Grotto: Screaming fast $(2+1)$-PC for $\mathbb{Z}_{2^n}$ via $(2,2)$-DPFs

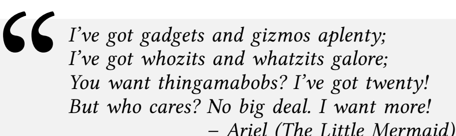
Kyle Storrier
University of Calgary
kyle.storrier@ucalgary.ca

Adithya Vadapalli
University of Waterloo
adithya.vadapalli@uwaterloo.ca

Allan Lyons
University of Calgary
allan.lyons@ucalgary.ca

Ryan Henry
University of Calgary
ryan.henry@ucalgary.ca

***Abstract*—We introduce Grotto, a framework and C++ library for space- and time-efficient $(2+1)$-party piecewise polynomial (i.e., spline) evaluation on secrets additively shared over $\mathbb{Z}_{2^n}$. Grotto improves on the state-of-the-art approaches based on distributed comparison functions (DCFs) in almost every metric, offering asymptotically superior communication and computation costs with the same or lower round complexity. At the heart of Grotto is a novel observation about the structure of the "tree" representation underlying the most efficient distributed point functions (DPFs) from the literature, alongside an efficient algorithm that leverages this structure to do with a single DPF what state-of-the-art approaches require many DCFs to do. Our open-source Grotto implementation supports evaluating dozens of useful functions out of the box, including trigonometric and hyperbolic functions (and their inverses); various logarithms; roots, reciprocals, and reciprocal roots; sign testing and bit counting; and over two dozen of the most common (univariate) activation functions from the deep-learning literature.**



1. Introduction

Secure multiparty computation (MPC) offers a wealth of opportunities to regain privacy in various facets our increasingly digital lives. For example, privacy-enhanced machine learning algorithms may one day soon allow us to contribute our data "for the common good" without ever revealing that data to anyone. A significant technical challenge on the path to this vision has been to efficiently mix different kinds of computation; i.e., evaluating non-linear functions when working in a linear scheme or performing heavy arithmetic in a digital circuit-based scheme.

The need to evaluate non-linear functions in a linear setting arises, for example, in the construction of private neural networks, where non-linear activation functions intersperse between linear layers. Early systems like SecureML [22] resorted to using "MPC-friendly" knock-offs of the activation functions used in non-MPC domains. This paper presents a novel method, and implementation thereof, for the fast and accurate approximation of such functions using so-called distributed point functions.

Roadmap

The remainder of the paper proceeds as follows. Section 2 lays out our notational conventions and recalls a few fundamental cryptographic primitives used in the subsequent sections. From here, Section 3 describes *selection vectors* and their applications to the oblivious evaluation of piecewise-polynomial functions (or splines); Section 4 presents a new data structure called the *parity-segment tree* along with our *prefix-parity algorithm* for rapidly extracting from a parity-segment tree the information we need in the sequel; and then Section 5 describes *point functions*, *distributed point functions*, and the specific construction thereof we employ. Section 6 serves as a confluence of the preceding three sections, detailing how to run the prefix-parity algorithm directly on a distributed point function for the oblivious evaluation of piecewise-polynomial functions. Section 7 introduces Grotto, our framework and open-source software implementation of the new techniques, and then Section 8 follows with a performance evaluation of Grotto and a head-to-head comparison with related work. Sections 9 and Section 10 wrap up with a discussion of related work and some concluding remarks. Additional material elaborating on and extending results from the main body is included as appendices.

2. Preliminaries & notation

We deal extensively with vectors over $\mathbb{Z}_N$. For the special case where $N = 2$, we equate such a vector with the corresponding bitstring (i.e., the bitstring composed of the same bits, in the same order). We use '$\oplus$' to denote the bitwise exclusive-OR (XOR) operator and '+' to denote normal addition in a ring (or a module over a ring) of characteristic other than 2.

We write $\gg$ and $\ll$ respectively for the arithmetic (sign-extended) right shift or logical left shift applied to fixed-width bitstrings; likewise, we write $\ggg$ and $\lll$ for *cyclic rotation* to the right or left, whether of a bitstring or of a vector.

The *substring* of x starting at index a (inclusive) and ending at b (exclusive) is denoted by $x[a\,..\,b)$. A substring $x[0\,..\,b)$ with starting index 0 is called a *prefix* of x.

2.1. Fixed-point arithmetic

Fixed-point representations encode (approximations to) real numbers using signed integers in two's-complement format. Specifically, the fixed-point approximation to $x \in \mathbb{R}$ is $\lfloor x \cdot 2^p \rceil$, where $p \in \mathbb{N}$ is a *fractional precision* parameter indicating how many bits to reserve for the fractional (non-integer) part of x. Assuming 64-bit representations, this leaves $64 - p - 1$ bits for the integer part of x (plus one bit for the sign).

For example, the fixed-point approximation to $\pi = 3.14159\ldots$ using $p = 16$ fractional bits is

$$\begin{aligned}\lfloor \pi \cdot 2^{16} \rceil &= \lfloor 205\,887.41614566\ldots \rceil \\ &= 205\,887 \\ &= \underbrace{\texttt{0x000000000003}}_{\text{sign bit + integer part}}\overbrace{\texttt{243f}}^{\text{fractional part}}.\end{aligned}$$

Addition (or subtraction) of fixed-point numbers (assuming a common p) is realized using addition (or subtraction) of the underlying integers. The resulting sum (or difference) is exact, provided no overflow occurs.

To multiply fixed-point numbers x_0 and x_1, respectively having p_0 and p_1 fractional bits, it suffices to multiply the underlying integer representations. The resulting product has $p = p_0 + p_1$ fractional bits and is likewise exact when no overflow occurs.

For example, we can compute the area of a circle with (unitless) radius $r = 1.25$ by expressing r as a fixed-point number and computing

$$\begin{aligned}\lfloor \pi \cdot 2^{16} \rceil \cdot \lfloor 1.25 \cdot 2^{16} \rceil^2 &= 205\,887 \cdot 81\,920^2 \\ &= 1\,381\,684\,268\,236\,800 \\ &= \underbrace{\texttt{0x0004}}_{\text{sign bit + integer part}}\overbrace{\texttt{e8a270000000}}^{\text{fractional part}},\end{aligned}$$

a fixed-point number with $p' = 16 + (16 + 16) = 48$ fractional bits and, consequently, just $64 - 48 - 1 = 15$ bits remaining for the integer part.

