Published at Progress in Cryptology – LATINCRYPT 2025 doi: 10.1007/978-3-032-06754-8_16

Rhizomes and the Roots of Efficiency—Improving Prio

Armando Faz-Hernandez

Cloudflare, Inc.
101 Townsend St, San Francisco CA 94107, USA
armfazh@cloudflare.com

Abstract. Prio, tailored under privacy-by-design principles, is a protocol for aggregating client-provided measurements between non-colluding entities. The validity of measurements is determined by using a fully linear probabilistically-checkable proof (FLPCP). The Prover distributes secret shares of the measurement and the proof to multiple Verifiers. These Verifiers can only use linear queries on the input statement for validation without accessing the actual measurement. Efficiency is key for the practical application of Prio. The FLPCP operates with polynomials represented in the Lagrange basis using roots of unity as the nodes. However, we observe opportunities to improve its performance by embracing the Lagrange basis more extensively. For instance, we show an inversion-free $O(n)$ time-complexity algorithm for polynomial evaluation in the Lagrange basis (an alternative to the classic rational barycentric formula). By applying our methods to libprio-rs, a cutting-edge Rust implementation, the Sharding phase (proof generation) runs a 36% faster and the Prep-Init phase (proof verification) is twice as fast, showing a substantial acceleration of the most time-consuming phases of Prio.

Keywords: Polynomial Basis · Lagrange Basis · Fully Linear Proofs · Distributed Aggregation Functions · Pólya Basis

1 Introduction

Calculating statistical functions on data generated by multiple users comes with the problem of leaking user's data points to collectors. Privacy-preserving techniques address this problem by allowing only the calculation of aggregated functions without revealing the individual contributions of each user. A recurrent use case is enabling service providers to run statistical measurements on its clients (e.g., to inspect the quality of the service) without learning any information about individual client measurements. Thus, protecting the privacy of users while still getting the aggregated values. Privacy-preserving measurements are a category of protocols that provide this functionality.

Under this category, Prio [16], introduced by Corrigan-Gibs and Boneh, is a cryptographic protocol that allows aggregation of some statistical functions. Other private aggregation algorithms exists, such as Prio+ [3], based on Boolean secret sharing, and Precio [4] based on oblivious shuffling. However, these protocols have not seen the same widespread use of Prio. In 2018, Mozilla [27,19] announced privacy-preserving telemetry collection for the Firefox browser. In 2020, ISRG [1] introduced a system infrastructure for private collection of metrics as well as continued support of libprio-rs, an open-sourced Prio's implementation. After 2020's global pandemic, the ENPA system [2,25], developed in conjunction by ISRG, Apple, Google, MITRE, and NIC, uses Prio for enabling health authorities to obtain epidemiology metrics from an exposure notification system. Cloudflare [24,34] has invested efforts in Prio's development. Davis et al. [17] showed that (with minor changes to) Prio meets formal security goals. Moreover, Prio is a candidate for standardization. The Privacy Preserving Measurement [29] working group at the Internet Engineering Task Force (IETF) has the goal of specifying protocols for the aggregation of statistics, and Prio is currently under consideration by the Crypto Forum Research Group [6].

Overview of Prio At a high level, Prio allows multiple non-colluding servers to aggregate measurements of multiple clients. The protocol, shown in Figure 1, starts with a client sending secret shares of its measurement to multiple servers. Each server, in possession of input shares from clients, is

limited to compute some functions (e.g., sum) on them to produce aggregate shares of the result. After that, the collector (or one of the servers) takes the aggregate shares from the participating servers and computes the final aggregate result. The protocol must be *private*, so the attacker learns nothing about the measurements, and *robust*, to prevent corruption of the aggregated result.

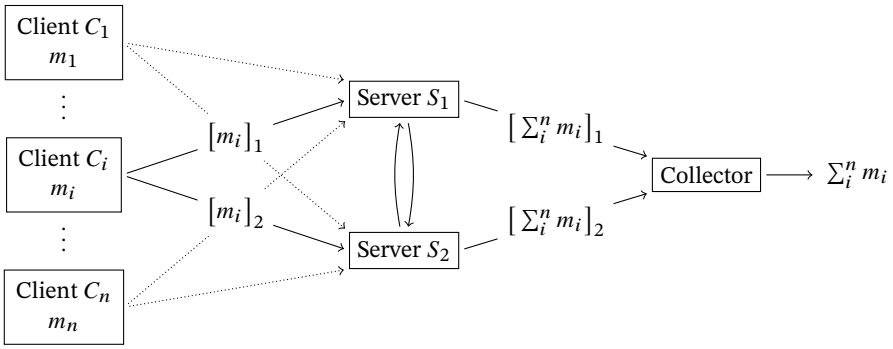


Fig. 1. Overview of the Prio protocol.

The protocol must prevent clients from entering invalid values that corrupt the aggregated value. To do so, clients produce a zero-knowledge proof that asserts the validity of the measurement. Thus, servers only aggregate measurements that have valid proofs. If the validity function is simple, e.g. checking a value is binary, specialized proof systems can be used instead of general-purpose proof systems. However in most proof systems, the input statement must be known for proof verification. For example, in a digital signature scheme, the message and the signature must be known during verification. In Prio, the server only has access to secret shares of the input measurement and the proof. For this reason, the proof system must make it possible to validate statements using only secret shares of the input *and* secret shares of the proof.

For validation of measurements, the validation function must be expressed as an arithmetic circuit over $\mathbb{F}$ with M multiplication gates. First, the client evaluates the circuit on its measurement x. Let u_t and v_t be the inputs of the multiplication gate number t, for $1 \leq t \leq M$. Then, the client defines three polynomials f, g, and h, such that $h = fg$, $f(t) = u_t$, and $g(t) = v_t$. Given x and h, the client uses an additive secret sharing function to produce k secret shares of $x = \sum_{i=1}^{k} [x]_i$ and $h = \sum_{i=1}^{k} [h]_i$, where k is the number of servers. Lastly, the client sends $([x]_i, [h]_i)$ to the ith server.

Upon receiving $([x]_i, [h]_i)$, the server recovers secret shares of $[f]_i$ and $[g]_i$, and performs the Schwartz-Zippel randomized polynomial identity test [39,43] to check whether $h = fg$; if so, the measurement is declared as valid. To do so, the server evaluates the secret-shared polynomials at $r \leftarrow_{\$} \mathbb{F}^*$ chosen at random to obtain $[f(r)]_i$, $[g(r)]_i$, and $[h(r)]_i$. Using these values, each server calculates $\sigma_i = [f(r) \times g(r) - h(r)]_i$, where $\times$ is a protocol for multiplication of secret shares. Finally, every server publishes its σ_i value and checks whether $\sum_{i=0}^{k} \sigma_i = 0$, if this check passes the measurement is valid for aggregation.

A Seed of Motivation The authors of Prio [16] proposed an optimization for performing verification without using interpolation. Their observation is that a polynomial P in the Lagrange basis can be evaluated at r in linear time whenever r is fixed and known in advance, instead of sampling it at random each time.

To see why this works, it is known that any polynomial P of degree n can be written as $P(x) = \sum_{i=0}^{n} y_i \ell_i(x)$, where y_i are the values and $\ell_i(x)$ are the Lagrange polynomials. Calculating the set of $\ell_i(x)$ requires $O(n^2)$ operations. However if they are precomputed, evaluating $P(r)$ takes $O(n)$ operations accounting for the dot product between $\ell_i(r)$ and the values y_i sent by the client.

In Prio, the value r must be a chosen at random, otherwise it affects the soundness of the protocol. If the server fixes the value r, then it must be guaranteed that the client never learns r, and that the server samples r frequently. The substantial overhead in computation and, more critically, in the security of the protocol presents a significant barrier for applying this optimization. This paper addresses these limitations and details a successful approach for implementing this (and other) optimizations.

Contributions Motivated by the traction gained in the past few years, its use on practical applications, and its path towards standardization, our focus centered on ways to reduce the computational cost of Prio.

- We show an inversion-free $O(n)$ time-complexity algorithm for polynomial evaluation in the Lagrange basis. Our algorithm is not derived from the well-known barycentric formula. It relies on the use of *Pólya polynomials* [8], whose origin traces back to a probabilistic model known as the Pólya urn. These polynomials are in close connection with B-splines and have been proven useful in computer-aided geometric design [9].
- A direct consequence of the above is showing a new application of the Pólya polynomials in the field of cryptography. Our algorithm enables that Prio's verification runs in linear time, without calculating inverses, and without making extra security assumptions on the server's sampled values.
- We study the basis extension operation for compactly storing polynomials and for polynomial multiplication. In both cases, we present explicit formulas and optimizations that are of general interest when working with polynomials in the Lagrange basis.
- We implement the proposed algorithms in (a fork of) libprio-rs, a state-of-the-art open-sourced Rust implementation of Prio. We plan to contribute the changes proposed in this investigation to the library maintainers as well as to the authors of the IETF specification document [6].
- At the Prio protocol level, we observe significant savings in the execution time of the Sharding phase (when the client generates a FLPCP proof) and during the Prep-Init phase (when the server verifies the proof).

To process polynomial arithmetic efficiently, we make an effort to update Prio to use the Lagrange representation for all polynomial arithmetic. In doing so, we faced several challenges including the change of representations, the basis extension, and the polynomial multiplication. We remark that while some standard techniques for Lagrange basis are already in place, we show more opportunities for improving the execution time of the protocol.

We discovered significant performance improvements by a careful analysis of Prio's operations. We found it fruitful to explore methods and solutions developed in other fields. For instance, we found abundant polynomial representations from numerical analysis, particularly those used in computer-aided geometric design, proved invaluable to our approach. However, these representations are less popular in the cryptographic community. This cross-disciplinary inspiration was a key factor in achieving our results.

The diversity of optimizations goals is a limitation for connecting fields of research. In numerical analysis, a primary goal is to reduce errors caused by ill-conditioned functions or the loss of precision during computation. These errors often arise when working with infinite domains like real and complex numbers. Consequently, algorithms in this field are designed to minimize errors first, with the reduction of computational operations being a secondary concern. In cryptography, most algorithms operate within finite domains, ensuring all calculations are exact and free of rounding errors or loss of precision. After guaranteeing security, common optimization goals in cryptography include improving the efficiency and security of implementations, as well as protecting against side-channel attacks. Despite their optimization goals differ, combining knowledge from both areas has yielded useful benefits. This work introduces new ideas from numerical analysis methods

and shows how to apply them effectively to reduce the number of operations of a cryptographic protocol.

