = \beta_i^2 + \beta_i = \beta_{i-1} \quad \text{for } 1 \le i \le m-1,$$

where $S(x) = x^2 + x$. In addition, we know that $S^i(\beta_i) = \beta_0 = 1$ for $0 \le i \le m-1$ and $S^{i+\ell}(\beta_{i+\ell}) = \beta_\ell$ for any $i, \ell \ge 0$ with $i + \ell \le m - 1$. Now, consider the Cantor special basis in the reversed order, i.e.,

$$W_m = \langle \beta_{m-1}, \beta_{m-2}, \ldots, \beta_1, 1 \rangle = \langle \beta_{m-1}, S(\beta_{m-1}), \ldots, S^{m-2}(\beta_{m-1}), 1 \rangle.$$

Thus, we have $G = \langle \beta_{m-1}, S(\beta_{m-1}), \ldots, S^{m-2}(\beta_{m-1}) \rangle$ and

$$D = \langle S(\beta_{m-1}), S^2(\beta_{m-1}), \ldots, S^{m-1}(\beta_{m-1}) \rangle = \langle S(\beta_{m-1}), S^2(\beta_{m-1}), \ldots, 1 \rangle.$$

Thus, we do not need the scaling for the functions $f_0(x)$ and $f_1(x)$. Also, at the j-th iteration, G and D will be of the form

$$G^{(j)} = \langle S^j(\beta_{m-1}), S^{j+1}(\beta_{m-1}), \ldots, S^{m-2}(\beta_{m-1})\rangle \quad \text{and}$$
$$D^{(j)} = \langle S^{j+1}(\beta_{m-1}), S^{j+2}(\beta_{m-1}), \ldots, S^{m-1}(\beta_{m-1}) = 1\rangle.$$

Therefore, at each iteration, there is no need for scaling. Also, due to the chosen basis, computing the basis elements in $G^{(j)}$ and $D^{(j)}$ does not require any multiplications or additions. This can be done simply by selecting one fewer element from $G^{(j-1)}$ and $D^{(j-1)}$.

Detailed cost analysis of the optimized algorithm When using the Cantor special basis in Algorithm 4, no scaling is required for the polynomials. All other steps in Algorithms 4 and 5 remain unchanged. Thus, the number of additions remains the same, as evaluated in Section 4.3. Consequently, the multiplication and addition costs are $\frac{1}{2}n\log_2 n$ and $\frac{1}{4}n(\log_2 n)^2 + \frac{3}{4}n\log_2 n$, respectively.

Furthermore, as discussed in Section 4.3, when performing the FFT over a subspace, the total multiplication and addition costs for the algorithm are $\frac{1}{2}n\log_2 n - n + 1$ and $\frac{1}{4}n(\log_2 n)^2 + \frac{3}{4}n\log_2 n - n + 1$, respectively. Thus, by using Cantor special basis in the Gao–Mateer algorithm, we can efficiently compute the additive FFT of $f(x) \in \mathbb{F}_{2^k}[x]$ over $\theta + W_m$ where $\mathbb{F}_{2^k}$ contains a subfield $\mathbb{F}_{2^{2^\ell}}$ with $m \leq 2^\ell$.

Note that the basis conversion (Expand Module) in the Gao–Mateer FFT involves scaling followed by a Taylor expansion with respect to $x^2 + x$. In the Cantor special basis, scaling is eliminated, reducing the multiplication cost; but the Taylor expansion remains. More efficient methods, such as that of Lin et al. [37], could improve this step, but would diverge from the original Gao–Mateer framework and yield a different FFT. We retain the original structure to allow fair comparison with the Cantor and LCH FFTs. Optimized conversions in this broader context are left for future work.

4.5 Precomputation

In this section, we introduce two levels of precomputation. The first level precomputes the scaling factors $\boldsymbol{\beta} = \{\beta_{0,m-1}, \beta_{1,m-2} \ldots, \beta_{m-1,0}\}$, basis vectors $\boldsymbol{\Gamma} = (\mathbf{G}_0 = \emptyset, \mathbf{G}_1, \ldots, \mathbf{G}_{m-1})$ and shift elements $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_{m-1})$. Each of $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$ requires m finite field elements and $\boldsymbol{\Gamma}$ needs $\sum_{i=0}^{m-1} i = m(m-1)/2$. Consequently, this algorithm stores a total of $(m^2+3m)/2$ finite field elements. Over $\mathbb{F}_{2^{256}}$ this translates to 2KB, 4.2KB, and 7.2KB for $m = 10, 15, 20$, respectively. This precomputation is inapplicable for the optimized Gao–Mateer algorithm for the Cantor special basis (see Section 4.4), because the scaling step is omitted and each basis vector reduces to $\mathbf{G}_r$ is simply $\{\beta_0, \beta_1, \ldots, \beta_{r-1}\}$, where the β_i are the Cantor special basis elements.

The second level precomputes all required powers of the scaling factors and all elements in each affine subspace $\theta_r + \mathbf{G}_r$. This requires storing $\sum_{r=0}^{m-1}(2^{m-r}-1) =$

$2^{m+1} - m - 2$ finite field elements for the powers of the scaling factors, and $\sum_{r=0}^{m-1} 2^r = 2^m - 1$ finite field elements for $\theta_r + \mathbf{G}_r$. Consequently, this algorithm stores a total of $3 \cdot 2^m - m - 3$ finite field elements. Over $\mathbb{F}_{2^{256}}$ this translates to 95.6KB, 3MB, and 96MB for $m = 10, 15, 20$, respectively. For the optimized Gao–Mateer algorithm for the Cantor special basis, the powers of the scaling factor are not required and the elements in $\theta + \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$ are computed, which equals 2^m finite field elements.

Table 4: Runtime speedup of the Gao–Mateer additive FFT level 1 (L1) versus level 2 (L2) precomputation methods over $\mathbb{F}_{2^{256}}$ (see Section 6 for platform details).

m	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
L1	11.5	6.41	3.81	2.48	1.80	1.46	1.29	1.20	1.12	1.07	1.06	1.05	1.05	1.06	1.06	1.05
L2	15.8	8.33	4.70	2.94	2.09	1.58	1.38	1.27	1.17	1.11	1.09	1.08	1.08	1.07	1.06	1.09

5 Aurora FFT Complexity Analysis

The FFT complexity of Aurora depends on the R1CS dimensions (d_1, d_2, d_3), the codeword domain size $|L|$, and the repetition parameters λ_i and λ'_i introduced in Section 2.4. While (d_1, d_2, d_3), λ_i, and λ'_i are fixed at setup, determining $|L|$ requires an iterative procedure since several constraints are mutually dependent.