To "reset" the number of fractional bits back to $p = 16$, it suffices to perform an arithmetic (sign-extending) right shift by $p' - p = 32$ bits; that is,

$$\begin{aligned}\big(\underbrace{\texttt{0x0004}}_{\text{sign bit + integer part}}\overbrace{\texttt{e8a270000000}}^{\text{fractional part}} \gg 32\big) &= \texttt{0x}\overbrace{\texttt{00000000}}^{\text{redundant sign bits}}\underbrace{\texttt{0004}}_{\text{sign bit + integer part}}\overbrace{\texttt{e8a2}}^{\text{fractional part}} \quad (2) \\ &= 321\,698 \\ &= \lfloor 4.908721923828125 \cdot 2^{16} \rceil.\end{aligned}$$

Meanwhile, $\pi \cdot 1.25^2 = 4.9087385\ldots$ so that

$$\pi \cdot r^2 - 4.908721923828125 \approx 0.00000165974059 < 2^{-16}.$$

2.2. Secret sharing

Secret sharing allows a *dealer* to distribute a secret among two or more *shareholders* in such a way that individual shareholders learn nothing while "authorized subsets" of shareholders easily learn the whole secret [29]. Unless otherwise stated, all shares are assumed to be 64-bit $(2, 2)$-*threshold shares.*

$(2, 2)$-Additive sharing. In $(2, 2)$-*additive sharing*, there are just two shareholders, both of whom must cooperate to recover the secret.

To share a 64-bit integer S, the dealer samples $[S]_0$ uniformly at random, sets $[S]_1 \leftarrow S - [S]_0 \bmod 2^{64}$, and then sends $[S]_b$ to shareholder b for $b = 0, 1$. To recover S, the shareholders pool their shares and compute

$$\begin{aligned}[S]_0 + [S]_1 &\equiv [S]_0 + \big(S - [S]_0\big) \\ &\equiv S \pmod{2^{64}}.\end{aligned}$$

Additive secret sharing is linearly homomorphic: Given additive sharings $[S] \coloneqq ([S]_0, [S]_1)$ and $[T] \coloneqq ([T]_0, [T]_1)$ alongside non-secret scalars c and d,

$$\begin{aligned}\big(c \cdot [S]_0 + d \cdot [T]_0\big) + \big(c \cdot [S]_1 + d \cdot [T]_1\big) &= c \cdot \big([S]_0 + [S]_1\big) \\ &\quad + d \cdot \big([T]_0 + [T]_1\big) \\ &\equiv c \cdot S + d \cdot T \pmod{2^{64}},\end{aligned}$$

so that $\big(c \cdot [S]_0 + d \cdot [T]_0, c \cdot [S]_1 + d \cdot [T]_1\big)$ is a $(2, 2)$-additive sharing of the linear combination $c \cdot S + d \cdot T$. Notice that shareholder b can compute $[c \cdot S + d \cdot T]_b \coloneqq c \cdot [S]_b + d \cdot [T]_b$ locally—i.e., without interacting with its peer.

$(2, 2)$-XOR sharing. XOR-shares are just n-tuples of additive shares over $\mathbb{Z}_2$. In this setting "addition", is just the bitwise exclusive-OR operator and, by convention, "multiplication" is the bitwise logical-AND operator. We write $(\!|x|\!) = ((\!|x|\!)_0, (\!|x|\!)_1)$ to denote an XOR-sharing of x (in contrast with writing $[x] = ([x]_0, [x]_1)$ for an additive sharing of the same).

2.3. Beaver multiplication triples

Beaver multiplication triples [1] enable the efficient multiplication of $(2, 2)$-additively shared secrets. Each triple comprises three sharings $\big([X], [Y], [Z]\big)$, where X and Y are uniform random scalars and

$$Z \coloneqq [X]_0 \cdot [Y]_1 + [X]_1 \cdot [Y]_0 \bmod 2^{64}.$$

The shareholders typically precompute Beaver multiplication triples using either additively homomorphic encryption [23] or oblivious transfer [26] during a (rather costly) precomputation phase; alternatively, in the case of $(2 + 1)$-*party computation* (also known as *server-aided* 2*-party computation*), well-formed triples are provided to the shareholders for "free" by a semi-honest third party [11].

Given a pair of sharings $[x]$ and $[y]$ and a Beaver triple $([X], [Y], [Z])$, each shareholder b sends

$$\big([x + X]_b, [y + Y]_b\big) \coloneqq \big([x]_b + [X]_b, [y]_b + [Y]_b\big)$$

to its peer, and then it outputs

$$[z]_b \coloneqq [x]_b \cdot ([y]_b + [y + Y]_{1-b}) - [Y]_b \cdot [x + X]_{1-b} + [Z]_b$$

so that

$$
\begin{aligned}
[z]_0 + [z]_1 &= \big([x]_0 \cdot ([y]_0 + [y+Y]_1) \\
&\qquad\qquad - [Y]_0 \cdot [x+X]_1 + [Z]_0\big) \\
&\quad + \big([x]_1 \cdot ([y]_1 + [y+Y]_0) \\
&\qquad\qquad\quad - [Y]_1 \cdot [x+X]_0 + [Z]_1\big) \\
&= \big(\underline{[x]_0} \cdot ([y]_0 + ([y]_1 + \cancel{[Y]_1})) \\
&\qquad\qquad - \underline{[Y]_0} \cdot (\cancel{[x]_1} + \cancel{[X]_1}) + \cancel{[Z]_0}\big) \\
&\quad + \big(\underline{[x]_1} \cdot ([y]_1 + ([y]_0 + \cancel{[Y]_0})) \\
&\qquad\qquad\quad - \underline{[Y]_1} \cdot (\cancel{[x]_0} + \cancel{[X]_0}) + \cancel{[Z]_1}\big) \\
&= [x]_0 \cdot [y]_0 + [x]_0 \cdot [y]_1 + [x]_1 \cdot [y]_1 + [x]_1 \cdot [y]_0 \\
&= x \cdot y\,.
\end{aligned}
$$

Beaver triples are ephemeral, each enabling just a single multiplication. The multiplication itself is agnostic to whether x and y represent "actual" integers or fixed-point numbers. Of course, in the latter case, multiplication will increase the fractional bits in z relative to x and y, possibly necessitating a subsequent fractional bit reduction.

2.4. Fractional-bit reduction for shared secrets

Reducing the fractional precision of a $(2, 2)$-additively shared fixed-point number $[z]$ is similar to—albeit somewhat more tedious than—directly reducing that of z. Recall that in a two's-complement encoding, the most-significant bit of z is a *sign bit* with $\mathrm{msb}(z) = 1$ if z is negative and $\mathrm{msb}(z) = 0$ if it is non-negative. Consequently, the most significant p bits of $z \gg p$ are each "redundant" copies of the original sign bit (which is now the $(p+1)$th-most-significant bit).