2 Background

2.1 The Roots of Unity

Let $\mathbb{K}$ be a field, for any $x \in \mathbb{K}$ is said to be an *Nth-root of unity* if $x^N = 1$ over $\mathbb{K}$ for $N > 0$. The Nth-roots of unity in $\mathbb{K}$ forms a cyclic group of order N under multiplication and is generated by a primitive Nth-root of unity ω_N.

From now on, N denotes a positive power of two, and $\mathbb{K}$ is restricted to be a finite field $\mathbb{F}$. The Nth-roots of unity have really useful properties:

$$\begin{aligned} &\text{Sum:} && \sum_{i=0}^{N-1} {\omega_N}^i = 0\,. \\ &\text{Inverse:} && {\omega_N}^{-j} = {\omega_N}^{N-j}\,. \\ &\text{Negative:} && -{\omega_N}^j = {\omega_N}^{j+\frac{N}{2}}\,, \text{ for } j > 0\,. \\ &\text{Exponent:} && (\omega_{jN})^{jk} = (\omega_N)^k\,, \text{ for } j, k > 0\,. \end{aligned} \tag{2.1.1}$$

Figure 2 shows the relation of the Nth-roots of unity for the first powers of two.

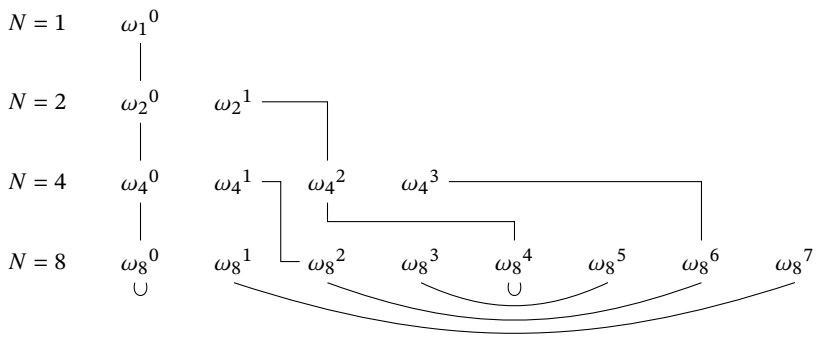


Fig. 2. Each row shows the Nth-roots of unity. Straight lines connecting the roots show that ${\omega_N}^j = {\omega_{2N}}^{2j}$, and curved lines show that ${\omega_N}^{-j} = {\omega_N}^{N-j}$ for $j \geq 0$.

It is possible to ensure the presence of the Nth-roots of unity in $\mathbb{F}$ [35]. Let p be the characteristic of $\mathbb{F}$, then any p of the form[1] $p = f2^k+1$, such that f is odd and $0 < f < 2^k$, guarantees the existence of subgroups of order 2^i in $\mathbb{F}^*$, for all $0 \leq i \leq k$. Each subgroup corresponds to the 2^ith-roots of unity and is generated by $\omega_{2^i} = g^{f2^{(k-i)}}$ for all $0 \leq i \leq k$, where g is a generator of $\mathbb{F}^*$.

2.2 Polynomials as a Vector Space

We describe well-known forms of representing polynomials: the monomial basis and the Lagrange basis. To facilitate the explanation of our methods, we opted to use vector spaces. This comes in handy, for example, when describing a change of polynomial representations (as a change of basis) rather than recurring to polynomial interpolation.

[1] This family of primes is also known as Proth primes [33].

Vector Space A vector space V over a field $\mathbb{F}$ is a set of elements where vector addition and scalar multiplication operations are defined. A set $B \subset V$ is a basis of V if every element $v \in V$ can be expressed as a linear combination of the elements in B. The coefficients of this linear combination are known as the coordinates of v with respect to the base B and are denoted as $[v]_B$. Every basis of V has the same number of elements, which is the dimension of V. We denote by $V_{[d]}$ the vector space of dimension d and by B_d its basis. It holds that $V_{[i]} \subsetneq V_{[d]}$, for all $0 < i < d$.

Let $k \in \mathbb{F}$, and $u, v \in V_{[d]}$ be two vectors of with coordinates $(u_0, \dots, u_{d-1})$ and $(v_0, \dots, v_{d-1})$, respectively; then vector addition and scalar multiplication is defined as

$$\begin{aligned} +\colon\ V \times V &\to V \\ \big((u_0, \dots, u_{d-1}), (v_0, \dots, v_{d-1})\big) &\mapsto (u_0 + v_0, \dots, u_{d-1} + v_{d-1})\,, \\ \cdot\colon\ \mathbb{F} \times V &\to V \\ \big(k, (u_0, \dots, u_{d-1})\big) &\mapsto (ku_0, \dots, ku_{d-1})\,. \end{aligned} \tag{2.2.1}$$

Polynomials Let $\mathbb{F}[x]$ denotes the univariate polynomial ring over a field $\mathbb{F}$. Elements of this ring are called *polynomials*. The set of polynomials also behaves as a vector space over $\mathbb{F}$ of countable-infinite dimension. The basis elements of $\mathbb{F}[x]$ are also polynomials. For finite dimension, $\mathbb{F}[x]_{[d]}$ is a d-dimensional vector space of polynomials, and they are compactly stored as a tuple of its coordinates. Evaluating a polynomial $P \in \mathbb{F}[x]$ at a given value z, denoted as $P(z)$, is the value calculated from the resulting expression after replacing x with z.

Monomial Basis The set $\mathcal{M} = \{x^0, \dots, x^{n-1}\}$ is the *monomial basis* of $\mathbb{F}[x]_{[n]}$. So any $P \in \mathbb{F}[x]_{[n]}$ is represented as

$$P(x) = \sum_{i=0}^{n-1} c_i x^i, \text{ where } c_i \in \mathbb{F}\,. \tag{2.2.2}$$

The c_i values are the coordinates of P simply known as its coefficients. The degree of P is the largest exponent of the basis element with a non-zero coefficient. $\mathbb{F}[x]_{[n]}$ contains all polynomials of degree lesser than n.

Lagrange Basis Given a set of $n+1$ distinct values $x_0, \dots, x_n \in \mathbb{F}$, called *nodes*, the Lagrange polynomials are defined as

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}\,, \text{ for } 0 \leq i \leq n\,. \tag{2.2.3}$$

Each of them has degree n, and it holds that $\ell_i(x_i) = 1$, and $\ell_i(x_j) = 0$ if $i \neq j$, for $0 \leq i, j \leq n$.

The set $\mathcal{L} = \{\ell_0(x), \dots, \ell_n(x)\}$ is the *Lagrange basis* of $\mathbb{F}[x]_{[n+1]}$, and any $P \in \mathbb{F}[x]_{[n+1]}$ is represented as

$$P(x) = \sum_{i=0}^{n} y_i \ell_i(x)\,, \text{ where } y_i \in \mathbb{F}\,. \tag{2.2.4}$$

The y_i values are the coordinates of P called the *values* (or ordinates) of P. The evaluation of P at the nodes corresponds to the values, i.e., $y_i = P(x_i)$ for $0 \leq i \leq n$. For storing P use a tuple containing the pairs (x_i, y_i) for $0 \leq i \leq n$, or one can store only the values when the nodes are well-known.

Change of Basis Switching between polynomial basis can be expressed with matrix operations. Gander [22] showed explicit vector-matrix products to change between several polynomial basis. In particular, the monomial and Lagrange basis are related as $V^T\mathcal{M} = \mathcal{L}$, and the coefficient and values as $V[P]_{\mathcal{M}} = [P]_{\mathcal{L}}$, where V is a Vandermonde matrix with entries $V_{i,j} = {x_i}^j$ for $0 \leq i, j \leq n$. The inverse transformation is possible since V is invertible whenever the nodes are distinct. In the general case, changing between basis of dimension n takes $O(n^2)$ field operations, which is the computational cost of a vector-matrix product.

Basis Extension Let $\mathbb{F}[x]_{[n]}$ and $\mathbb{F}[x]_{[m]}$ of dimension $n < m$, and let B_n and B_m be its basis, respectively. For any $P \in \mathbb{F}[x]_{[n]}$, the *basis extension*[2] of P is a function that given $[P]_{B_n}$, it finds $[P]_{B_m}$. That is, it produces an equivalent set of coordinates that represents the same polynomial in a basis of larger dimension; this is always possible since $\mathbb{F}[x]_{[n]} \subsetneq \mathbb{F}[x]_{[m]}$.

In the monomial basis, basis extension is easy by setting the higher-degree coefficients to zero, i.e., $c_i = 0$ for $n < i \leq m$. In the Lagrange basis, extending a polynomial in basis $\mathcal{L}_n$ to $\mathcal{L}_{n+1}$ takes $O(n)$ field operations [38]. Hence, in the general case, it takes $O(nm)$ field operations to arbitrarily extend polynomials in the Lagrange basis. In Section 3.3, we show faster algorithms for basis extension for certain special cases of interest.

Polynomial Multiplication There is a substantial gap in the computational complexity of polynomial multiplication [42]. For an n-dimensional basis, the standard algorithm in the monomial basis takes $O(n^2)$ field operations[3], in contrast to the $O(n)$ field operations performed while working in the Lagrange basis. To achieve this lower complexity, it requires input polynomials of the correct dimension. If dimensions are wrong, a basis extension is needed and must be factored in.

For any $P \in \mathbb{F}[x]_{[n]}$ and $Q \in \mathbb{F}[x]_{[m]}$, we have that $P \times Q \in \mathbb{F}[x]_{[n+m-1]}$. Let $(p_0, \dots, p_{n-1})$ and $(q_0, \dots, q_{m-1})$ be the coordinates of P and Q. In the monomial basis, the coefficients of $P \times Q$ are calculated as

$$\big((p_0, \dots, p_{n-1}), (q_0, \dots, q_{m-1})\big) \mapsto (r_0, \dots, r_{n+m-2}), \tag{2.2.5}$$

where $r_{i+j} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} p_i q_j$ for $0 \leq i < n$ and $0 \leq j < m$. In the Lagrange basis, the values of $P \times Q$ are calculated as

$$\big((p_0, \dots, p_{n+m-2}), (q_0, \dots, q_{n+m-2})\big) \mapsto (p_0 q_0, \dots, p_{n+m-2} q_{n+m-2}). \tag{2.2.6}$$

This algorithm holds under the assumption that P and Q were previously extended to a basis of dimension $n+m-1$. If this is the case, polynomial multiplication takes $O(n)$ operations. Otherwise, input polynomials must initially be extended to a higher dimension, an operation requiring $O(n^2)$ operations in the general case, yielding the same complexity as the monomial basis algorithm. Fortunately when restricting the nodes to some special values enables faster $O(n \log n)$ methods for polynomial multiplication. In Section 3.3, we present these methods and demonstrate how to optimize execution speed by reducing the constant factors behind the asymptotic notation.