Initialization. Set $|L| = 4t/\rho$.

Iterative check. Repeat the following steps until no parameter changes:

1. For the target security level $\boldsymbol{\lambda}$, the RS rate ρ, and the FRI localization parameter η, the required number of codeword queries is $\mathtt{b}$ (see Appendix B).
2. Given $\mathtt{b}$, the maximum lincheck degree is $d' = 2t + \mathtt{b} - 1$ (see Table 6).
3. The number of FRI reductions (rounds) is $\mathtt{r} := \lfloor \log(\rho|L|)/\eta \rfloor$. Consequently, $d' \equiv 0 \pmod{2^{\mathtt{r}\eta}}$. If not, replace d' with the next multiple of $2^{\mathtt{r}\eta}$.
4. If $d' > \rho|L|$, enlarge the domain by one dimension (i.e. set $|L| \leftarrow 2|L|$)

The loop terminates when $\mathtt{b}$ and $|L|$ stabilize; at that point, $|L|$ satisfies all constraints.

Now, we construct the evaluation domains H_1, H_2, and L. Let $\{\beta_0, \beta_1, \ldots, \beta_{k-1}\}$ be the basis of $\mathbb{F}_{2^k}$,

$$H_1 = \langle \beta_0, \beta_1, \cdots \beta_{\lceil \log d_1 \rceil} \rangle, \quad H_2 = \langle \beta_0, \beta_1, \cdots \beta_{\lceil \log(d_2+1) \rceil} \rangle,$$
$$L = \beta_{\lceil \log(|L|) \rceil + 1} + \langle \beta_0, \beta_1, \cdots \beta_{\lceil \log(|L|) \rceil} \rangle,$$

where L is an affine subspace that is disjoint from the linear subspaces H_1 and H_2; specifically, $L \cap (H_1 \cup H_2) = \varnothing$. If the basis elements are the Cantor special

basis, all three domains admit the Cantor additive FFT. Moreover, because the affine shift in L is a basis element, the special precomputation in Section 3.5 applies. Table 5 shows the FFT and IFFT calls in the Aurora prover algorithm.

Table 5: The FFT and IFFT calls in the Aurora zkSNARK protocol.

FFT Calls	**Description**
$\lambda_i\times$FFT of len. $\|L\|$	$\hat{\mathbf{r}}_\ell$: Evaluate $r_\ell(X)$ of degree $< 2t+b-1$ over L ($\ell = 1, \ldots, \lambda_i$).
1) IFFT of len. d_3+1 2) FFT of len. d_2+1 3) IFFT of len. d_2+1 4) FFT of len $\|L\|$	1) Interpolate $f_{(1,\mathbf{v})}(X)$ of degree $< d_3+1$. 2) Evaluate $f_{(1,\mathbf{v})}(X)$ over H_2 to compute $\mathbf{f}_{(1,\mathbf{v})}$. Then, compute $\mathbf{f}'_{\mathbf{w}} = \mathbf{w}[0 : d_2 - d_3 - 1] - \mathbf{f}_{(1,\mathbf{v})}[d_3+1 : d_2]$ 3) Interpolate $\mathbf{f}'_{\mathbf{w}}$ over H_2 to get $f'_{\mathbf{w}}(X)$ of degree $< d_2+1$. Then, divide $f'_{\mathbf{w}}(X)$ by $\mathbb{Z}_{\{h_0,\ldots,h_{d_3}\}}(X)$ to compute $f^*_{\mathbf{w}}(X)$. 4) Evaluate $f^*_{\mathbf{w}}(X)$ over L to compute $\hat{\mathbf{f}}_{\mathbf{w}}$.
1) 3×IFFT of len. d_1 2) 3×FFT of len. $\|L\|$	For $\mathbf{M} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$: 1) Interpolate $\mathbf{Mz}$ to get $f^*_{\mathbf{Mz}}(X)$ of degree $< d_1$. 2) Evaluate $f^*_{\mathbf{Mz}}(X)$, over L to compute $\hat{\mathbf{f}}_{\mathbf{Mz}}$.
$\lambda'_i\times$FFT of len. $\|L\|$	$\hat{\mathbf{r}}'_\ell$: Evaluate $r'_\ell(X)$ of degree $< 2t+2b$ over L ($\ell = 1, \ldots, \lambda'_i$).
$2\lambda_i\times$IFFT of len. t	Interpolate $p_{\alpha_\ell}(X)$ and $p^{ABC}_{\alpha_\ell}(X)$ ($\ell = 1, \ldots, \lambda_i$).
FFT of len. $\|L\|$	Evaluate $f_{(1,\mathbf{v})}(X)$ over L to compute $\hat{\mathbf{f}}_{\mathbf{z}} = \hat{\mathbf{f}}_{\mathbf{w}} \cdot (\mathbb{Z}_{\{h_0,\ldots,h_\kappa\}}(X)\|_L) + \hat{\mathbf{f}}_{(1,\mathbf{v})}$.
$2\lambda_i\times$FFT of len. $\|L\|$	Evaluate $p_{\alpha_\ell}(X)$ and $p^{ABC}_{\alpha_\ell}(X)$ over L to compute $\hat{\mathbf{q}}^{\mathbf{M}}_\ell :=$ $\hat{\mathbf{f}}_{\mathbf{Mz}} \cdot \hat{\mathbf{p}}_{\alpha_\ell} - \hat{\mathbf{f}}_{\mathbf{z}} \cdot \hat{\mathbf{p}}^{ABC}_{\alpha_\ell}$ ($\ell = 1, \ldots \lambda_i$ and $\mathbf{M} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$).
1) $\lambda_i\times$IFFT of len. d where $d = 2^{\lceil \log(t+b) \rceil}$ 2)$\lambda_i\times$FFT of len. $\|L\|$	1) Interpolate $\sum_{\mathbf{M}\in\{\mathbf{A},\mathbf{B},\mathbf{C}\}} s^{\mathbf{M}}_\ell \hat{\mathbf{q}}^{\mathbf{M}}_\ell$ to get a polynomial of degree $< 2^{\lceil \log(t+b) \rceil}$. Then, compute $h(X)$ (see Table 6). 2) Evaluate $h(X)$ over L to compute $\hat{\mathbf{h}}_\ell$.