To reduce the number of fractional bits in $[z] = ([z]_0, [z]_1)$, each shareholder b computes

$$[\tilde{z}]_b \coloneqq \big([z]_b \gg p\big)\,.$$

From here, there are three cases to consider:

Case 1 ($\mathrm{msb}(z) = 0$; $\mathrm{msb}([z]_0) = \mathrm{msb}([z]_1) = 1$): Here $[z]_0 + [z]_1$ overflows (carries out from the most-significant bit) so that reconstructing z entails an *implicit* reduction modulo 2^{64}. Furthermore, each of the p most-significant bits of $[\tilde{z}]_b$ are set; hence, in the sum $[\tilde{z}]_0 + [\tilde{z}]_1$, the carry-out from the $(p+1)$th-most-significant bit induces a carry chain that leaves the p leftmost bits errantly set. Consequently,

$$[\tilde{z}]_0 + [\tilde{z}]_1 + 2^{64-p} \equiv z \pmod{2^{64}}\,.$$

Case 2 ($\mathrm{msb}(z) = 1$; $\mathrm{msb}([z]_0) = \mathrm{msb}([z]_1) = 0$): This is similar to the first case, except now signs are flipped so that

$$[\tilde{z}]_0 + [\tilde{z}]_1 - 2^{64-p} \equiv z \pmod{2^{64}}\,.$$

Case 3 ($\mathrm{msb}(z) = \mathrm{msb}([z]_0)$ or $\mathrm{msb}(z) = \mathrm{msb}([z]_1)$): It is easy to check that

$$[\tilde{z}]_0 + [\tilde{z}]_1 \equiv z \pmod{2^{64}}$$

always holds in this case.

The first two cases require an additional correction, wherein the shareholders conditionally (and obliviously) add $[\pm 2^{64-p}]$ to $[\tilde{z}]$ to get $[z \gg p]$. There exist a multitude of options for how to implement this conditional correction; however, they all require one or more rounds of interaction. Computationally, the "best" case occurs when $\mathrm{msb}(z)$ is known (say, because application logic allows its deduction) so that what correction to apply depends solely on *either* $\mathrm{msb}([z]_0) \wedge \mathrm{msb}([z]_1)$ *or* $\neg\mathrm{msb}([z]_0) \wedge \neg\mathrm{msb}([z]_1)$. For the general case with B-bit integers where $\mathrm{msb}(z)$ is *not* known, a simple calculation confirms that we can always use

$$[\pm 2^{B-p}] \coloneqq 2^{B-p} \cdot \big(Z_0 \cdot Z_1 + (Z_0 + Z_1 - 2 \cdot Z_0 \cdot Z_1 - 1) \cdot [Z]\big)\,, \qquad (1)$$

where $Z_b = \mathrm{msb}([z]_b)$ for $b = 0, 1$ and $Z = \mathrm{msb}(z)$.

To illustrate why such "corrections" are needed, we consider the problem of resetting the number of fractional bits in the area of a circle. Let $A =$ `0x0004e8a270000000`, as computed in the prequel, and consider the $(2, 2)$-additive sharing of A via

$$[A]_0 = \texttt{0x80014bf61ed29a6b}$$

and

$$[A]_1 = \texttt{0x80039cac512d6595}\,.$$

Notice that $\mathrm{msb}([A]_0) = \mathrm{msb}([A]_1) = 1$ whereas $\mathrm{msb}(A) = 0$, yet $[A]_0 + [A]_1 = A$ over $\mathbb{Z}_{2^{64}}$. Then

$$\big(A \gg 32\big) = \texttt{0x}\overbrace{\texttt{0000000}}^{\text{redundant sign bits}}\underbrace{\texttt{00004}}_{\text{original sign bit + integer part}}\overbrace{\texttt{e8a2}}^{\text{fractional part}},$$

while

$$
\begin{aligned}
[\tilde{A}]_0 &= \big([A]_0 \gg 32\big) \\
&= \texttt{0x}\underbrace{\texttt{ffffffff}}_{\text{redundant sign bits}}\overbrace{\texttt{80014bf6}}^{\text{shifted share}}
\end{aligned}
$$

and

$$
\begin{aligned}
[\tilde{A}]_1 &= \big([A]_1 \gg 32\big) \\
&= \texttt{0x}\underbrace{\texttt{ffffffff}}_{\text{redundant sign bits}}\overbrace{\texttt{80039cac}}^{\text{shifted share}},
\end{aligned}
$$

so that

$$
\begin{aligned}
\big([\tilde{A}]_0 + [\tilde{A}]_1\big) + 2^{64-32} &\equiv \big(\texttt{0xffffffff80014bf6} \\
&\quad + \texttt{0xffffffff80039cac}\big) \\
&\quad + \texttt{0x0000000100000000} \\
&\equiv \texttt{0x}\textcolor{red}{\texttt{ffffffff}}\texttt{0004e8a2} \\
&\quad + \texttt{0x0000000}\textcolor{red}{\texttt{1}}\texttt{00000000} \\
&\equiv \texttt{0x000000000004e8a2} \pmod{2^{64}}\,.
\end{aligned}
$$

Mohassel and Zhang prove [22; Appendix B] that each of Cases 1 and 2 only occurs with probability negligible in the number of "extra" integer bits; thus, if program logic suffices to prove that integer parts are sufficiently small, P_0 and P_1 can forgo explicit corrections and still get the correct result with very high probability.

3. Selection vectors

A *selection vector* is a vector in which one element is 1 and all others are 0. We refer to the length-N selection vector having its 1 in position $i \in [0\,..\,N)$ as the *ith selection vector of length N*.

Observation 1. *All selection vectors of a given length are equivalent up to cyclic rotation. Specifically, for any $i, j \in$*

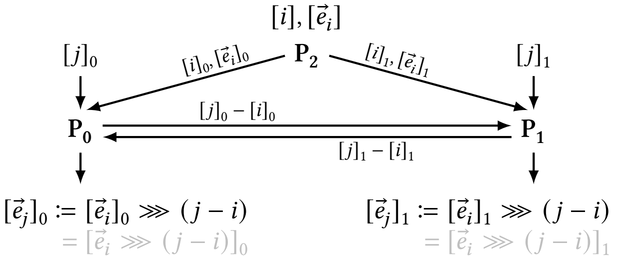


Figure 1: A (2+1)-party protocol for converting an additive sharing $[j]$ of a scalar $j \in [0..N)$ into an additive sharing of the jth selection vector $\vec{e}_j$ of length N.

$[0..N)$, if $\vec{e}_i$ is the ith selection vector of length N, then $\vec{e}_j = \vec{e}_i \ggg (j-i)$ is the jth selection vector of length N.