Polynomial Evaluation There exist efficient algorithms for polynomial evaluation. Horner's method [28] is an $O(n)$ time-complexity algorithm for evaluating polynomials represented in the monomial basis and is shown in Algorithm 1. For the Lagrange basis, directly calculating Eq. (2.2.4) takes $O(n^2)$ field operations. That is, evaluating a dot product between the values and the Lagrange polynomials. However, one can derive an asymptotically faster algorithm from the

[2] This extension is sometimes denominated as degree extension. However, this term may not be the best since the degree of the polynomial must remain unchanged.

[3] Asymptotically faster algorithms for polynomial multiplication exist: Karatsuba [30] algorithm takes $O(n^{\log(3)/\log(2)})$ operations, and Toom-Cook [13,40] algorithm takes $O(n^{\log(5)/\log(3)})$ operations.

barycentric[4] formula [38]. In Section 3.1, we show an alternative $O(n)$ time-complexity algorithm for polynomial evaluation in the Lagrange basis.

Barycentric Algorithm Given a polynomial $P \in \mathbb{F}[x]_{[n]}$ represented by its set of nodes and values, (x_i, y_i) for $0 \leq i \leq n$, we can evaluate $P(x)$ using the second form of the barycentric formula as

$$P(x) = \frac{\sum_{i=0}^{n} \frac{w_i}{x - x_i} y_i}{\sum_{i=0}^{n} \frac{w_i}{x - x_i}}, \quad \text{where } w_i = 1/\prod_{\substack{j=0 \\ j \neq i}}^{n} (x_i - x_j) \in \mathbb{F}. \tag{2.2.7}$$

If all the w_i scalars are computed in advance (a task taking $O(n^2)$ field operations), then evaluating the barycentric formula takes $O(n)$ field operations, as shown in Algorithm 2.

Algorithm 1 Horner's Algorithm for Polynomial Evaluation in $\mathcal{M}$ Basis

Input: $[P]_{\mathcal{M}} = (c_0, \ldots, c_n)$, and $x \in \mathbb{F}$.
Output: $P(x)$.

1: $y = c_n$
2: **for** $i = n - 1$ **to** 0 **do**
3: $\quad y = yx + c_i$
4: **end for**
5: **return** y

Algorithm 2 Barycentric Algorithm for Polynomial Evaluation in $\mathcal{L}$ Basis

Input: $[P]_{\mathcal{L}} = (y_0, \ldots, y_n)$, and $x \in \mathbb{F}$.
Constant: $x_0, \ldots, x_n$.
Precompute: $w_0, \ldots, w_n$.
Output: $P(x)$.

1: $u = 0$, and $v = 0$
2: **for** $i = 0$ **to** n **do**
3: $\quad t = w_i (x - x_i)^{-1}$
4: $\quad u = u + t y_i$
5: $\quad v = v + t$
6: **end for**
7: **return** u/v

Let **A**, **M**, **S**, and **I** represent the cost of calculating, respectively, additions (or subtractions), multiplications, squares, and multiplicative inverses over $\mathbb{F}$. The number of field operations for evaluating a polynomial $P \in \mathbb{F}[x]_{[n]}$ is:

$$\begin{aligned} \text{Horner Algorithm:} \quad & n\mathbf{M} + n\mathbf{A} \\ \text{Barycentric Algorithm:} \quad & 2(n+1)\mathbf{M} + 3(n+1)\mathbf{A} + (n+2)\mathbf{I} \end{aligned} \tag{2.2.8}$$

Despite both algorithms having $O(n)$ time-complexity, the practical efficiency is heavily influenced by the cost of field operations. Inverses, for example, are generally more computationally expensive than multiplications and additions, which significantly impacts the overall execution time. Consequently, developing an inversion-free algorithm is a crucial goal for improving efficiency.

2.3 The Roots of Unity as the Nodes of the Lagrange Basis

A number of simplifications and faster algorithms result from choosing the Nth-roots of unity as the nodes in the Lagrange basis representation. We use $\mathcal{L}\mu$ to denote this choice and distinguish it from the general case $\mathcal{L}$.

Polynomial Evaluation A classic result, often found in some textbooks [37,36], is that any $P \in \mathbb{F}[x]_{[N]}$ represented in $\mathcal{L}\mu$ can be expressed as

$$P(x) = \frac{x^N - 1}{N} \sum_{i=0}^{N-1} \frac{{\omega_N}^i y_i}{x - {\omega_N}^i}. \tag{2.3.1}$$

[4] A barycentric combination is a weighted sum of points where weights sum to one. See [20, Ch. 2] for a description on the origin of the barycentric term.

This holds, in part, because the following calculations are simplified:

$$w_i = \prod_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{1}{(\omega_N{}^i - \omega_N{}^j)} = \frac{\omega_N{}^i}{N}, \text{ and } \prod_{i=0}^{N-1}(x - \omega_N{}^i) = x^N - 1\,. \tag{2.3.2}$$

Based on the above, Alg. 3 evaluates a polynomial taking $(2N+2)\mathbf{M}+\log_2(N)\mathbf{S}+(2N+1)\mathbf{A}+(N+1)\mathbf{I}$. This requires a similar number of operations as the general case, but without precomputation.

Algorithm 3 Barycentric Algorithm for Polynomial Evaluation in $\mathcal{L}\mu$ Basis

Input: $[P]_{\mathcal{L}\mu} = (y_0, \ldots, y_{N-1})$, and $x \in \mathbb{F}$.
Constant: $\omega_N{}^0, \ldots, \omega_N{}^{N-1}$.
Output: $P(x)$.

$u = 0$
for $i = 0$ **to** $N-1$ **do**
  $t = \omega_N{}^i (x - \omega_N{}^i)^{-1}$
  $u = u + t y_i$
end for
$r = x$
for $i = 0$ **to** $\log_2(N) - 1$ **do**
  $r = r^2$
end for
return $(r-1)uN^{-1}$

Algorithm 4 NTT Algorithm

Input: ω_N, and $U = (u_0, \ldots, u_{N-1})$.
Output: $\texttt{NTT}(\omega_N, U)$.

$V_i = U_{\texttt{bitrev}(i, \log_2(N))}$ for $0 \le i < N$.
for $l = 1$ **to** $\log_2(N)$ **do**
  $w = 1$ {★}
  $y = 2^{l-1}$
  $r = \omega_{2^l}$
  for $i = 0$ **to** $y-1$ **do**
    for $j = 0$ **to** $N/2y - 1$ **do**
      $x = 2^l j + i$
      $u = V_x$
      $v = wV_{x+y}$
      $V_x = u + v$
      $V_{x+y} = u - v$
    end for
    $w = wr$
  end for
end for
return V

Faster Change of Basis (i.e., Fast Fourier Transform) The Faster Fourier Transform (FFT) is an $O(N \log N)$ time-complexity algorithm, independently proposed by Cooley-Tukey [14] and Gentleman-Sandle [23], for the Discrete Fourier Transform over $\mathbb{K}$. When it is specialized to $\mathbb{F}$, it is known as the Number Theoretic Transform (NTT).

Changing between the monomial and Lagrange basis is asymptotically faster by using NTT. Let $(c_0, \ldots, c_{N-1})$ be the coefficients of $P \in \mathbb{F}[x]_{[N]}$, the NTT algorithm is defined as

$$\texttt{NTT}\colon \big(\omega_N, (c_0, \ldots, c_{N-1})\big) \mapsto \big(P(\omega_N{}^0), \ldots, P(\omega_N{}^{N-1})\big)\,. \tag{2.3.3}$$

Thus, it holds that $[P]_{\mathcal{L}\mu} = \texttt{NTT}(\omega_N, [P]_{\mathcal{M}})$. NTT also makes it possible to obtain the coefficients from the values. The inverse NTT takes the values $(y_0, \ldots, y_{N-1})$ and calculates the coefficients as

$$\texttt{iNTT}\colon \big(\omega_N, (y_0, \ldots, y_{N-1})\big) \mapsto \tfrac{1}{N}\texttt{NTT}\big(\omega_N{}^{-1}, (y_0, \ldots, y_{N-1})\big)\,. \tag{2.3.4}$$

Algorithm 4 implements NTT taking $N\log_2(N)(\mathbf{M}+\mathbf{A})$. Since NTT takes $O(N \log N)$ operations, switching between the $\mathcal{M}$ and $\mathcal{L}\mu$ basis is asymptotically faster than the $O(N^2)$ algorithm for switching between the $\mathcal{M}$ and $\mathcal{L}$ basis.

3 Improvements for Lagrange Polynomial Basis $\mathcal{L}\mu$

We propose algorithms that use the Nth-roots of unity as Lagrange basis nodes to reduce the operations needed for polynomial evaluation and basis extension. These algorithms have utility beyond the specific optimization problem addressed in this work.

3.1 Inversion-free Linear-Time Polynomial Evaluation in Lagrange Basis $\mathcal{L}\mu$

We show an $O(n)$ time-complexity algorithm for polynomial evaluation, an inversion-free alternative to the barycentric formula. We first discuss limitations of the barycentric formula's implementation that motivated our algorithm.

Motivation. In Section 2.3, we show that polynomials in the Lagrange basis can be evaluated with a linear number of operations using either Algorithm 2 in the general case, or Algorithm 3 over a field where the Nth-roots of unity exist. Both algorithms calculate $(N+1)\mathbf{I}$. For finite fields[5], inverses are more expensive to compute and are often avoided. Given $a \in \mathbb{F}^*$, its inverse is often calculated in constant-time as $a^{-1} = a^{q-2}$, where q is the number of elements of $\mathbb{F}$. Thus, the ratio $\mathbf{I}/\mathbf{M}$ grows with the size of the field, which potentially dominates the execution time of polynomial evaluation for large fields.