6 Implementation and Benchmarking

With the goal of accelerating Aurora, we implemented the Cantor FFT in C++, making it compatible with the `libiop` library [9], which implements Aurora. We also converted the C implementation of the LCH FFT for subspaces over the Cantor special basis provided by [19] to C++, and extended it to support affine subspaces, when the affine shift is one of the basis elements and made it compatible with `libiop`. Additionally, we optimized the Gao–Mateer implementation by using the Cantor special basis. The `libff` library [11] is used for finite field operations in `libiop` as well as in our FFT implementations, enabling polymorphism over finite fields. Notably, the Cantor special basis must exist in a finite field in order to enable the use of FFT algorithms over that basis. We

computed and hardcoded the Cantor special basis for $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{192}}$, and $\mathbb{F}_{2^{256}}$. Our implementation of additive FFT algorithms is available on GitHub[1].

Using our FFT implementations, we extended the `libiop` library to support switching between the standard basis, which was originally used, and the Cantor special basis, which we hardcoded for $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{192}}$, and $\mathbb{F}_{2^{256}}$ and the dimensions less than 32. Consequently, the Gao–Mateer algorithm is used for the standard basis, while either our Cantor or LCH implementation is employed for the Cantor special basis. Our implementation of the accelerated `libiop` is available on GitHub[2].

Benchmark setup All measurements were taken with Google Benchmark [28], with a minimum 10-second warm-up period, on an AMD Ryzen 9 9950X @ 5.7 GHz, with 64 GB of DDR5 RAM and running Debian 12 with kernel 6.12.12.

Standalone FFT benchmark For each input size $n = 2^m$, the reported runtime of every additive FFT implementation is the mean of 1,000 independent evaluations when $m \leq 20$ and over 300 evaluations when $m > 20$. In each trial a polynomial $f(x) \in \mathbb{F}_{2^{256}}[x]$ of degree $< 2^m$ is sampled uniformly at random and evaluated on the affine subspace $W_m = \langle \beta_0, \ldots, \beta_{m-1} \rangle + \theta \subset \mathbb{F}_{2^{256}}$ where the shift $\theta \in \mathbb{F}_{2^{256}}$ is chosen uniformly at random for that trial. For the special case discussed in Section 3.5, the shift is fixed to $\theta = \beta_{31}$. The 99.9% confidence interval of the measurements is no greater than 1%.

Aurora benchmark We benchmark the Aurora protocol using three FFT implementations: Gao–Mateer, our Cantor implementation, and our C++ version of the LCH algorithm implemented in [37]. In this benchmarking, the multiplication factors in both Cantor and LCH are precomputed based on the method described in Section 3.5, and the coefficients of the vanishing polynomials in the Cantor algorithm are also precomputed. For each input size N, the reported runtime is the average proving and verifying time measured over 100 randomly generated satisfiable R1CS instances. Each R1CS instance uses $d_1 = N$, $d_2 = N - 1$, and $d_3 = 31$ (Definition 1). The codeword length is $|L| = 2^7 N$ by adopting Preon's choice of $\rho = 2^{-5}$ (Definition 2), to tighten the FRI soundness error [18], and two extra dimensions are added as described in Section 5. The 99.9% confidence interval of the measurements is no greater than 1%.

Comparisons Figure 1 compares the runtime of the FFT algorithms. It shows that the savings in Cantor PC are consistent across different input lengths. Table 3 presents the runtime improvements achieved through precomputations in the Cantor FFT. Table 4 shows the runtime gains from L1 and L2 precomputations in Gao–Mateer. However, these improvements become insignificant for larger m since Gao–Mateer requires extensive memory access for Taylor expansion, which suppresses the savings gained from the precomputations. Figure 3 in Appendix D depicts the fraction of each sub-algorithm in Gao–Mateer.

[1] https://github.com/mtbadakhshan/additive-fft
[2] https://github.com/mtbadakhshan/cantor-libiop

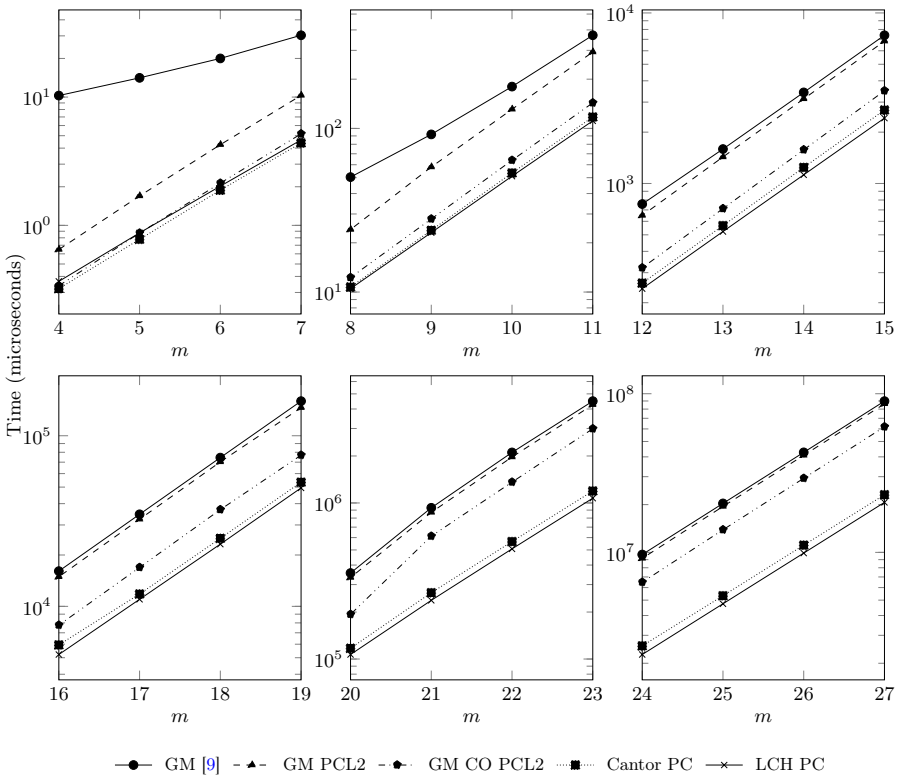


Fig. 1: Log-scale runtime of additive FFTs of length 2^m over $\mathbb{F}_{2^{256}}$ for Gao–Mateer (GM), GM with level-2 precomputation (GM PCL2), Cantor optimized GM with level-2 precomputation (GM CO PCL2), and Cantor with precomputation (Cantor PC), and LCH with special-case precomputation (LCH PC)