Observation 1 is especially relevant when the selection vector is secret shared: If P_0 and P_1 hold additive sharings $[i]$ and $[j]$ of two numbers from the ring of integers modulo N alongside a sharing of the ith selection vector of length N, then they can arrive at a sharing of the jth selection vector of length N as follows. First, they leverage linearity to learn

$$\begin{aligned}(j-i) \bmod N &= \big(([j]_0 + [j]_1) - ([i]_0 + [i]_1)\big) \bmod N \\ &= \big(\underbrace{([j]_0 - [i]_0)}_{P_0 \text{ shares}} + \underbrace{([j]_1 - [i]_1)}_{P_1 \text{ shares}}\big) \bmod N \end{aligned} \quad (2)$$

without revealing i or j individually, after which each party cyclically rotates its own share of $\vec{e}_i$ to the right by this quantity.[1] Notice that if i is uniform, then $(j-i) \bmod N$ perfectly hides j, making this transformation from the ith into the jth selection vector perfectly oblivious.

Scalar-to-selection vector share conversion. Observation 1 suggests the following $(2+1)$-party protocol for converting additive sharings of j into additive sharings of jth selection vectors. Let N be given. In a preprocessing phase, a semi-trusted third party P_2 chooses $i \in [0..N)$ uniformly and provides additive sharings $[i]$ and $[\vec{e}_i]$ to the first parties P_0 and P_1, where $\vec{e}_i$ is the ith selection vector of length N. Upon learning the sharing $[j]$ in the online phase, P_0 and P_1 interactively reconstruct $(j-i) \bmod N$ using Equation (2) and then they compute shares of $\vec{e}_j$ via $[\vec{e}_j] \coloneqq [\vec{e}_i] \ggg (j-i) \bmod N$. A diagrammatic view of this $(2+1)$-party protocol is included as Figure 1.

PIR from selection vectors. As their name hints, selection vectors are useful for selecting items from a list. For example, consider a lookup table (LUT) mapping each of the first N non-negative integers to a scalar outcome. By encoding this LUT as a length-N vector $\vec{P}$ in the obvious way and taking an inner product with $\vec{e}_j$, we find that $\langle \vec{e}_j, \vec{P} \rangle = P_j$, where P_j is the image of j in the LUT. Moreover, because inner products are linear, the first parties to the above $(2+1)$-party protocol can likewise *obliviously* fetch the sharing $[P_j]$ from $\vec{P}$ using $[\vec{e}_j]$. Astute readers may recognize this procedure as a variant of 2-server *private information retrieval* (PIR) [10] over $\vec{P}$ in which the "client" P_2 pre-distributes random queries to "servers" P_0 and P_1 in an offline phase.

As an alternative to rotating $[\vec{e}_i]$ to the right by $(j-i)$, P_0 and P_1 can instead rotate $\vec{P}$ to the left by the same distance. Taking inner products as before, this alternative is guaranteed to produce the same result because (i) commutativity implies inner products are invariant under cyclic reordering of summands, and (ii) left and right cyclic rotations are mutually inverse operations, so that

$$\begin{aligned}\langle \vec{e}_j, \vec{P} \rangle &= \langle \vec{e}_i \ggg (j-i), \vec{P} \rangle \\ &= \langle \big(\vec{e}_i \ggg (j-i)\big) \lll (j-i), \vec{P} \lll (j-i) \rangle && \text{via (i)} \\ &= \langle \vec{e}_i, \vec{P} \lll (j-i) \rangle. && \text{via (ii)}\end{aligned}$$

Figure 2 illustrates the equivalence between these two options. We will make use of this equivalence later on.

Function evaluation via PIR. If $f\colon \mathbb{Z}_N \to \mathbb{Z}_N$ is some function and $\vec{P}$ its truth table (that is, $P_j \coloneqq f(j)$ for all $j \in \mathbb{Z}_N$), then the above steps realize a $(2+1)$-party oblivious evaluation of $f(j)$ at the secret input j. When N is small (and f nonlinear), this procedure can perform well relative to evaluating $f(j)$ directly using arithmetic (or Boolean) circuits [14], [31], [33]. We stress that the $\mathbb{Z}_N$ elements may represent fixed-point numbers so that this is not limited to the oblivious evaluation of integer-valued functions.

As a potentially significant optimization, wherever f is *constant* within some interval, P_0 and P_1 can apply the distributive law to save some work in the inner product calculation. As an extreme example of this in action, suppose that $f\colon \mathbb{Z}_N \to \mathbb{Z}_N$ is the step function defined by

$$f(j) \coloneqq \begin{cases} A & \text{if } j \in [0..5), \text{ and} \\ B & \text{otherwise,} \end{cases}$$

so that $\vec{P} = (A, A, A, A, A, B, \ldots, B)$. Then the inner product between $[\vec{e}_j]_b = ([e_{j0}]_b, [e_{j1}]_b, \ldots, [e_{jN}]_b)$ and $\vec{P}$ has the very simple form

$$\begin{aligned}\langle [\vec{e}_j]_b, \vec{P} \rangle = A\cdot([e_{j0}]_b + [e_{j1}]_b + [e_{j2}]_b + [e_{j3}]_b + [e_{j4}]_b) \\ + B\cdot([e_{j5}]_b + \ldots + [e_{jN}]_b),\end{aligned}$$

which can be evaluated using $N-1$ additions and just two scalar multiplications. For comparison, a naïve evaluation would require $N-1$ additions and N scalar multiplications. (Furthermore, if, e.g., $B = 0$, then the cost further shrinks to just 4 additions and a single scalar multiplication.)

Function evaluation via <u>binary</u> selection vectors. So far, we have assumed that length-N selection vectors $\vec{e}_j$ are shared additively over $\mathbb{Z}_N$; however, this is not a formal requirement. Indeed, because selection vectors consist solely of elements from $\{0, 1\}$, they can be shared more compactly as length-N bitstrings (i.e., as length-N vectors over $\mathbb{Z}_2$). This shaves a factor $\lceil \lg N \rceil$ from the size of $(\!|\vec{e}_j|\!)$ relative to that of $[\vec{e}_j]$, but not without introducing a minor technicality: Before they can evaluate the inner product between $(\!|\vec{e}_j|\!)$ and $\vec{P}$, the shareholders P_0 and P_1

1. This approach is mathematically sound for additive sharings because cyclic rotation is a linear operation, namely multiplication by a cyclic permutation matrix (i.e., a cyclic rotation of the identity matrix).