Two well-known methods for reducing inversions in the barycentric formula calculation are immediately apparent. The Montgomery's trick [31] is a classic technique for calculating $n\mathbf{I}$ inverses at the expense of $3(n-1)\mathbf{M} + \mathbf{I}$ operations. A different approach is to express the barycentric formula as an algebraic fraction, i.e., as the ratio of two values in $\mathbb{F}$. So all the calculations use fraction arithmetic, with a single inversion at the end for the final fraction value.

Problem. Unfortunately, neither approach completely eliminates the calculation of inverses. The cost of polynomial evaluation in large fields can be dominated by even one inverse calculation. Is it possible to develop an inversion-free algorithm for polynomial evaluation in the Lagrange basis $\mathcal{L}\mu$?

Solution. Our response is affirmative. We observe that Horner's algorithm evaluates a polynomial without inverses. In numerical analysis [12,26], Horner's algorithm is described as a nested multiplication algorithm, and some textbooks use ladder diagrams to visualize its flow of operations. Inspired by this structure, we looked for an algorithm with a similar approach and found Warren's algorithm [41] that evaluates polynomials represented in the *Pólya basis*. Consequently, the Pólya polynomial basis, introduced by Barry and Goldman [8,10] as a way to generalize the Bézier and Lagrange polynomials, served us as a starting point. We then analyze the operations of Warren's algorithm and explore potential improvements.

Pólya Basis Let $t_1, \ldots, t_{2n} \in \mathbb{F}$ be arbitrary scalars, the Pólya basis functions $d_i(x)$ are polynomials of degree n defined as

$$d_i(x) = \prod_{j=1}^{n} (t_{i+j} - x)\,. \tag{3.1.1}$$

The set $\mathcal{P} = \{d_0(x), \ldots, d_n(x)\}$ is the *Pólya basis* of $\mathbb{F}[x]_{[n+1]}$, so any $P \in \mathbb{F}[x]_{[n+1]}$ is represented as

$$P(x) = \sum_{i=0}^{n} p_i d_i(x), \text{ where } p_i \in \mathbb{F}\,. \tag{3.1.2}$$

The p_i values are its coordinates. More details about the properties of Pólya polynomials can be found in [9, Ch 2, pp. 28] and [8,10]. For our purposes, the most relevant property is that both the monomial and the Lagrange basis are special cases of the Pólya basis.

[5] For applications in which $\mathbb{K} = \mathbb{R}$, programs use floating-point arithmetic. Multiplications and inverses are calculated with the FMUL and FDIV instructions, where the ratio $\mathbf{I}/\mathbf{M} = \frac{16}{4} = 4$ clock cycles according to Fog's report [21].

Change of Basis: Lagrange ↔ Pólya To instantiate the Pólya basis as the Lagrange basis, assign $(t_1, \dots, t_n, t_{n+1}, \dots, t_{2n}) = (x_1, \dots, x_n, x_0, \dots, x_{n-1})$, so the elements in the basis are related as

$$d_i(x) = \alpha_i \ell_i(x), \text{ where } \alpha_i = \prod_{\substack{j=0 \\ j \neq i}}^{n} (x_j - x_i) \text{ for } 0 \leq i \leq n\,, \tag{3.1.3}$$

and the coordinates are related as $(p_0, \dots, p_n) = (y_0/\alpha_0, \dots, y_n/\alpha_n)$.

Polynomial Evaluation As Warren's algorithm [41] generalizes Horner's algorithm means no inverses are required. Given $(t_1, \dots, t_{2n})$, Warren's algorithm evaluates a polynomial $P \in \mathbb{F}[x]_{[n+1]}$ with coordinates $(p_0, \dots, p_n)$ in Pólya basis. It requires $3n\mathbf{M} + 3n\mathbf{A}$, as shown in Algorithm 5.

Algorithm 5 Warren's Algorithm for Polynomial Evaluation in $\mathcal{P}$ Basis

Input: $[P]_{\mathcal{P}} = (p_0, \dots, p_n)$, and $x \in \mathbb{F}$.
Constant: $t_1, \dots, t_{2n}$.
Output: $P(x)$.
1: $l = 1$
2: $u = p_0$
3: **for** $i = 0$ **to** $n - 1$ **do**
4: $\quad l = l(t_{i+n+1} - x)$
5: $\quad u = u(t_{i+1} - x) + l p_{i+1}$
6: **end for**
7: **return** u

Proposed Algorithm We describe the last step that takes us to the promised $O(n)$ time-complexity algorithm for polynomial evaluation: namely, to instantiate Warren's algorithm leveraging the presence of the Nth-roots of unity, plus some additional tricks. We denote this choice as $\mathcal{P}\mu$ to distinguish it from the general case $\mathcal{P}$.

Here we describe some simplifications we observed, which could potentially be helpful in other contexts. First, Warren's algorithm uses $2n$ auxiliary scalars (t_i) that correspond directly to the nodes, so no additional storage is required. Moreover, at the ith iteration the algorithm calculates $(t_{i+n+1} - x)$ and $(t_{i+1} - x)$, and after instantiating with the roots of unity the values are $({\omega_N}^i - x)$ and $({\omega_N}^{i+1} - x)$, respectively. So in the next iteration we can reuse the second value effectively removing one subtraction per iteration.

Let us describe the change of basis. Each value y_i is converted to a Pólya coordinate as $p_i = y_i/\alpha_i$ for $0 \leq i < N$. A direct implementation of this conversion takes $N(\mathbf{M} + \mathbf{I})$, or $(4N - 3)\mathbf{M} + \mathbf{I}$ by applying the Montgomery's trick [31]. However, one (undesired) inverse remains.

We present an inversion-free method for the change of basis. Observe that α_i is related to w_i (defined in Section 2.3) as follows

$$\begin{aligned} \alpha_i &= \prod_{\substack{j=0 \\ j \neq i}}^{N-1} ({\omega_N}^j - {\omega_N}^i) = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} (-1)({\omega_N}^i - {\omega_N}^j) = (-1)^{N-1} \prod_{\substack{j=0 \\ j \neq i}}^{N-1} ({\omega_N}^i - {\omega_N}^j) \\ &= -1/w_i = -N/{\omega_N}^i, \quad \text{for } 0 \leq i < N\,. \end{aligned} \tag{3.1.4}$$

This equivalence replaces inversions of α_i with multiplications by ${\omega_N}^i$. Since the scalar $1/N$ appears in every coordinate, we can evaluate the scaled polynomial, and perform a scalar multiplication after evaluating the polynomial.

For implementation purposes, N can be either a fixed constant (hard-coded in the program), or a dynamic value. In the latter case, $1/N$ can be calculated faster with $O(\log N)$ operations and

no inverses. First, store the fixed constant $H = 2^{-1} \in \mathbb{F}$, for $N > 1$. Then, set $h = H$ and repeat $\log_2(N) - 1$ squares on $h = h^2$. Finally $1/N = h$ because N is always a power of two.

Finally, we present the proposed inversion-free $O(N)$ time-complexity algorithm for evaluating polynomials in the Lagrange basis with the Nth-roots of unity as nodes. We present two cases depending whether one or multiple polynomials are evaluated.

Single Evaluation. Given a polynomial $P \in \mathbb{F}[x]_{[N]}$ with values $(y_0, \ldots, y_{N-1})$ in the Lagrange basis $\mathcal{L}\mu$, Algorithm 6 evaluates $P(x)$ taking $(4N-3)\mathbf{M} + 2N\mathbf{A}$, plus $(\log_2(N)-1)\mathbf{S}$ when $1/N$ is not precomputed.

Batched Evaluation. Evaluating multiple polynomials at the same value of x is faster when processed as a batch. Given k polynomials $P_j \in \mathbb{F}[x]_{[N]}$ with values $\big((y_0)_j, \ldots, (y_{N-1})_j\big)$ in Lagrange basis $\mathcal{L}\mu$, for $0 < j \leq k$, Algorithm 7 evaluates $P_j(x)$ taking $\big((2N-1)(k+1)-1\big)\mathbf{M} + N(k+1)\mathbf{A}$. This eliminates the need to compute $2(N-1)(k-1)\mathbf{M} + N(k-1)\mathbf{A}$ when invoking Algorithm 6 k times. As expected, $k = 1$ covers the single evaluation case.

Algorithm 6 Our Algorithm for Polynomial Evaluation in $\mathcal{L}\mu$ Basis

Input: $[P]_{\mathcal{L}\mu} = (y_0, \ldots, y_{N-1})$ and $x \in \mathbb{F}$.
Constant: ${\omega_N}^0, \ldots, {\omega_N}^{N-1}$.
Output: $P(x)$.

$N' = 1/N$ {Using $(\log_2(N)-1)\mathbf{S}$.}
$l = 1$
$u = y_0$
$d = {\omega_N}^0 - x$
for $i = 1$ **to** $N-1$ **do**
  $l = ld$
  $d = {\omega_N}^i - x$
  $u = ud + l{\omega_N}^i y_i$
end for
return $-uN'$

Algorithm 7 Batched Algorithm for Polynomial Evaluation in $\mathcal{L}\mu$ Basis

Input: $[P_j]_{\mathcal{L}\mu} = \big((y_0)_j, \ldots, (y_{N-1})_j\big)$, for $0 < j \leq k$, and $x \in \mathbb{F}$.
Constant: ${\omega_N}^0, \ldots, {\omega_N}^{N-1}$.
Output: $P_j(x)$ for $0 < j \leq k$.

$N' = 1/N$ {Using $(\log_2(N)-1)\mathbf{S}$.}
$l = 1$
$u_j = (y_0)_j$ for $0 < j \leq k$.
$d = {\omega_N}^0 - x$
for $i = 1$ **to** $N-1$ **do**
  $l = ld$
  $d = {\omega_N}^i - x$
  $t = l{\omega_N}^i$
  for $j = 1$ **to** k **do**
    $u_j = u_j d + t(y_i)_j$
  end for
end for
return $-u_j N'$ for $0 < j \leq k$.