Note that the Gao–Mateer FFT with Cantor special basis, the LCH FFT with Cantor special basis, and the Cantor FFT require the same number of multiplications (see Table 1), but differ in their addition counts. The Gao–Mateer FFT uses $\frac{1}{4}n(\log_2 n)^2 + \frac{3}{4}n\log_2 n$ additions, while the Cantor FFT uses $\frac{1}{2}n\log_2 n + \frac{1}{2}n\sum_{r=0}^{\log_2(n)-1} 2^{\mathrm{wt}(r)}$, with $\mathrm{wt}(r)$ denoting the Hamming weight. No closed-form expression is known for the LCH FFT. Numerical evaluation shows that the Cantor FFT consistently requires fewer additions than the Gao–Mateer FFT for the m values in Figure 1; this, along with the basis conversion overhead in the Gao–Mateer algorithm, accounts for its slower performance. Furthermore, Figure 1 shows that, when used as standalone FFT algorithms, Cantor outperforms LCH for smaller dimensions (specifically, $m \leq 7$), while LCH exhibits better performance for $8 \leq m \leq 27$, though the performance difference remains marginal even at higher dimensions. It should also be noted that, similar to the Gao–Mateer FFT, the LCH FFT incurs a basis conversion overhead [37], while

the Cantor FFT completely avoids this by employing a pure divide-and-conquer structure.

Table 2 shows that integrating our optimized Cantor algorithm into the Aurora prover results in nearly twice the speed compared to using the Gao–Mateer FFT for large input sizes. It also outperforms our C++ integration of the LCH algorithm [19] into Aurora for smaller inputs. This performance differential highlights how the interaction between FFT algorithms and the underlying protocol structure significantly influences overall system efficiency, beyond what is captured in the standalone benchmarks. The observed slowdown in the LCH-based Aurora implementation for smaller input sizes is likely attributable to the memory overhead incurred by basis conversion. As a result, the Cantor FFT is more suitable for the circuit sizes commonly found in practical zkSNARK applications. For example, the Preon [18] proposes multiple security levels, where the number of constraints ranges from 2^{12} to 2^{14}. Additionally, constructing a circuit that proves knowledge of a Merkle tree leaf in a tree of 2^{30} elements using the POSEIDON-128 [29] hash function typically requires $2^{12} \leq N \leq 2^{13}$ constraints, depending on the arity of the Merkle tree.

7 Conclusions and Future Works

This work demonstrates how leveraging the Cantor special basis enables the integration of the Cantor and LCH additive FFTs into post-quantum secure zkSNARKs, focusing on Aurora [10]. We show that replacing the Gao–Mateer FFT with the Cantor and LCH additive FFTs significantly reduces computation time and Cantor is generally the best choice in typical zkSNARK applications. Our implementation is supported by a detailed cost analysis, including exact counts of additions and multiplications for both FFTs, and a complexity evaluation of FFT calls in Aurora's R1CS encoding, parameterized by constraints, variables, and the security level. We also propose optimized building blocks for the Cantor FFT and precomputation techniques that reduce overhead for both Cantor and Gao–Mateer FFTs when the affine subspace basis is fixed.

Building on the presented optimizations, several promising directions can be explored for future research. One possibility is optimizing other components of Aurora, such as the FRI protocol, or applying tower field constructions to accelerate field multiplications. Additionally, the proposed optimizations may extend to other post-quantum secure zkSNARKs over binary extension fields, such as STARK [4], which is used by zk-rollups and requires heavy CPU/GPU computation for proof generation [16].

Another direction is improving additive FFT throughput through parallelization. The Cantor FFT supports pure divide-and-conquer parallelism [45] as it does not require basis conversion. Thus, in hierarchical memory architectures, radix-4 or radix-8 variations could reduce rounds. Additive FFTs could also benefit from processing-in-memory (PIM) architectures, improving data locality like in multiplicative FFT [34].

Additionally, exploring more efficient basis conversion methods, such as replacing the Taylor expansion in Gao–Mateer FFT with the approach by Lin et al. [37], could reduce addition complexity, offering new FFT variants with better trade-offs. Comparing these with Cantor and LCH FFTs remains a valuable avenue. Finally, thorough side-channel analysis of additive FFT implementations is essential for security-critical applications such as post-quantum zkSNARKs and code-based cryptosystems, as demonstrated leakages in Gao–Mateer implementations reveal critical vulnerabilities [30,33].

Acknowledgments:

This work was supported by a MITACS–BTQ research grant. We sincerely appreciate the constructive feedback from the anonymous reviewers, which significantly enhanced the clarity and precision of this manuscript.

References

1. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. *Designs, Codes and Cryptography*, 91(11):3379–3424, Nov 2023.
2. Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly Verifiable Zero-Knowledge and Post-Quantum Signatures from VOLE-in-the-Head. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 581–615, Cham, 2023. Springer Nature Switzerland.
3. 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.
4. 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.
5. Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Scalable and Transparent Proofs over All Large Fields, via Elliptic Curves. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography*, pages 467–496, Cham, 2022. Springer Nature Switzerland.
6. Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Elliptic Curve Fast Fourier Transform (ECFFT) Part I: Low-degree Extension in Time $o(n \log n)$ over all finite fields. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 700–737, 2023.
7. Eli Ben-Sasson, Alessandro Chiesa, Michael A. Forbes, Ariel Gabizon, Michael Riabzev, and Nicholas Spooner. Zero Knowledge Protocols from Succinct Constraint Detection. In *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, page 172–206, Berlin, Heidelberg, 2017. Springer-Verlag.

8. Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.
9. 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. Accessed: 2025-01-10.
10. 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*, pages 103–128, Cham, 2019. Springer International Publishing.
11. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, Madars Virza, Howard Wu, Alexander Chernyakhovsky, and Aleksejs Popovs. libff: C++ library for finite fields and elliptic curves. https://github.com/scipr-lab/libff. Accessed: 2025-01-10.
12. Daniel J. Bernstein and Tung Chou. Faster Binary-Field Multiplication and Faster Binary-Field MACs. In Antoine Joux and Amr Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, pages 92–111, Cham, 2014. Springer International Publishing.
13. Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast Constant-Time Code-Based Cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 250–272, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
14. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards Privacy in a Smart Contract World. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 423–443, Cham, 2020. Springer International Publishing.
15. David G. Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, 1989.
16. Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. Analyzing and Benchmarking ZK-Rollups. In Rainer Böhme and Lucianna Kiffer, editors, *6th Conference on Advances in Financial Technologies (AFT 2024)*, volume 316 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:24, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS 2017, page 1825–1842, New York, NY, USA, 2017. Association for Computing Machinery.
18. 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, et al. Preon: zk-SNARK based Signature Scheme. *Technical report. NIST*, 2023.
19. Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster Multiplication for Long Binary Polynomials. arXiv: 1708.09746, 2018. https://arxiv.org/abs/1708.09746.
20. Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Multiplying boolean Polynomials with Frobenius Partitions in Additive Fast Fourier Transform. arXiv: 1803.11301, 2018. https://arxiv.org/abs/1803.11301.