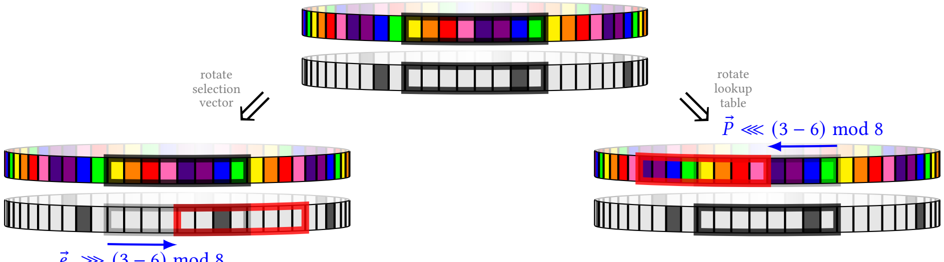


Figure 2: Equivalence between rotating selection vectors rightward versus LUTs leftward. In the diagram, the selection vector is $\vec{e}_6$ and the desired record is P_3 (the red element). The left subdiagram shows the outcome of rotating $\vec{e}_6$ rightward to get $\vec{e}_3$; the right subdiagram shows the outcome rotating $\vec{P}$ leftward to move the red element into position 6.

must first *lift* each bit of $(\!|\vec{e}_j|\!)$ into an additive sharing over $\mathbb{Z}_N$. In general, such lifting is costly, requiring a round of interaction between P_0 and P_1 to ensure the resulting additive shares all have the correct *signs* in $\mathbb{Z}_N$ (indeed, it is impossible to differentiate among ± 1 in $\mathbb{Z}_2$).

Fortunately, the special form of selection vectors makes it possible for the shareholders to defer the latter interaction needed for sign correction to a *post-processing* step, to be performed only *after* evaluating the inner product.[2] In particular, for each of $b = 0, 1$, shareholder P_b lifts the ith bit $(\!|e_{ji}|\!)_b$ of $(\!|\vec{e}_j|\!)_b$ into an additive share over $\mathbb{Z}_N$ via

$$[\pm e_{ji}]_b \coloneqq \begin{cases} 0 & \text{if } (\!|e_{ji}|\!)_b = 0\text{, and} \\ (-1)^b & \text{otherwise,} \end{cases} \tag{3}$$

so that $[\pm e_{ji}]_0 + [\pm e_{ji}]_1 \in \{-1, 0, 1\}$. Now, the inner product $\langle [\pm \vec{e}_j], \vec{P} \rangle$ yields $\pm P_j$, a scalar that is correct up to sign. From here, there are a few options for how to implement the sign correction; we defer our discussion of those techniques to Section 7.1.1.

The earlier optimization for when $\vec{P}$ is constant over some interval ports nicely to the case where $\vec{e}_j$ is shared bitwise as $(\!|\vec{e}_j|\!)$: The shareholders simply perform the required summation over $\mathbb{Z}_2$ and then convert the resulting sums (i.e., not the individual addends) into additive shares over $\mathbb{Z}_N$ using Equation (3). Notice that computing such sums of segments of $(\!|\vec{e}_j|\!)$ is equivalent to computing the *parities* of the bitstrings corresponding to those segments.

As a concrete example, let us consider the evaluation of the step function f from the earlier example with vector length $N = 8$ and input $j = 4$. Supposing $\vec{e}_4$ is shared as $(\!|\vec{e}_4|\!) = (1101\underline{1}010, 1101\underline{0}010)$, shareholder P_0 computes

$$\big(1 \oplus 1 \oplus 0 \oplus 1 \oplus \underline{1}, 0 \oplus 1 \oplus 0\big) = \big(\text{parity}(1101\underline{1}), \text{parity}(010)\big) = (\underline{0}, 1),$$

while shareholder P_1 computes

$$\big(1 \oplus 1 \oplus 0 \oplus 1 \oplus \underline{0},\ 0 \oplus 1 \oplus 0\big) = \big(\text{parity}(1101\underline{0}), \text{parity}(010)\big) = (\underline{1}, 1).$$

Upon lifting these values to $\mathbb{Z}_N$, the shareholders respectively hold vectors $[-\vec{e}_4]_0 \coloneqq (0, 1)$ and $[-\vec{e}_4]_1 \coloneqq (-1, -1)$, from which they evaluate

$$\langle [-\vec{e}_4]_0, \vec{P} \rangle = 0 \cdot a + 1 \cdot b = b$$

and

$$\langle [-\vec{e}_4]_1, \vec{P} \rangle = (-1) \cdot a + (-1) \cdot b = (-a) + (-b),$$

and we find that $b + \big((-a) + (-b)\big) = -a$, the negation of $f(4)$. A sign correction completes the process.

Spline evaluation via selection vectors. Up till now, we have considered LUTs only for scalar-valued functions, yet the technique generalizes seamlessly to vector- or matrix-valued functions. As one useful application of this, we can evaluate functions $\mathcal{F}\colon \mathbb{Z}_N \to (\mathbb{Z}_N)^{1\times d}$ that output (vectors of coefficients defining) polynomials, including *piecewise-linear functions* and general *splines*. As a bonus, in light of the optimizations already discussed, such piecewise functions typically result in comparatively inexpensive inner product computations (i.e., requiring far fewer than N multiplications, since each "part" covers a non-trivial subinterval of the domain).

To see why this is useful, suppose we wish to approximate some highly nonlinear function $f(x)$ that is prohibitively costly to evaluate exactly using arithmetic circuits. We can do so by constructing a piecewise-polynomial function $\mathcal{F}$ such that, for all $j \in \mathbb{Z}_N$, the coefficients vector $\langle a_d, \ldots, a_1, a_0 \rangle \leftarrow \mathcal{F}(j)$ defines a good low-degree-polynomial approximation $f'(x) = a_d x^d + \cdots + a_1 x + a_0$ to $f(x)$ in the vicinity of $x = j$. Given additive sharings of such a coefficient vector $[\mathcal{F}(j)] = [f'(\cdot)]$ and of the input $[j]$, several well-known techniques can obliviously compute $[f'(j)]$, thereby obtaining shares of a good approximation to $f(j)$. We describe two such techniques in Sections 7.1.1 and 7.1.2, respectively based on Horner's

2. Specifically, since $\vec{e}_j$ consists entirely of 0s save for the 1 in position j, the initial lifting of $(\!|\vec{e}_j|\!)$ into $\mathbb{Z}_N$ yields $[\pm \vec{e}_j]$; that is, the requisite sign correction can occur at the granularity of the *entire vector*. For vectors with two or more non-zero entries, this would not be true.

method [16] together with Du–Atallah multiplication [11] and on ABY2.0-style D-ary multiplication [25].

4. Parity-segment trees

A *parity-segment tree* is a data structure for answering parity queries over the substrings of a binary string, with a worst-case complexity logarithmic in the bitstring's length. That is, given the parity-segment tree $T(x)$ for a length-N bitstring $x = x_0 x_1 \cdots x_{N-1}$, a parity query for the substring $x[a\,..\,b)$ of x, $0 \le a < b \le N$, returns the parity

$$\mathrm{parity}(x[a\,..\,b)) \coloneqq \bigoplus\nolimits_{i=a}^{b-1} x_i$$

with a running time in $\mathrm{O}(\lg N)$.