3.2 Compactly Storing Polynomials

Motivation. Suppose we want to compactly store a polynomial of degree k represented in an N-dimensional basis such that $0 < k < N$. Storing the N coordinates is suboptimal since it is known that $k+1$ coordinates are enough to determine a polynomial of degree k uniquely. In the monomial basis, we store only the first $k+1$ coefficients because the others are zero. To recover the polynomial from storage, we take the $k+1$ coefficients and pad them with zeros until obtaining N coefficients.

Problem. In the Lagrange basis, it is likely that the N values of the polynomial be non-zero. So, is it safe to store the first $k+1$ values and discard the others (even if they are non-zero)? If this is possible, how to recover the N original values (without switching to the monomial basis, nor re-evaluating the polynomial)?

Solution. As mentioned in Section 2.2, extending a polynomial in a Lagrange basis of dimension n to dimension $n+1$ takes $O(n)$ operations in the general case. To solve the problem described above, we give explicit formulas for basis extension for the special case of the Lagrange basis $\mathcal{L}\mu$.

Let $(y_0, \dots, y_{N-1})$ be the values of a polynomial of degree k, where $0 < k < N-1$, given the first $k+1$ values, $(y_0, \dots, y_k)$, the term y_{k+1} is calculated as

$$y_{k+1} = -\frac{1}{W_{(k+1,k+1)}} \sum_{i=0}^{k} y_i W_{(k+1,i)}, \quad \text{where } W_{(m,i)} = 1/\prod_{\substack{j=0 \\ j\neq i}}^{m} (\omega_N{}^i - \omega_N{}^j)\,. \tag{3.2.1}$$

This process takes $O(N)$ field operations, and can be iterated until the N original values are recovered taking $O(N^2)$ field operations in total.

Special Case We describe the case when $k = N-2$, i.e., the degree of the polynomial is one below the maximum degree allowed in an N-dimensional basis. In the monomial basis, padding one zero coefficient is enough. In the Lagrange basis $\mathcal{L}\mu$, the value y_{N-1} is calculated with $(N-1)\mathbf{M}+(N-2)\mathbf{A}$ operations as

$$\begin{aligned} y_{N-1} &= -\frac{1}{W_{(N-1,N-1)}} \sum_{i=0}^{N-2} y_i W_{(N-1,i)} = -\frac{1}{w_{N-1}} \sum_{i=0}^{N-2} y_i w_i \\ &= \sum_{i=0}^{N-2} -\frac{w_i}{w_{N-1}} y_i = \sum_{i=0}^{N-2} -\frac{\omega_N{}^i}{\omega_N{}^{N-1}} y_i \\ &= \sum_{i=0}^{N-2} \omega_N{}^j y_i\,, \text{ where } j = (i+1+N/2) \bmod N\,. \end{aligned} \tag{3.2.2}$$

The formulas presented in this section are helpful when extending polynomials just by a few number of dimensions. Increasing the dimension by one requires a linear number of operations, whereas extending to an arbitrarily larger dimension takes $O(N^2)$ operations. Despite this, particular instances of this basis extension are amenable to optimizations, as discussed in the next section.

3.3 Polynomial Multiplication in the Lagrange Basis $\mathcal{L}\mu$

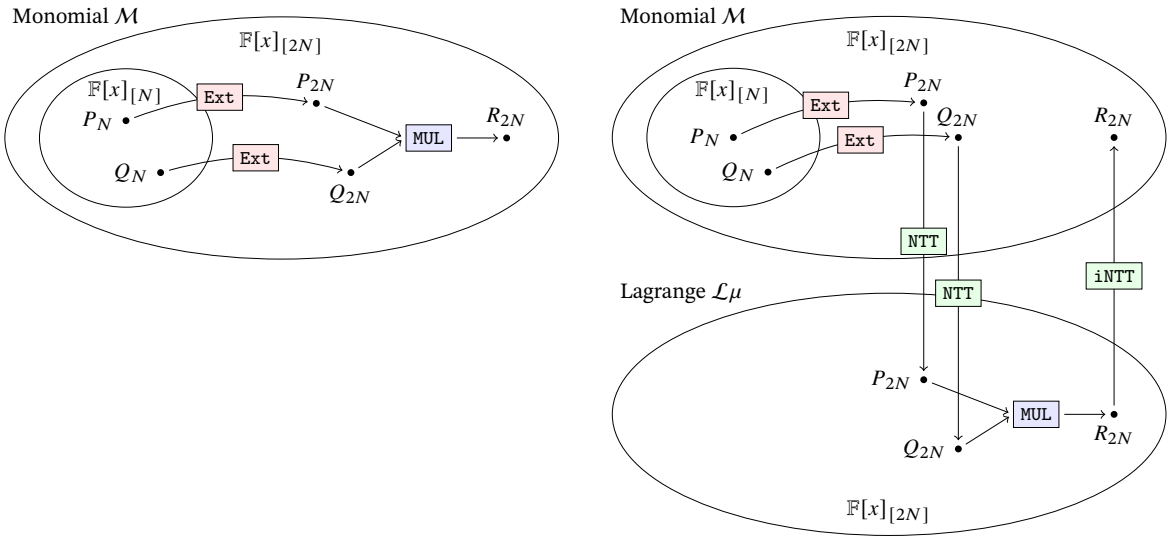
Motivation. Many high-level applications involve polynomial arithmetic, but are often implemented using the monomial basis. For efficiency, switching to alternative polynomial basis results in faster computations. For instance, polynomial multiplication takes, in general, $O(N^2)$ operations (see Figure 3a), but only $O(N \log N)$ operations over a field where the Nth-roots of unity exist.

NTT is key for achieving such a complexity. Let $\mathbf{NTT}_k$ denote the cost of calculating an NTT over a vector of size $k > 0$, and $\mathbf{iNTT}_k = \mathbf{NTT}_k + k\mathbf{M}$ (recall calculating $1/N$ is cheap). Also, $\mathbf{NTT}_{2k} = 2\,\mathbf{NTT}_k + k\mathbf{M}$ due to the two-way recursive structure of the algorithm.

We describe the classic $O(N \log N)$ algorithm for polynomial multiplication in monomial basis, as shown in Figure 3b. Starting with $P, Q \in \mathbb{F}[x]_{[N]}$ in basis $\mathcal{M}$, we want to calculate $R = P \times Q \in \mathbb{F}[x]_{[2N-1]}$ also in basis $\mathcal{M}$. First, P and Q must be extended to a basis of dimension $2N$. This step is trivial in $\mathcal{M}$, a reason why the basis extension operation does not receive too much attention. After that, P and Q are changed to basis $\mathcal{L}\mu$ by means of two $\mathbf{NTT}_{2N}$. Then, the values are multiplied taking $2N\mathbf{M}$, and the result is converted back to basis $\mathcal{M}$ taking one $\mathbf{iNTT}_{2N}$. In total, this method takes

$$\begin{aligned} \mathrm{Cost}_{\mathrm{PolyMul}}(\mathcal{M}) &= 2\,\mathbf{NTT}_{2N} + \mathbf{iNTT}_{2N} + 2N\mathbf{M} \\ &= 6\,\mathbf{NTT}_N + 7N\mathbf{M}\,. \end{aligned} \tag{3.3.1}$$

Because this method is asymptotically faster than the conventional algorithm, it has seen widely adopted in practice.



(a) $O(N^2)$ conventional algorithm.

(b) $O(N \log N)$ NTT-based algorithm.

Fig. 3. Algorithms for Polynomial Multiplication in the Monomial Basis $\mathcal{M}$.

Problem. Now we turn to the case of the Lagrange basis $\mathcal{L}\mu$. Starting with $P, Q \in \mathbb{F}[x]_{[N]}$ in basis $\mathcal{L}\mu$, we want to calculate $R = P \times Q \in \mathbb{F}[x]_{[2N-1]}$ also in basis $\mathcal{L}\mu$. The first step is to extend the basis of the input polynomials from a basis of dimension N to a basis of dimension $2N$. Unlike in the monomial basis, this basis extension is not trivial since it takes $O(N^2)$ operations in the general case. Once the polynomials are extended, it takes $2N\mathbf{M}$ to multiply their values for obtaining R. The challenge now is to determine efficient methods for this specific basis extension.

Faster Basis Extension from $\mathcal{L}\mu_N$ to $\mathcal{L}\mu_{2N}$ We describe two methods for polynomial multiplication based on two efficient algorithms for basis extension in Lagrange basis $\mathcal{L}\mu_N$.

Strategy 1 (Stolon). As shown in Figure 4a, convert each polynomial to basis $\mathcal{M}$ of dimension N (taking $2\,\mathbf{iNTT}_N$), and pad with zeros the resultant polynomials to get $2N$ coefficients (basis extension in basis $\mathcal{M}$). Then, convert the extended polynomials back to the Lagrange basis $\mathcal{L}\mu$ (taking $2\,\mathbf{NTT}_{2N}$), and multiply them in $\mathcal{L}\mu_{2N}$ basis (taking $2N\mathbf{M}$). In total, Strategy 1 requires

$$\begin{aligned}\mathrm{Cost}_{\mathrm{PolyMul}}(\mathcal{L}\mu) &= 2\,\mathbf{iNTT}_N + 2\,\mathbf{NTT}_{2N} + 2N\mathbf{M} \\ &= 6\,\mathbf{NTT}_N + 6N\mathbf{M}\,.\end{aligned} \tag{3.3.2}$$

Strategy 2 (Rhizome). We introduce a more efficient polynomial multiplication algorithm based on a faster basis extension. Note that Strategy 1 uses NTTs of size $2N$. In contrast, our method uses only NTTs of size N to generate N new values, capitalizing on the existing N input values.