21. 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.
22. Nicholas Coxon. Fast transforms over finite fields of characteristic two. *Journal of Symbolic Computation*, 104:824–854, 2021.
23. 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.
24. N. J. Fine. Binomial Coefficients Modulo a Prime. *The American Mathematical Monthly*, 54(10):589–592, 1947.
25. Shihui Fu and Guang Gong. Polaris: Transparent Succinct Zero-Knowledge Arguments for R1CS with Efficient Verifier. *Proceedings on Privacy Enhancing Technologies*, 2022.
26. Shuhong Gao and Todd Mateer. Additive Fast Fourier Transforms Over Finite Fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010.
27. Joachim von zur Gathen and Jürgen Gerhard. Arithmetic and factorization of polynomial over $\mathbb{F}_2$ (extended abstract). In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ISSAC 1996, page 1–9, New York, NY, USA, 1996. Association for Computing Machinery.
28. Google Inc. and contributors. Benchmark - a microbenchmark support library. https://github.com/google/benchmark. Accessed: 2025-01-10.
29. Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535. USENIX Association, August 2021.
30. Qian Guo, Andreas Johansson, and Thomas Johansson. A key-recovery side-channel attack on classic McEliece implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):800–827, Aug. 2022.
31. Tanmayi Jandhyala. Air-FRI: Acceleration of the FRI protocol on the GPU for low-degree polynomial testing in zk-SNARK applications. Master's thesis, University of Waterloo, Canada, 2024.
32. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS 2018, page 525–537, New York, NY, USA, 2018. Association for Computing Machinery.
33. Norman Lahr, Ruben Niederhagen, Richard Petri, and Simona Samardjiska. Side Channel Information Set Decoding Using Iterative Chunking: Plaintext Recovery from the "Classic McEliece" Hardware Reference Implementation. In *Advances in Cryptology – ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part I*, page 881–910, Berlin, Heidelberg, 2020. Springer-Verlag.
34. Dai Li, Akhil Pakala, and Kaiyuan Yang. MeNTT: A Compact and Efficient Processing-in-Memory Number Theoretic Transform (NTT) Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(5):579–588, 2022.
35. Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius Additive Fast Fourier Transform. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC 2018, page 263–270, 2018.

36. Sian-Jheng Lin, Tareq Y. Al-Naffouri, and Yunghsiang Sam Han. FFT Algorithm for Binary Extension Finite Fields and Its Application to Reed–Solomon Codes. *IEEE Transactions on Information Theory*, 62(10):5343–5358, 2016.
37. Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yunghsiang Sam 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.
38. Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang Sam Han. Novel Polynomial Basis and Its Application to Reed-Solomon Erasure Codes. *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 316–325, 2014.
39. Edouard Lucas. Théorie des fonctions numériques simplement périodiques. *American Journal of Mathematics*, 1(4):289–321, 1878.
40. Guiwen Luo, Shihui Fu, and Guang Gong. Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):358–380, Mar. 2023.
41. Todd Mateer. *Fast Fourier Transform Algorithms with Applications, PhD Thesis*. Clemson University, 2008. https://open.clemson.edu/all_dissertations/231/.
42. National Institute of Standards and Technology. Post-Quantum Cryptography: Standardization Process. https://csrc.nist.gov/projects/pqc-dig-sig/round-1-additional-signatures, 2023. Accessed: 2025-01-10.
43. Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 179–197, 2022.
44. Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain Scaling Using Rollups: A Comprehensive Survey. *IEEE Access*, 10:93039–93054, 2022.
45. Rob V. Van Nieuwpoort, Gosia Wrzesińska, Ceriel J. H. Jacobs, and Henri E. Bal. Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.*, 32(3), March 2010.
46. Yao Wang and Xuelong Zhu. A fast algorithm for the Fourier transform over finite fields and its VLSI implementation. *IEEE Journal on Selected Areas in Communications*, 6(3):572–577, 1988.
47. Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VeriZexe: Decentralized Private Computation with Universal Setup. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4445–4462, Anaheim, CA, August 2023. USENIX Association.

A The Cantor and Gao–Mateer FFT Algorithm

In this section, we present two additive FFT algorithms central to our contributions: the Cantor algorithm [15] and the algorithm by Gao and Mateer [26]. While Gao and Mateer propose two algorithms based on Taylor expansion for additive FFT, we focus on the first one, which applies to lengths $n = 2^m$ for any m. This algorithm is originally designed for the subspace $W_m = \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$, but we adapt it for use over the affine subspace $\theta + W_m$. Algorithms 6 and 7 provide the algorithmic formulations of the Cantor and Gao–Mateer algorithms, respectively.

Algorithm 6: Cantor additive FFT of length $n = 2^m$

Input: $f(x) \in \mathbb{F}_{2^k}[x]$ of degree $< n = 2^m$, where $k = 2^\ell$ and the affine subspace $\theta + W_m = \theta + \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$, where $\{\beta_0 = 1, \beta_1, \ldots, \beta_{m-1}\}$ is a Cantor special basis.
Output: $\mathrm{FFT}(f, \theta + W_m)$.

if $m = 0$ **then**
  return $f(\theta)$.
end
Compute

$$f_0(x) = f(x) \mod S^{m-1}(x + \theta), \text{ and}$$
$$f_1(x) = f(x) \mod S^{m-1}(x + \theta + \beta_{m-1}).$$

return $\mathrm{FFT}(f_0, \theta + W_{m-1}) || \mathrm{FFT}(f_1, \theta + \beta_{m-1} + W_{m-1})$.