Constructing the parity-segment tree $T(x)$ for a given bitstring x is straightforward, if tedious. For ease of exposition, suppose that x has length $N = 2^{n+k}$ for some nonnegative integers n and k and that each leaf node represents $\lambda = 2^k$ consecutive bits of x. Notice that $\lambda \mid N$ by construction.

Given x, we construct the tree $T(x)$ from the bottom up: To form the base of the tree, split x into $N/\lambda = 2^n$ many λ-bit substrings and then insert one leaf node for each of these substrings, storing its parity inside the node. Next, for each successive pair of leaf nodes, insert a parent and store within it the combined parity of its two children. Now repeat this process on each successive pair of parents, and so on, until a single root emerges (after $n-1$ recursions). Notice that the parity bit held by any given node equals the parity of a concatenation over all substrings of x associated with leaves descendant from that node.

It follows easily by inspection that constructing the parity-segment tree $T(x)$ from x requires the computation of (N/λ)-many λ-bit parities (for the leaf nodes) plus $2^n - 1$ single-bit XORs (for the interior nodes), giving a total complexity of $\mathrm{O}(N)$ bit operations; the tree itself occupies $2^{n+1} - 1 \in \mathrm{O}(N/\lambda)$ bits.

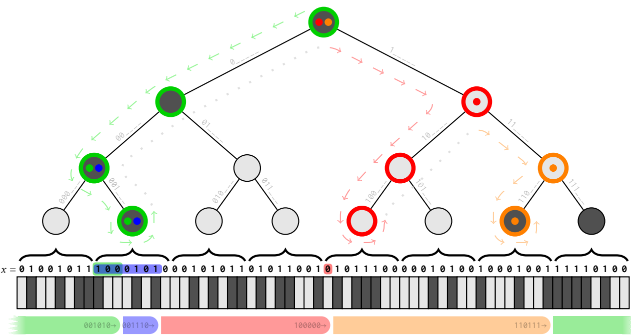


Figure 3: The parity-segment tree for a 64-bit string with 8-bit leaf nodes (i.e., $N = 2^{3+3}$ and $\lambda = 2^3$).

4.1. Computing segment parities

Figure 3 shows the parity-segment tree for an arbitrary 64-bit string x, which is written immediately below the tree. In this toy example, each leaf node is associated with a 1-byte (8-bit) substring—yielding $64/8 = 8$ leaf nodes in total—and holds the parity of that substring internally. Likewise, each non-leaf node holds 1 bit indicating the parity of its immediate descendants. We use light shading (e.g., ◯) to indicate a node holds even parity (the bit is 0) and dark shading (e.g., ●) to indicate it holds odd parity (the bit is 1).

Beneath the bitstring, we draw several half-open segments that collectively partition the bitstring into four contiguous (up to cyclic rotation) substrings, totally ordered by their rightmost endpoints. With 0-based indexing, the first (**green**) segment ends after bit 10; the second (**blue**) segment after bit 14; the third (**red**) segment after bit 32; and the fourth (**orange**) segment after bit 55.

The diagram also contains several pictorial annotations conveying information about how our *prefix-parity algorithm* helps to find the parity of each segment. As its name suggests, the prefix-parity algorithm computes the parity of each substring using the parities of *prefixes* of x sharing the same rightmost endpoints as the desired segments. In this example, it finds each of the prefix parities $\mathrm{parity}(x[0\,..\,11))$, $\mathrm{parity}(x[0\,..\,15))$, $\mathrm{parity}(x[0\,..\,33))$, and $\mathrm{parity}(x[0\,..\,56))$. A subsequent post-processing phase exploits the nilpotency of XOR to compute the desired segment parities from these prefix parities via

$$\begin{aligned}
\mathrm{parity}(\textbf{green}) &= \mathrm{parity}(x[56\,..\,64)) \oplus \mathrm{parity}(x[0\,..\,11)) \\
&= \big(\mathrm{parity}(x) \oplus \mathrm{parity}(x[0\,..\,56))\big) \\
&\qquad \oplus \mathrm{parity}(x[0\,..\,11))\,; \\
\mathrm{parity}(\textbf{blue}) &= \mathrm{parity}(x[11\,..\,15)) \\
&= \mathrm{parity}(x[0\,..\,15)) \oplus \mathrm{parity}(x[0\,..\,11))\,; \\
\mathrm{parity}(\textbf{red}) &= \mathrm{parity}(x[15\,..\,33)) \\
&= \mathrm{parity}(x[0\,..\,33)) \oplus \mathrm{parity}(x[0\,..\,15))\,; \text{ and} \\
\mathrm{parity}(\textbf{orange}) &= \mathrm{parity}(x[33\,..\,56)) \\
&= \mathrm{parity}(x[0\,..\,56)) \oplus \mathrm{parity}(x[0\,..\,33))\,.
\end{aligned}$$

In the diagram, a node is drawn with a thick coloured outline (e.g., ◯) if the prefix-parity algorithm visits that node during the computation of one or more prefix parities. We employ a memoization (and backtracking) strategy that ensures each node is visited at most once throughout the computation of all prefix parities; the outline's colour indicates which prefix the algorithm is computing when it *first* visits that node. The dashed-and-dotted path emanating from the root likewise shows the traversal order through the tree; we decorate the path with coloured-arrow dashes (e.g., → →) when the traversal is visiting new nodes and with faint gray dots (e.g., · · ·) when "backtracking" to a previously visited (memoized) node. We place a thick coloured dot (e.g., ⊙) within a node if the 1-bit parity stored at that node appears as an operand when computing the prefix parity for the correspondingly coloured segment; moreover, we highlight a prefix of the λ-bit substring associated with a leaf if the parity of that prefix also appears as an operand in the prefix-parity computation.

The prefix-parity algorithm performs a binary search–like traversal through $T(x)$, employing a simple inclusion-exclusion strategy to compute a sequence of

TABLE 1: The sequences of prefixes (and associated parities) of x considered (accounting for memoization) while computing the four prefix parities arising in Figure 3.

level	green	blue	red	orange
root → 0	$x[0..0)$ ∘	$x[0..0)$ ∘	$x[0..64)$ •	$x[0..64)$ •
1	$x[0..0)$ ∘	$x[0..0)$ ∘	$x[0..32)$ •	$x[0..64)$ •
2	$x[0..16)$ •	$x[0..16)$ •	$x[0..32)$ •	$x[0..48)$ •
leaf → 3	$x[0..8)$ ∘	$x[0..8)$ ∘	$x[0..32)$ •	$x[0..56)$ •
substring → 4	$x[0..11)$ •	$x[0..15)$ •	$x[0..33)$ •	—

parities of prefixes that alternately over- and undershoot the desired prefix. By adopting the convention that one always "traverses left" both (i) to arrive at the root and (ii) to access the substring associated with a leaf node, we obtain the following procedure:

1) initialize a "running parity" to 0;
2) starting from the root, traverse to the leaf node associated with the rightmost bit in the prefix;
3) wherever the root-to-leaf path changes directions, update the running parity by XORing in the parity stored at the node where the change-of-direction occurs; and
4) finally, XOR in any bits of the prefix that reside in the substring associated with the leaf node.