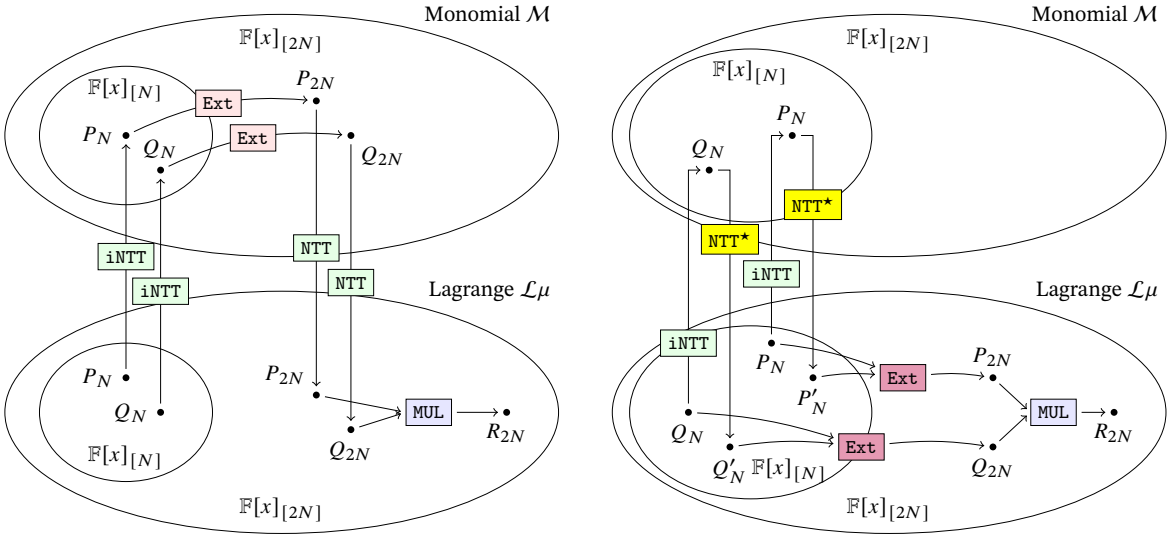
We define the NTT* function, a slightly modified version of the NTT, as

$$\mathtt{NTT}^\star\colon \big(\omega_N, s, (c_0, \dots, c_{N-1})\big) \mapsto \big(P(s{\omega_N}^0), \dots, P(s{\omega_N}^{N-1})\big), \text{ where } s \in \mathbb{F}\,. \tag{3.3.3}$$

The implementation of this function is identically to the NTT in Algorithm 4, except that the Step 3 (marked with ★) is replaced with $w = s$. NTT* runs as fast as NTT. It can be seen that $\mathtt{NTT}(\omega_N, \cdot) = \mathtt{NTT}^\star(\omega_N, 1, \cdot)$.

Given the values $[P]_{\mathcal{L}\mu_N} = (y_0, \dots, y_{N-1})$ on dimension N, we calculate the values $[P]_{\mathcal{L}\mu_{2N}} = (y'_0, \dots, y'_{2N-1})$ on dimension $2N$ as

$$\begin{aligned}(v_0, \dots, v_{N-1}) &= \mathtt{NTT}^\star(\omega_N, \omega_{2N}, \mathtt{iNTT}(\omega_N, (y_0, \dots, y_{N-1}))) \\ [P]_{\mathcal{L}\mu_{2N}} &= (y'_0, \dots, y'_{2N-1}) = (y_0, v_0, \dots, y_{N-1}, v_{N-1})\end{aligned} \tag{3.3.4}$$



(a) Strategy 1 – Stolon. (b) Strategy 2 – Rhizome.

Fig. 4. Strategies for polynomial multiplication in the Lagrange basis $\mathcal{L}\mu$.

This basis extension takes $\mathbf{NTT}_N + \mathbf{iNTT}_N$, i.e., it evaluates NTTs of size N.

Polynomial multiplication using Strategy 2, shown in Figure 4b, is as follows. Extend each input polynomial with the basis extension described above, and multiply the extended polynomials taking $2N\mathbf{M}$. In total, Strategy 2 requires

$$\begin{aligned} \text{Cost}_{\text{PolyMul}}(\mathcal{L}\mu) &= 2\,(\mathbf{NTT}_N + \mathbf{iNTT}_N) + 2N\mathbf{M} \\ &= 4\,\mathbf{NTT}_N + 4N\mathbf{M}\,. \end{aligned} \tag{3.3.5}$$

In summary, Strategy 2 avoids the calculation of two NTTs of size N in comparison to Strategy 1. This results in polynomial multiplications in Lagrange basis $\mathcal{L}\mu$ being one third faster.

Although the polynomial multiplication algorithms described above share the same $O(N \log N)$ time-complexity, we have demonstrated that their actual efficiency is significantly affected by the choice of basis and the specific inner operations used.

4 Improving Prio

We show key points of the Prio protocol that benefit from using polynomials in the Lagrange basis $\mathcal{L}\mu$. The Prio's core component is the proof used to validate measurements. We describe the proof system used in the (work-in-progress) specification document [6] since this version of Prio is currently under consideration for standardization.

4.1 Fully Linear Proofs

Boneh et al. [11] introduced the notion of fully linear probabilistically-checkable proofs (FLPCP) that allows the verifier to make linear queries to the input statement. This is useful in settings where the input statement (and the proof) is not available to the verifier. Davis et al. [17] showed a provable security treatment of verifiable distributed aggregation functions (VDAF). They showed that with minor changes to a version of Prio (the one described in the draft-v03 [7]), Prio is private and robust if the fully linear proof is sound and honest-verifier zero-knowledge. This FLPCP construction corresponds to the 1.5-round, public coin, fully linear, interactive oracle proof system by Boneh et al. [11].

A FLPCP is proof system for the relation $R_C = \{(x, w) \mid C(x, w) = 0\}$, where x is the input, w is the witness, and C is an arithmetic circuit over $\mathbb{F}$ composed of M instances of a G-gate and any number of affine gates. A G-gate is an L-arity circuit composed of multiplication gates and affine gates.

`FLPCP.Prove` The Prover evaluates C on input (x, w). While doing so, it defines polynomials $f_1, \ldots, f_L \in \mathbb{F}[x]$ as follows: $f_i(0)$ is chosen at random, and $f_i(j)$ is the value on the ith input wire to the jth G-gate, for all $1 \leq i \leq L$, and $1 \leq j \leq M$. Then, it constructs a polynomial $p = G(f_1, \ldots, f_L) \in \mathbb{F}[x]$ such that $p(j)$ is the value on the output wire of the jth G-gate, for all $1 \leq j \leq M$. By construction, $p(M) = C(x, w)$. Finally, the Prover outputs the proof $\pi = (w, f_1(0), \ldots, f_L(0), p)$.

`FLPCP.Query` The Verifier receives $\pi' = (w', z'_1, \ldots, z'_L, p')$, such that $z'_i = f'_i(0)$, and uses these values to recover the polynomials $f'_1, \ldots, f'_L \in \mathbb{F}[x]$. The goal is to check whether $p' = G(f'_1, \ldots, f'_L)$ and $p'(M) = 0$. To do so, the Verifier samples $r \leftarrow_\$ \mathbb{F} \setminus \{1, \ldots, M\}$ at random and outputs the verifier message $V = \{p'(r), f'_1(r), \ldots, f'_L(r), p'(M)\}$.

`FLPCP.Decision` The decision algorithm takes $V = (a, a_1, \ldots, a_L, b)$, such that $a = p'(r)$, $a_1 = f'_1(r), \ldots, a_L = f'_L(r)$, and $b = p'(M)$. The Verifier accepts if $a = G(a_1, \ldots, a_L)$ and $b = 0$.

The FLPCP construction is useful scenarios where exist k verifiers. The Prover obtains a proof π by running `FLPCP.Prove`, produces secret shares $\pi = \sum_{i=1}^{k} [\pi]_i$, and sends $[\pi]_i$ to the ith verifier. Each verifier runs `FLPCP.Query` on $[\pi]_i$ to get secret shares of the verifier message $[V]_i$. After that, each verifier publishes $[V]_i$ to recover $V = \sum_{i=1}^{k} [V]_i$ and invoke the `FLPCP.Decision` algorithm.

4.2 Efficient Proof Generation

To benefit from the asymptotically faster algorithms derived from the Lagrange basis $\mathcal{L}\mu$, the first observation is to evaluate the f_i polynomials on the powers of a root of unity. That is, $f_i(\omega^j)$ for all $1 \leq j \leq M$, where ω is a root of unity in $\mathbb{F}$.

In `FLPCP.Prove`, the polynomials $f_1, \ldots, f_L$ are represented in the Lagrange basis $\mathcal{L}\mu$ by construction and must remain in this representation. We note that libprio-rs converts every f_i to the monomial basis invoking an iNTT of size $2N$ per polynomial. All these iNTT can be eliminated if the polynomials f_i remain in the Lagrange basis $\mathcal{L}\mu$.

To construct the polynomial p, the Prover requires to calculate a product of polynomials $f_1 \times f_2$ for each multiplication gate. We observe that it is faster to perform polynomial multiplications entirely in the Lagrange basis $\mathcal{L}\mu$ employing Strategy 2, presented in Section 3.3. As p is a product of polynomials f_1 and f_2 of degree $N - 1$, the degree of p is $2N - 2$ and the coefficient of the highest-degree term is always zero.

Once p was generated, the Prover encodes p as part of the proof. Storing a polynomial takes the same amount of space in any basis as long as the dimension remains invariant. We note that the polynomial can be compactly stored in monomial basis: the Prover encodes only the first $2N - 1$ coefficients (because the last one is zero), and the Verifier pads with one zero to get p in monomial basis of dimension N. We have shown in Section 3.2 that this trick is also applicable when p is in the Lagrange basis $\mathcal{L}\mu$ representation.

4.3 Verification in Linear-time

Once the changes in `FLPCP.Prove` have been adopted, the Verifier receives $\pi' = (w', z'_1, \ldots, z'_L, p')$, where p' is a polynomial in the Lagrange basis $\mathcal{L}\mu$. If the Prover sent only $2N - 1$ values of p', the Verifier is able to recover the last value by performing basis extension taking $O(N)$ operations, as we have shown in the special case of Section 3.2.

The Verifier recovers the polynomials $f'_1, \ldots, f'_L \in \mathbb{F}[x]$ from $z'_1, \ldots, z'_L$. Along with p these polynomials are, by construction, in the Lagrange basis $\mathcal{L}\mu$. The Verifier samples $r \leftarrow_\$ \mathbb{F} \setminus \{\omega^0, \ldots, \omega^M\}$

at random and evaluates these polynomials at r. Note that no extra security assumptions are necessary on r.

In libprio-rs, the polynomials f_i are changed to the monomial basis and then evaluated. So for each polynomial, it performs one iNTT of size N and one invocation of Horner's algorithm, which takes $O(N \log N)$ operations. This procedure can be improved.