Algorithm 7: Gao–Mateer additive FFT of length $n = 2^m$

Input: $f(x) \in \mathbb{F}_{2^k}[x]$ of degree $< n = 2^m$ and the affine subspace $\theta + W_m = \theta + \langle \beta_0, \beta_1, \ldots, \beta_{m-1} \rangle$
Output: $\mathrm{FFT}(f, \theta + W_m)$.

if $m = 1$ **then**
  return $(f(\theta), f(\theta + \beta_0))$.
end
else
  Compute $g(x) = f(\beta_{m-1} x)$
  Compute the Taylor expansion of $g(x)$ at $x^2 + x$ to obtain $f_0(x)$ and $f_1(x)$.
  Specifically, express $g(x)$ as $g(x) = f_0(x^2 + x) + x f_1(x^2 + x)$.
  Compute $\gamma_i = \beta_i \cdot \beta_{m-1}^{-1}$ and $\delta_i = \gamma_i^2 + \gamma_i$ for $0 \le i \le m - 2$.
  Let $\theta_0 = \beta_{m-1}^{-1} \theta$. Consider the affine subspaces

$$\theta_0 + G = \theta_0 + \langle \gamma_0, \ldots, \gamma_{m-2} \rangle \quad \text{and} \quad \theta_0^2 + \theta_0 + D = \theta_0^2 + \theta_0 + \langle \delta_0, \ldots, \delta_{m-2} \rangle.$$

  Let $\ell = 2^{m-1}$ and

$$\mathrm{FFT}(f_0, \theta_0^2 + \theta_0 + D) = (u_0, \ldots, u_{\ell-1}) \text{ and } \mathrm{FFT}(f_1, \theta_0^2 + \theta_0 + D) = (v_0, \ldots, v_{\ell-1}).$$

  For $0 \le i \le \ell - 1$, set $\omega_i = u_i + \eta_i \cdot v_i$ and $\omega_{\ell+i} = \omega_i + v_i$.
  return $(\omega_0, \ldots, \omega_{n-1})$.
end

B FRI Soundness Errors: Number of Queries Analysis

The FRI of proximity parameter is defined as

$$\delta := \min\left(\frac{1-2\rho}{2}, \frac{1-\rho}{3}, 1-\rho\right). \tag{2}$$

Given $\boldsymbol{\lambda}$, $\boldsymbol{\epsilon_q}$ and $\boldsymbol{\epsilon_i}$ represents query and interactive soundness errors such that $\boldsymbol{\epsilon_q} + \boldsymbol{\epsilon_i} < 2^{-\boldsymbol{\lambda}}$, where $2^{-\boldsymbol{\lambda}-1}$ is allocated to each. According to [10, Theorem 4],

$$\boldsymbol{\epsilon_i} = \left(\frac{d_1+1}{|\mathbb{F}|}\right)^{\lambda_i} + \left(\frac{|L|}{|\mathbb{F}|}\right)^{\lambda_i'} + \epsilon_i^{\mathrm{FRI}}, \text{ and } \boldsymbol{\epsilon_q} = \epsilon_q^{\mathrm{FRI}},$$

where $\left(\frac{\eta+1}{|\mathbb{F}|}\right)^{\lambda_i}$ and $\left(\frac{|L|}{|\mathbb{F}|}\right)^{\lambda_i'}$ denote the lincheck and LDT soundness errors respectively. $\epsilon_i^{\mathrm{FRI}}$ and $\epsilon_q^{\mathrm{FRI}}$ denote the interactive and query soundness errors in

FRI, respectively. Each term in $\boldsymbol{\epsilon_i}$ gets $2^{-\boldsymbol{\lambda}-3}$ and $\boldsymbol{\epsilon_q}$ gets $2^{-\boldsymbol{\lambda}-1}$ soundness error bound. The codeword is queried during FRI. Given the target proximity parameter in (2), the number of query repetitions in FRI is:

$$\lambda_q^{\text{FRI}} = \frac{\log(\epsilon_q^{\text{FRI}})}{\log\left(1 - \min\left(\delta, \frac{1-3\rho-2^{\eta}/\sqrt{|L|}}{4}\right)\right)}. \tag{3}$$

The number of queries to the codeword is set to $\mathtt{b} = \lambda_q^{\text{FRI}} \cdot 2^{\eta}$ to ensure zero-knowledge.

C Aurora Codewords

Table 6: The primary codewords encoded by prover. The mechanism for randomizing polynomials is omitted for the sake of simplicity.

Codeword	Description
$\hat{\mathbf{f}}_{\mathbf{w}} \in \text{RS}[L, \frac{d_2-d_3+\mathtt{b}}{\lvert L\rvert}]$	$\hat{\mathbf{f}}_{\mathbf{w}} := f^*_{\mathbf{w}}\lvert_L$, where $f^*_{\mathbf{w}}$ is a random polynomial of degree $< d_2 - d_3 + \mathtt{b}$ such that for $d_3 < i \leq d_2$, $f_{\mathbf{w}}(h_i) = (w_{i-d_3-1} - f_{(1,\mathbf{v})}(h_i))/\mathbb{Z}_{\{h_0,\ldots,h_{d_3}\}}(h_i)$ (w_i denotes the i-th element in $\mathbf{w}$).
$\hat{\mathbf{f}}_{\mathbf{Az}} \in \text{RS}[L, \frac{d_1+\mathtt{b}}{\lvert L\rvert}]$, $\hat{\mathbf{f}}_{\mathbf{Bz}} \in \text{RS}[L, \frac{d_1+\mathtt{b}}{\lvert L\rvert}]$, $\hat{\mathbf{f}}_{\mathbf{Cz}} \in \text{RS}[L, \frac{d_1+\mathtt{b}}{\lvert L\rvert}]$	$\hat{\mathbf{f}}_{\mathbf{Mz}} := f^*_{\mathbf{Mz}}\lvert_L$ for $\mathbf{M} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, where $f^*_{\mathbf{Mz}}$ is a random polynomial of degree $< d_1 + \mathtt{b}$ such that for $i = 0, \ldots, d_1 - 1$, $f_{\mathbf{Mz}}(h_i) = (\mathbf{Mz})_i$, where $(\mathbf{Mz})_i$ denotes the i-th element of the matrix–vector product $\mathbf{Mz}$.
$\hat{\mathbf{r}}_\ell \in_R \text{RS}[L, \frac{2t+\mathtt{b}-1}{\lvert L\rvert}]$	For $\ell = 1, \ldots, \lambda_i$, $\hat{\mathbf{r}}_\ell$ is a random lincheck masking codeword.
$\hat{\mathbf{h}}_\ell \in \text{RS}[L, \frac{t+\mathtt{b}}{\lvert L\rvert}]$, $\hat{\mathbf{g}}_\ell \in \text{RS}[L, \frac{t-1}{\lvert L\rvert}]$	For $\ell = 1, \ldots, \lambda_i$, $\hat{\mathbf{h}}_\ell := h_\ell\lvert_L$, where h_ℓ is a polynomial of degree $< t + b$ derived from the following polynomial division: $r_\ell(X) + \sum_{\mathbf{M}\in\{\mathbf{A},\mathbf{B},\mathbf{C}\}} s_\ell^{\mathbf{M}} \left(f_{\mathbf{Mz}}(X)p_{\alpha_\ell}(X) - f_{\mathbf{z}}(X)p^{\mathbf{M}}_{\alpha_\ell}(X)\right)$ $= g_\ell(X) + \frac{\sum_{a\in H_1\cup H_2} r_\ell(a)}{\sum_{a\in H_1\cup H_2} a^{t-1}} \cdot X^{t-1} + \mathbb{Z}_{H_1\cup H_2}(X) \cdot h_\ell(X)$
$\hat{\mathbf{r}}'_{\ell'} \in_R \text{RS}[L, \frac{2t+2\mathtt{b}}{\lvert L\rvert}]$	For $\ell' = 1, \ldots, \lambda'_i$, $\hat{\mathbf{r}}'_{\ell'}$ is a random LDT masking codeword.