As a modest optimization, one can terminate the traversal early if ever the rightmost endpoint is one bit past the end of the rightmost descendant of the left child of the node presently being traversed (as in such cases, the running parity is already guaranteed to be correct).

Annotated C-like pseudocode for the prefix-parity algorithm—incorporating both the above *early-termination optimization* as well as the bookkeeping needed for effective memoization—is included as Appendix B.

Illustrated walkthroughs for Figure 3. We strategically chose the segments in Figure 3 to illustrate some notable sub-cases, namely (i) cyclically wrapping segments (**green**), (ii) two segments terminating at the same leaf node (**green** and **blue**), (iii) segments terminating immediately following a leaf node (**orange**), and (iv) the "typical" case where a segment is alone in terminating partway through some leaf (**red**). We remark on the implications of these cases in the algorithm walkthroughs below.

Prefix 1 (green): The first segment ends 3 bits into the second leaf node. Our algorithm traverses leftward twice to arrive at the parent of that node. Since the next traversal goes right (a change of direction), it reads the parity bit (odd) within that parent. After traversing right to the second leaf node, it must traverse left to access the associated substring; thus, it XORs in that leaf node's parity bit (also odd). Finally, it inspects the substring beneath that leaf node, XORing in the parity of its 3-bit prefix (odd yet again). The resulting parity is therefore

$$\begin{aligned} \mathrm{parity}(x[0..11)) &= \mathrm{parity}(\texttt{01001011 10001010}) && 1 \\ &\oplus \mathrm{parity}(\texttt{00000000 10001010}) && \oplus 1 \\ &\oplus \mathrm{parity}(\texttt{00000000 10000000}) && \oplus 1 \\ &= \mathrm{parity}(\texttt{01001011 10000000}) && = 1\,. \end{aligned}$$

In total, the algorithm visits four nodes (and examines one λ-bit substring) to compute this prefix parity.

Prefix 2 (blue): The second segment also ends part of the way through the second leaf; consequently, the algorithm reuses almost the entire first parity computation, merely substituting in a longer substring prefix in the last step. In total, the algorithm visits zero new nodes (and examines one λ-bit substring) to compute this prefix parity.

Prefix 3 (red): The third segment extends one bit into the right subtree of the root (a direction change at the outset). Hence, the algorithm XORs the parity stored in the root (which captures the parity of the *entire* string x) together with the parity stored within the root's right child (which captures the parity of the *second half* of x), after which it holds the parity of the first half of x. For the remaining bit, it traverses to the leftmost leaf beneath the right child of the root (which involves no direction changes), and examines the single bit of the associated substring that is part of the segment. In total, the algorithm visits three new nodes (and examines one λ-bit substring) to compute this prefix parity.

Prefix 4 (orange): The fourth prefix terminates immediately after the substring in the second-last leaf node. It XORs the memoized 1-bit parity from the root together with the parity stored within the second-last leaf node and its parent, after which it holds the desired parity. In total, the algorithm visits two new nodes (and does not examine any substring) to compute this prefix parity.

All told, the prefix-parity algorithm in this example visited $4+0+3+2=9$ (out of 15) nodes and examined $1+0+1+0=2$ (out of 8) distinct λ-bit substrings of x (computing $1+1+1+0=3$ substring-prefix parities).

Table 1 lists the sequence of prefixes of x whose parities are computed (and memoized) as the prefix-parity algorithm computes the above four prefix parities. In the table, segment parities are colored gray wherever a memoized value is in use; the arrows indicate where each memoized value was most recently used.

4.2. Analysis

Our primary concern in the sequel will be how many distinct *edges* the prefix-parity algorithm must traverse—and, to a lesser extent, how many λ-bit *substrings* it must examine—for a given partitioning of x into segments. The next theorem characterizes the worst-case cost for the number of edges (assuming optimal memoization).

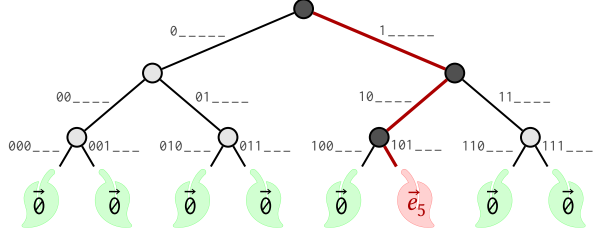


Figure 4: Binary-tree representation for the 45th point function on $\mathbb{Z}_{2^{3+3}}$. Each leaf holds either a zero vector or a selection vector of length 2^3.

Theorem 1. *Given a parity-segment tree $T(x)$ of height n and a lexicographically sorted list E of S distinct prefix endpoints, the prefix-parity algorithm traverses at most $Sn - \sum_{i=2}^{S}\lfloor \lg(i-1)\rfloor$ edges to compute all S prefix parities.*

Proof of Theorem 1 is included as Appendix A.

Theorem 1 implies that the prefix-parity algorithm visits $o(Sn)$ edges to compute S prefix parities (of a given 2^{n+k}-bit string). We also note that tree $T(x)$ has just $2^{n+1}-2$ edges in total, which yields another upper bound on the number of edges traversed. Notably, as S approaches the length $N = 2^{n+k}$ of x, the amortized number of edges traversed per prefix tends to 2^{-k+1} as each of the $2^{n+1}-2$ edges is traversed exactly once (owing to memoization).[3]

The theorem deals with worst-cast costs. The *expected* number of edges traversed depends on the distribution of the prefixes. Generally speaking, more densely packed prefix endpoints (i.e., shorter segments) imply greater amortization savings. Appendix C plots empirically measured edge-traversal counts for sets of endpoints sampled from a handful of standard probability distributions.

Expected savings from early termination. Notice that the "early-termination" optimization saves exactly $i+1$ traversals (this includes the "traversal" from a leaf node to its associated substring) if and only if the endpoint is a multiple of $2^i{\cdot}\lambda$ but not of $2^{i+1}{\cdot}\lambda$. The next theorem follows easily from this observation.