It is faster to evaluate the polynomials directly in the Lagrange basis $\mathcal{L}\mu$ using our proposed Algorithm 6, described in Section 3.1. Moreover, note that all the f_i polynomials must be evaluated at the same value r, this enables the use of a batched evaluation by employing Algorithm 7 for greater efficiency. Since both algorithms have $O(N)$ time-complexity, this demonstrates that proof verification runs in linear time and without performing inverses.

5 Performance Benchmarks

We show our algorithms in practice. Looking for existing implementations, the authors of Prio [15] made available a prototype written in Go. Mozilla's implementation [32] is a C language library used for experimentation purposes only, but is not maintained anymore. On the other hand, the libprio-rs [18] is a Rust library actively maintained and is compliant with the VDAF specification document [6]. We forked libprio-rs in https://github.com/armfazh/rhizomes/ applying the recommendations given in Section 4. We plan to contribute the changes to the upstream library.

For performance measurements, the library includes a test suite and a set of benchmarks for different aggregation functions. It also supports field arithmetic for 64-bit and 128-bit NTT-friendly prime fields. We use the bigger field $\mathbb{F} = \mathrm{GF}(p)$, where $p = 2^{66} \cdot 4611686018427387897 + 1$ is a 128-bit prime. We use the Rust compiler v1.71 and run benchmarks on a Core i7-13800H processor. The measurements reported are obtained using the Criterion.rs library [5], which is a tool for performing statistical measurements of code's execution time.

Several benchmarks for the high-level phases of the Prio protocol are supported in the libprio-rs library. We focus on the most time-consuming phases of the Prio protocol that benefit from our recommendations:

- Sharding. In this phase, a client takes its measurement and creates a proof to be sent to to the Verifier. Internally, this phase invokes to `FLPCP.Prove` for generating the proof.
- Prepare-Init. After a server received a secret share of the proof from the client, the server validates the measurement for aggregation. Internally, here is when the `FLPCP.Query` algorithm is executed.

We report execution times of two aggregation functions used in practice, where the time required scales with the input size n (which internally determines the degree of polynomials):

- SumVec sums n-length vectors of integers in the range 0 to 2^b, for some b.
- Histogram estimates the distribution of some measurements. Let n be the number of buckets, each measurement increments the counter of one bucket. The aggregate result is the counter of each bucket.

Table 1 and Table 2 show, respectively, the execution time of the Sharding and Prepare-Init phases as the input size grows. These measurements are plotted in logarithmic scale and are shown in Figure 5 and Figure 6.

For the Sharding phase, larger speedup factors are obtained as the input size n grows. From Figure 5, we can see that the inflection point appears as n approaches to $n = 1000$ in both aggregation functions. The optimizations described in Section 4.2 resulted in a 19% to 36% reduction in execution time for both the SumVec and Histogram aggregation functions.

For the Prepare-Init phase, we replaced the interpolate-and-evaluate method in `FLPCP.Query` with a direct polynomial evaluation in the Lagrange basis, as described in Section 4.3. As Table 2

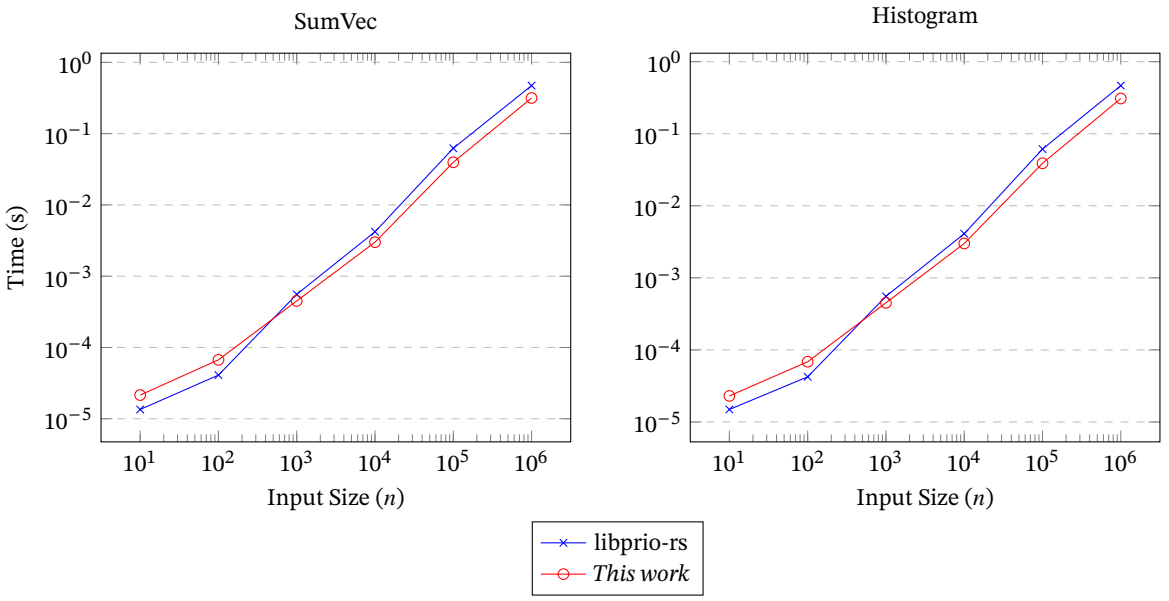


Fig. 5. Execution time (plotted in logarithmic scale) of Prio's Sharding phase.

Table 1. Execution time of Prio's Sharding phase.

Aggregation Function	Code	Input Size (n)					
		10^1	10^2	10^3	10^4	10^5	10^6
SumVec	libprio-rs	13.5 µs	41.0 µs	556.7 µs	4.2 ms	62.3 ms	471.7 ms
	This work	21.5 µs	67.1 µs	449.5 µs	3.0 ms	39.6 ms	316.0 ms
	Speedup	0.62x	0.61x	1.24x	1.37x	1.58x	1.49x
Histogram	libprio-rs	14.9 µs	42.3 µs	554.5 µs	4.1 ms	61.2 ms	465.8 ms
	This work	23.0 µs	68.5 µs	449.9 µs	3.0 ms	38.9 ms	308.1 ms
	Speedup	0.64x	0.61x	1.23x	1.39x	1.57x	1.51x

shows, the latency of the Prepare-Init phase achieves significant speedup factors. For $n \geq 1000$, its execution time is now twice as fast. Moreover, the fact that no inverses are calculated brings savings even for small values of n.

These results confirm that Prio clearly benefits from the use of efficient representations for polynomial arithmetic. The speedups achieved are result of removing redundant operations. We show that switching to the monomial basis is unnecessary, it is better to keep polynomials in the Lagrange basis because this enables faster operations. These modifications to Prio positively influence the design of the protocol currently proposed for standardization.

6 Conclusions

Polynomials are useful in many research areas. In cryptographic protocols like Prio, polynomials help to detect invalid client inputs. When dealing with polynomial arithmetic operations, there is a need for methods that are not only asymptotically efficient, but also faster in practice.

Our focus centered on finding optimizations for polynomial arithmetic in the Lagrange basis. Relying on the Pólya basis, we show an $O(n)$ time-complexity algorithm for polynomial evaluation in the Lagrange basis. The proposed algorithm is an inversion-free alternative to the barycentric formula. In addition, we study the case of basis extension showing explicit formulas with applications on compact storage of polynomials and polynomial multiplication.

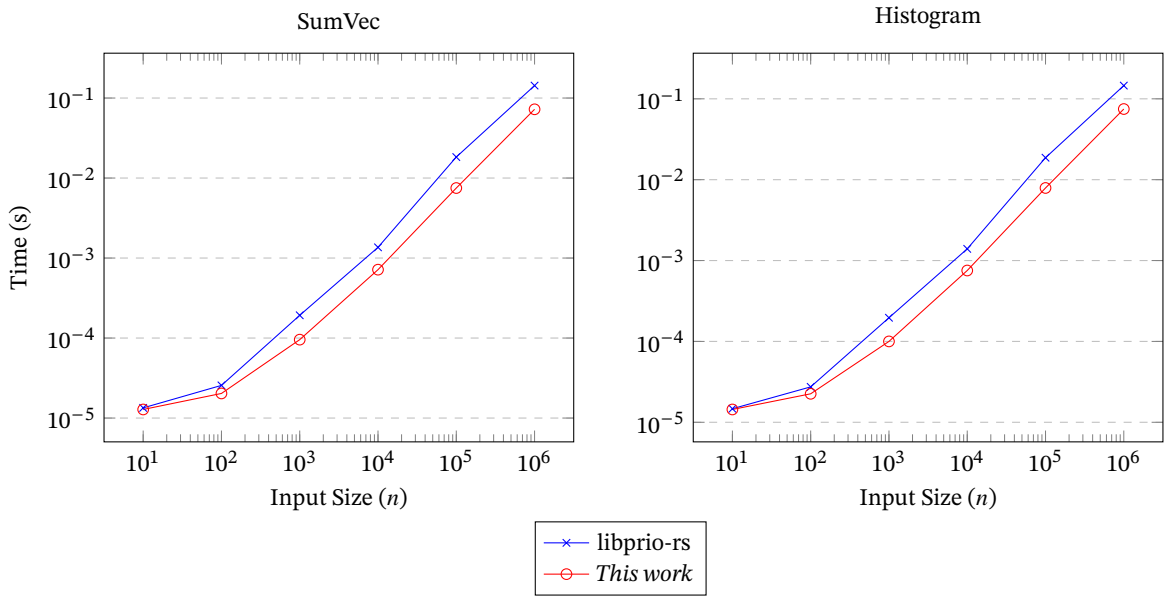


Fig. 6. Execution time (plotted in logarithmic scale) of Prio's Prepare-Init phase.

Table 2. Execution time of Prio's Prepare-Init phase.

Aggregation Function	Code	Input Size (n)					
		10^1	10^2	10^3	10^4	10^5	10^6
SumVec	libprio-rs	13.4 µs	25.6 µs	192.5 µs	1360.8 µs	18.3 ms	143.1 ms
	This work	12.8 µs	20.3 µs	95.5 µs	715.6 µs	7.5 ms	72.3 ms
	Speedup	1.04x	1.26x	2.02x	1.90x	2.44x	1.98x
Histogram	libprio-rs	14.7 µs	27.4 µs	196.2 µs	1395.2 µs	18.7 ms	145.8 ms
	This work	14.4 µs	22.5 µs	100.0 µs	753.4 µs	7.9 ms	74.8 ms
	Speedup	1.02x	1.22x	1.96x	1.85x	2.38x	1.95x

Prio achieved significant savings in its execution time. This is in part because we make extensive use of the Lagrange basis across the time-consuming operations. With our algorithm for polynomial evaluation, verification runs in linear time without extra security assumptions. Evaluation of our changes demonstrated that Sharding is 36% faster, and Prepare-Init runs in half the time.