D Additional Algorithms

Algorithm 8 outlines the procedures within the $\mathsf{Canopy}_{p,i}$ module, while Figure 2 illustrates how the polynomials for the subsequent round are derived from the quotient and remainder at each step.

We now present the Taylor expansion algorithm, which constitutes the core of the Expand module. Let $f(x) = \sum_{i=0}^{m-1} c_i x^i, c_i \in \mathbb{F}_q$, $\delta(x) = x^2 + x$. We denote $y = \delta(x)$. Then we may write f as

$$f(x) = f_0(y) + x f_1(y). \tag{4}$$

Algorithm 8: $\mathsf{Canopy}_{p,i}$ $(\mathbf{f}_{\text{in}}, \{\beta_0, \beta_1, \ldots, \beta_{m-1}\}, \mathbf{z}_{p-1}, \mathbb{Z}_{W_{p-1}}(\theta), p, i)$

Input: $\mathbf{f}_{\text{in}}$, $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ is the Cantor special basis, $\mathbf{z}_{p-1} = (\zeta_0, \zeta_1, \ldots, \zeta_{2^{\text{wt}(i)}-2})$, $\mathbb{Z}_{W_{p-1}}(\theta) \in \mathbb{F}_{2^k}$, $p = m - r$, and $i = (i_0, i_1, i_2, \ldots, i_r)$.
Output: $\mathbf{f}_{\text{out}}$.

$\psi_{i,r} \leftarrow \mathbb{Z}_{W_{p-1}}(\theta)$
for $j = 0$ **to** $r - 1$ **do**
  $\psi_{i,r} \leftarrow \psi_{i,r} + i_j \times \beta_{j+1}$
end
$\mathbf{f}_{\text{out}} \leftarrow$ Polynomial Division $(\mathbf{f}_{\text{in}}, \mathbf{z}_{p-1}, \psi_{i,r}, p, i)$ // Algorithm 2, where $\mathbb{Z}_{W_{p-1}}(\theta_{i,r}) = \psi_{i,r}$
return $\mathbf{f}_{\text{out}}$.

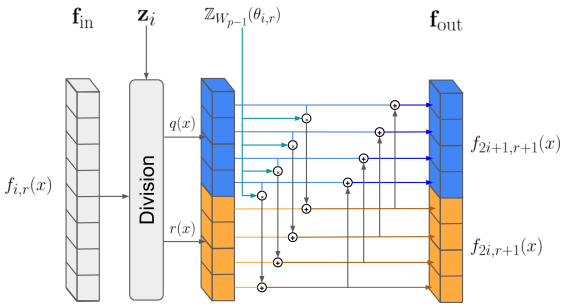


Fig. 2: Computing the polynomials for the next round from the outputs of dividing $f_{i,r}(x)$ (of degree < 8) by $\mathbb{Z}_{W_{p-1}}$, represented by $\mathbf{z}_{p-1}$.

Here, f_i, $i \in \{0, 1\}$, are polynomials in $\mathbb{F}_{2^k}$ with degree $< 2^{m-1}$. The representation of f by (4) is referred to as the Taylor expansion in [26]. The polynomials f_i can be obtained iteratively as follows. Let $\mathbf{f} = (c_0, \ldots, c_{n-1})$, referred to as the coefficient vector of $f(x)$, where $n = 2^m$, denoted by $f(x) \leftrightarrow \mathbf{f}$. Let $t = m - 2$, we define a *quadrant concatenation* of $\mathbf{f}$ as $\mathbf{f} = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$ where $c_i = (c_{i \cdot 2^t}, c_{i \cdot 2^t + 1}, \ldots, c_{(i+1) \cdot 2^t - 1})$. In the following, we model the Taylor expansion of f in terms of the concept of T_n module. The T_n module operation on f is defined as $\mathbf{b} = (\mathbf{c}_0, \mathbf{c}_1 + \mathbf{h}, \mathbf{h}, \mathbf{c}_3)$, where $\mathbf{h} = \mathbf{c}_2 + \mathbf{c}_3$. From $\mathbf{b} = (\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3)$, we have the following result.

Lemma 1. *With the above notation, we have*

$$f(x) = g_0(x) + g_1(x)y^{2^t}, \text{where } y = \delta(x) \tag{5}$$

where $g_0(x) \leftrightarrow (\mathbf{b}_0, \mathbf{b}_1)$ *and* $g_1(x) \leftrightarrow (\mathbf{b}_2, \mathbf{b}_3)$.

Now, to compute the Taylor expansion of $f(x) \in \mathbb{F}[x]$ of degree $< n = 2^m$, the Taylor expansion of $f(x)$ at $\delta(x) = x^2 + x$, denoted by $\mathsf{TE}(f, n, 2)$ can be computed through the Taylor expansions of two polynomials:

$$\mathsf{TE}(f, n, 2) = (\mathsf{TE}(g_0, \frac{n}{2}, 2), \mathsf{TE}(g_1, \frac{n}{2}, 2)) \tag{6}$$

where g_0 and g_1 are computed in Lemma 1 by T_n module.

Algorithm 9: Taylor Expansion $(\mathbf{f}_{\text{in}}, r)$

Input: $\mathbf{f}_{\text{in}} = (c_0, c_1, \ldots, c_{n-1})$ and r denotes the number of rounds reduction.
Output: $\mathbf{f}_{\text{out}}$.

$k \leftarrow m - 2$
while $k \geq r$ **do**
  $j \leftarrow 0$
  while $j \leq n - 4 \cdot 2^k$ **do**
    for $i = 0$ **to** $2^k - 1$ **do**
      $c_{2 \cdot 2^k + i + j} \leftarrow c_{2 \cdot 2^k + i + j} + c_{3 \cdot 2^k + i + j}$
      $c_{2^k + i + j} \leftarrow c_{2^k + i + j} + c_{2 \cdot 2^k + i + j}$
    end
    $j \leftarrow j + 4 \cdot 2^k$
  end
  $k \leftarrow k - 1$
end
return $\mathbf{f}_{\text{out}} \leftarrow (c_0, c_1, \ldots, c_{n-1})$

Theorem 2. *Let* $\mathbf{u} = (u_0, u_1, \ldots, u_{n-1})$ *be the output at the recursion* t *in (6), then the Taylor expansion of* f*, defined by (4), is given by*

$$f_0 \leftrightarrow (u_0, u_2, \ldots, u_{2i}, \ldots, u_{2\lfloor n/2 \rfloor})$$
$$f_1 \leftrightarrow (u_1, u_3, \ldots, u_{2i+1}, \ldots, u_{2\lfloor n/2 \rfloor + 1}),$$

where f_0 *and* f_1*, are polynomials each of degree* $< \lfloor n/2 \rfloor = 2^{m-1}$.

As described in the above theorem, the polynomials f_0 and f_1 are constructed by performing an even-odd rearrangement on $\mathbf{u}$. Subsequently, as described in Algorithm 7, the evaluations $\mathsf{TE}(f_0, n/2, 2)$ and $\mathsf{TE}(f_1, n/2, 2)$ also need to be computed and so on their outputs as well. Alternatively, as implemented in [9], instead of performing the even-odd rearrangements, the positions of the coefficients of these two polynomials can be retained within $\mathbf{u}$. This allows us to compute a single $\mathsf{TE}(u, n/2, 2)$ on the entire polynomial $u(x)$, represented by $\mathbf{u}$, in $t - 1$ recursive steps (one fewer recursion than required for f). Finally, to account for the skipped even-odd rearrangements, in the last round the resulting vector is rearranged in bit-reversal order as discussed below:

Define a recursive process that partitions the index set $\{0, 1, \ldots, 2^n - 1\}$ by the least significant bit (LSB): indices with LSB 0 form the even group, and those with LSB 1 form the odd group. This splitting is applied recursively to each group using the next least significant bit, continuing until each group contains exactly two indices. The resulting arrangement corresponds to the bit-reversal permutation of the original sequence.

Figure 3 illustrates the contribution of each sub-algorithm in the Gao–Mateer FFT implementation from [9]. The basis computations, which are precomputed in GM PCL1, require fewer resources as m increases. In contrast, the computational cost of bit-reversal rearrangement and Taylor expansion grows more rapidly for larger m, significantly impacting the overall FFT runtime.

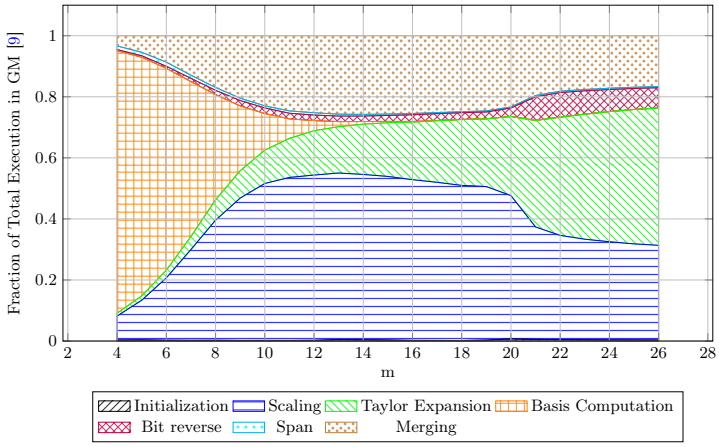
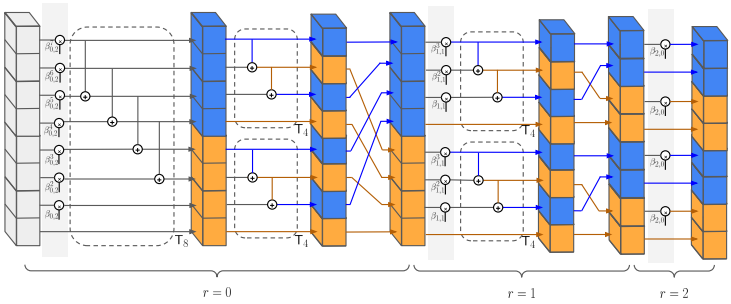
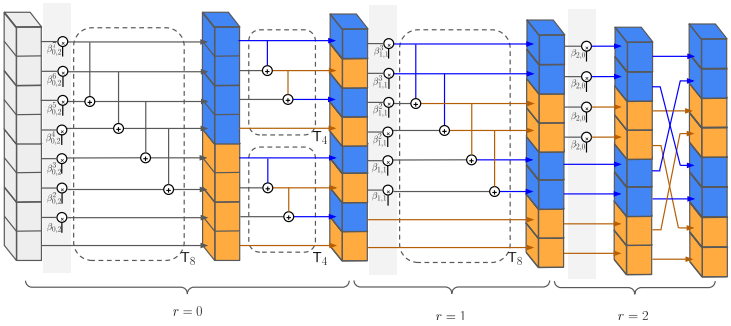


Fig. 3: The fraction of execution time for sub-algorithms in Gao–Mateer over $\mathbb{F}_{2^{256}}$ in [9]. The execution times of the Taylor expansion and bit-reversal sub-algorithms increase more rapidly with m. *Initialization* is a one-time computing for copying the input polynomial to a new vector. *Span* and *Merging* are the Span function and the nested for loop in the Aggregate Module.



(a) Original Approach: Groups even and odd indices after each Taylor expansion



(b) Alternative Approach: Bit-reversal rearrangement in the final round

Fig. 4: The Expand module in Gao–Mateer of length $n = 2^3$, which evaluates a polynomial $f(x) \in \mathbb{F}[x]$ of degree < 8 over $\theta + W_3$, where $W_3 = \langle \beta_0, \beta_1, \beta_2 \rangle$. In Figure (a), the Expand module groups even and odd indices after each Taylor expansion. In contrast, Figure (b) skips these rearrangements by using one T_8 module instead of two T_4 modules in round $r = 1$ then performs the bit-reversal in the final round.