A key takeaway for protocol designers is the critical importance of selecting efficient data representations as part of the protocol development. As we demonstrated in this optimization work, effective parametrization is fundamental for achieving high efficiency, which directly impacts on the performance and scalability of the deployed systems.

Acknowledgments. The author appreciates the anonymous reviewers for their feedback. My sincere thanks to Cefan Rubin for reading and providing thoughtful comments on an early draft, and to Christopher Patton for his encouragement to pursue work on fully linear proof systems.

Disclosure of Interests. The author has no conflicts of interests to declare that are relevant to the content of this article.

References

1. Aas, J., Geoghegan, T.: Introducing ISRG Prio Services for Privacy Respecting Metrics. Tech. rep., Internet Security Research Group (Nov 2020), https://www.abetterinternet.org/post/introducing-prio-services/
2. Aas, J., Gran, S.: ISRG Prio Services for Preserving Privacy in COVID 19 EN Apps. Tech. rep., Internet Security Research Group (Jun 2021), https://www.abetterinternet.org/post/prio-services-for-covid-en/
3. Addanki, S., Garbe, K., Jaffe, E., Ostrovsky, R., Polychroniadou, A.: Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares. In: Galdi, C., Jarecki, S. (eds.) Security and Cryptography for Networks. p. 516–539. Springer, Cham (Sep 2022). https://doi.org/10.1007/978-3-031-14791-3_23
4. Anderson, E., Chase, M., Durak, F.B., Laine, K., Weng, C.: Precio: Private Aggregate Measurement via Oblivious Shuffling. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. p. 1819–1833. CCS '24, Association for Computing Machinery, New York, NY, USA (Dec 2024). https://doi.org/10.1145/3658644.3670280
5. Aparicio, J., Heisler, B.: Criterion.rs: Statistics-driven Microbenchmarking in Rust (Jul 2025), https://github.com/bheisler/criterion.rs, version 0.7.0
6. Barnes, R., Cook, D., Patton, C., Schoppmann, P.: Verifiable Distributed Aggregation Functions. Internet-Draft draft-irtf-cfrg-vdaf-15, Internet Engineering Task Force (Jun 2025), https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf/15, Work in Progress
7. Barnes, R., Patton, C., Schoppmann, P.: Verifiable Distributed Aggregation Functions. Internet-Draft draft-irtf-cfrg-vdaf-03, Internet Engineering Task Force (Aug 2022), https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf/03/, Work in Progress
8. Barry, P.J., Goldman, R.N.: Interpolation and approximation of curves and surfaces using Pólya polynomials. CVGIP: Graphical Models and Image Processing **53**(2), 137–148 (Mar 1991). https://doi.org/10.1016/1049-9652(91)90057-Q
9. Barry, P.J., Goldman, R.N.: Chapter 2: Algorithms for Progressive Curves: Extending B-Spline and Blossoming Techniques to the Monomial, Power, and Newton Dual Bases. In: Goldman, R.N., Lyche, T. (eds.) Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces, chap. 2, p. 11–63. Other Titles in Applied Mathematics (1992). https://doi.org/10.1137/1.9781611971583.ch2
10. Barry, P.J., Goldman, R.N., DeRose, T.D.: B-splines, Pólya curves, and duality. Journal of Approximation Theory **65**(1), 3–21 (Apr 1991). https://doi.org/10.1016/0021-9045(91)90109-N
11. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. Lecture Notes in Computer Science, vol. 11694, p. 67–97. Springer, Cham (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8_3
12. Burden, R.L., Faires, .D.: Numerical Analysis. Brooks/Cole, Cengage Learning, 9th edn. (2010), https://faculty.cengage.com/works/9781305253667
13. Cook, S.A., Aanderaa, S.O.: On the Minimum Computation Time of Functions. Transactions of the American Mathematical Society **142**, 291–314 (Aug 1969). https://doi.org/10.2307/1995359
14. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19**(90), 297–301 (Apr 1965). https://doi.org/10.2307/2003354
15. Corrigan-Gibbs, H., Boneh, D.: Prototype implementation of Prio, a system for the private computation of aggregate statistics., https://github.com/henrycg/prio
16. Corrigan-Gibbs, H., Boneh, D.: Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). p. 259–282. USENIX Association, Boston, MA (Mar 2017), https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs
17. Davis, H., Patton, C., Rosulek, M., Schoppmann, P.: Verifiable Distributed Aggregation Functions. Proceedings on Privacy Enhancing Technologies (PoPETs) **2023**(4), 578–592 (Jun 2023). https://doi.org/10.56553/popets-2023-0126
18. Divvi Up: libprio-rs: Implementation of Prio in Rust (Jan 2025), https://github.com/divviup/libprio-rs, version 0.17.0-alpha.0
19. Englehardt, S.: Next steps in privacy-preserving Telemetry with Prio. Tech. rep., Mozilla Security Blog (Jun 2019), https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/
20. Farin, G.: Chapter 2 - Introductory Material. In: Farin, G. (ed.) Curves and Surfaces for Computer-Aided Geometric Design, p. 13–28. Academic Press, Boston, third edn. (1993). https://doi.org/10.1016/B978-0-12-249052-1.50007-6
21. Fog, A.: Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs (Jul 2025), https://www.agner.org/optimize/instruction_tables.pdf
22. Gander, W.: Change of basis in polynomial interpolation. Numerical Linear Algebra with Applications **12**(8), 769–778 (Oct 2005). https://doi.org/10.1002/nla.450
23. Gentleman, W.M., Sande, G.: Fast Fourier Transforms: for fun and profit. In: Fall Joint Computer Conference. AFIPS '66 (Fall), vol. 29, p. 563–578. Association for Computing Machinery, New York, NY, USA (Nov 1966). https://doi.org/10.1145/1464291.1464352
24. Geoghegan, T., Patton, C., Rescorla, E., Wood, C.A.: Standardizing MPC for Privacy Preserving Measurement. In: Real World Crypto Symposium 2022 (Apr 2022), https://rwc.iacr.org/2022/
25. Geoghegan, T., Raykova, M., Jacobs, F.: Exposure Notifications Private Analytics. In: Real World Crypto Symposium 2022 (Apr 2022), https://rwc.iacr.org/2022/

26. Goldman, R.: Lecture 24: Blending Functions: An Alternative Approach to Freeform Curves and Surfaces. Tech. rep., Rice University (Nov 2004), https://www.clear.rice.edu/comp360/lectures/old/BasisFText.pdf
27. Helmer, R., Miyaguchi, A., Rescorla, E.: Testing Privacy-Preserving Telemetry with Prio. Tech. rep., Mozilla Hacks (Oct 2018), https://hacks.mozilla.org/2018/10/testing-privacy-preserving-telemetry-with-prio/
28. Horner, W.G.: A new method of solving numerical equations of all orders, by continuous approximation. Philosophical Transactions of the Royal Society of London **109**, 308–335 (Dec 1819). https://doi.org/10.1098/rstl.1819.0023
29. IETF Security Area: Privacy Preserving Measurement (Working Group) (Jan 2022), https://datatracker.ietf.org/wg/ppm/
30. Karatsuba, A., Ofman, Y.: Multiplication of Multidigit Numbers on Automata. Soviet Physics. Doklady **7**(7), 595–596 (Jan 1963)
31. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177), 243–264 (Jan 1987). https://doi.org/10.1090/s0025-5718-1987-0866113-7
32. Mozilla: libprio - A Prio library in C using NSS (May 2020), https://github.com/mozilla/libprio, version 1.6
33. OEIS Foundation Inc.: Proth primes, Entry A080076. In: The on-line encyclopedia of integer sequences (2025), https://oeis.org/A080076
34. Patton, C., Galicer, M.: Privacy-preserving measurement and machine learning. Tech. rep., The Cloudflare Blog (Sep 2023), https://blog.cloudflare.com/deep-dive-privacy-preserving-measurement
35. Pollard, J.M.: The fast Fourier transform in a finite field. Math. Comp. **25**(114), 365–374 (Apr 1971). https://doi.org/10.1090/S0025-5718-1971-0301966-0
36. Pólya, G., Szegö, G.: Problems and Theorems in Analysis II, Classics in Mathematics, vol. 2. Springer, Berlin, Heidelberg, first edn. (1976). https://doi.org/10.1007/978-3-642-61905-2
37. Ruscheweyh, S., Saff, E.B., Salinas, L.C., Varga, R.S. (eds.): Behavior of the Lagrange interpolants in the roots of unity, Lecture Notes in Mathematics, vol. 1435. Springer, Berlin, Heidelberg (Mar 1990). https://doi.org/10.1007/BFb0087899
38. Rutishauser, H.: Lectures on Numerical Mathematics. Birkhäuser, Boston, MA, first edn. (1990). https://doi.org/10.1007/978-1-4612-3468-5
39. Schwartz, J.T.: Fast Probabilistic Algorithms for Verification of Polynomial Identities. Journal of the ACM **27**(4), 701–717 (Oct 1980). https://doi.org/10.1145/322217.322225
40. Toom, A.L.: The complexity of a scheme of functional elements realizing the multiplication of integers. Doklady Akademii Nauk SSSR **150**, 496–498 (1963)
41. Warren, J.: An Efficient Algorithm for Evaluating Polynomials in the Pólya Basis. In: Hagen, H., Farin, G., Noltemeier, H. (eds.) Geometric Modelling. Computing Supplement, vol. 10, p. 357–361. Springer, Vienna (1995). https://doi.org/10.1007/978-3-7091-7584-2_24
42. Winograd, S.: On the Time Required to Perform Multiplication. Journal of the ACM **14**(4), 793–802 (Oct 1967). https://doi.org/10.1145/321420.321438
43. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, E.W. (ed.) Symbolic and Algebraic Computation. Lecture Notes in Computer Science , vol. 72, p. 216–226. Springer, Berlin, Heidelberg (Jun 1979). https://doi.org/10.1007/3-540-09519-5_73