Theorem 2. *For a uniform random endpoint $X \in [0..N)$, the early-termination optimization saves $(2-2^{-n})/\lambda$ traversals in expectation.*

Proof. For $i = 1,\dots,n-1$, the probability that uniform X is a multiple of $\lambda{\cdot}2^i$ but not $\lambda{\cdot}2^{i+1}$ is given by $\frac{1}{\lambda\cdot 2^i} - \frac{1}{\lambda\cdot 2^{i+1}} = \frac{1}{\lambda\cdot 2^{i+1}}$; for $i = n$, it is just $\frac{1}{\lambda\cdot 2^n}$. Hence, in expectation, we save

$$\frac{n+1}{\lambda\cdot 2^n} + \sum_{i=0}^{n-1} \frac{i+1}{\lambda\cdot 2^{i+1}} = \frac{2-2^{-n}}{\lambda}$$

traversals (counting "traversals" from leaf nodes to substrings). □

3. Indeed, it is easy to check that $Sn - \sum_{i=2}^{S}\lfloor \lg(i-1)\rfloor = 2^{n+1}-1$.

5. Point functions

A *binary point function* is just a "functional" representation of a selection vector; that is, a Boolean-valued function that has a selection vector as its truth table.

Definition 1. The *ith binary point function on $\mathbb{Z}_N$* is the function $p_i \colon \mathbb{Z}_N \to \{0,1\}$ for which

$$\vec{e}_i \coloneqq \big(p_i(0), p_i(1), \dots, p_i(N-1)\big)$$

is the ith selection vector of length N.

All point functions that we consider in this work are binary point function; thus, for brevity, we herein omit the "binary" qualifier and speak merely of *point functions* on $\mathbb{Z}_N$. If we suppose, as in the preceding section, that $N = 2^{n+k}$ for some non-negative integers n and k, then the ith point function on $\mathbb{Z}_N$ has a natural representation as a full binary tree of height n whose 2^n leaf nodes partition $\vec{e}_i$ into λ-bit segments, where $\lambda = 2^k$. Figure 4 illustrates this representation for the 45th point function on $\mathbb{Z}_{2^{3+3}}$.

The above-described correspondence is the basis for several constructions of so-called *distributed point functions* [4], [6], [12]. We describe one such construction in the sequel; in preparation for this, Section 5.1 first recalls and expands upon some elementary definitions and results from a recent manuscript of Vadapalli, Storrier, and Henry [32; §II.A and §V.A].

5.1. 0/1-leaves, 0/1-nodes, and 1-paths

Let $B(p)$ denote the height-n binary-tree representation of the ith point function on $\mathbb{Z}_N$. Vadapalli et al. assign a discrete "type" to each node of $B(p)$ based on its pedigree. Here we propose a (very modest) generalization of their taxonomy that has been extended to account for the case where $k > 0$ so that each leaf node captures the image of $\lambda > 1$ consecutive inputs to p_i. This lets us use their taxonomy on trees whose structure matches that of general parity-segment trees as defined in Section 4 (as well as that of the most compact distributed point functions in the literature [6]).

Definition 2. A leaf node is called a 1-*leaf* if it holds a λ-bit selection vector; it is called a 0-*leaf* if it holds a λ-bit zero vector.

Notice that, for every point function p, precisely one leaf node of $B(p)$ is a 1-leaf and all others are 0-leaves. We color the sole 1-leaf in Figure 4 (and, likewise, in Figure 5 below) red (i.e.,) while all 0-leaves are green (i.e.,). The following observation about these leaves is immediate.

Observation 2. *The parity of the λ-bit vector held in the 1-leaf is 1 while the parity of the λ-bit vector held in each 0-leaf is 0.*

The notions of 0-leaves and 1-leaves have natural, recursively defined analogues for interior nodes.

Definition 3. An interior node is a 0-*node* if its children are both 0-leaves or both 0-nodes; it is a 1-*node* if its children are either (i) a 1-leaf and a 0-leaf or (ii) a 1-node and a 0-node.

In Figure 4, we use light shading (i.e., ◯) to indicate an interior node is a 0-node and dark shading (i.e., ⬤) to indicate that it is a 1-node. Notice that, by construction, every leaf node descendant from any 0-node is a 0-leaf whereas exactly one leaf node descendant from any 1-node is a 1-leaf (and all others are 0-leaves). From here, we can further define the notion of a 1-*path* as follows.

Definition 4. A sequence of edges in $B(p)$ is a 1-*path* if it originates at a 1-node and terminates at the sole 1-leaf descendent from that 1-node.

Notice that *every* node along a 1-path is either a 1-node or a 1-leaf. An immediate consequence of Definitions 2–4 follows in Theorem 3.

Theorem 3 ([32; Corollary 1])**.** *The following three characterizations are all equivalent: A full binary tree of height n represents a point function if and only if*

1) *exactly one leaf is a 1-leaf and all others are 0-leaves;*
2) *its root is a 1-node; or*
3) *it contains a 1-path of height n.*

Consequent to Bullet 3 of Theorem 3, the tree $B(p)$ has exactly one 1-node at each non-leaf level (and exactly one 1-leaf at the leaf level). Composing this fact with Observation 2 yields the following (relatively obvious) theorem, which serves as one of two lynchpins of our main construction.

Theorem 4. *Fix $b \in \{0,1\}$. If p is the ith point function on $\mathbb{Z}_N$ and $B(p)$ its binary-tree representation, then an interior node X of $B(p)$ is a b-node if and only if the joint parity of the vectors stored in leaves descendant from X is b.*

5.2. Distributed point functions

Intuitively, a *distributed point function* (DPF) is a compact, secret-shared representation of a point function (or, equivalently, of a selection vector). We adapted the following formal definition from Vadapalli et al. [32; Definition 4], with minimal modifications to specialize it for the case of $(2,2)$*-DPFs with* 1*-bit outputs.*

Definition 5. A $(2,2)$*-distributed point function*, or $(2,2)$*-DPF*, with 1-bit outputs is an ordered pair of PPT algorithms $(\mathsf{Gen}, \mathsf{Eval})$ defining an infinite family of secret-shared representations of point functions; that is, given (i) a security parameter $\lambda \in \mathbb{N}$, (ii) an upper bound $N \in \mathbb{N}$ for the domain, and (iii) a distinguished input $i \in [0\,..\,N)$, we have

1. **Correctness:** If $(\llbracket\vec{e}_i\rrbracket_0, \llbracket\vec{e}_i\rrbracket_1) \leftarrow \mathsf{Gen}(1^\lambda, N;\, i)$, then, for all $j \in [0\,..\,N)$,
$$\mathsf{Eval}(\llbracket\vec{e}_i\rrbracket_0, j) \oplus \mathsf{Eval}(\llbracket\vec{e}_i\rrbracket_1, j) := \begin{cases} 1 & \text{if } j = i \text{, and} \\ 0 & \text{otherwise.} \end{cases}$$
2. **Simulatability**: There exists a PPT simulator $\mathcal{S}$ such that, for any given N, distinguished input $i \in [0\,..\,N)$, and bit $b \in \{0,1\}$, the distribution ensembles
$$\big\{\mathcal{S}(1^\lambda, N;\, b)\big\}_{\lambda\in\mathbb{N}}$$
and
$$\big\{\llbracket\vec{e}_i\rrbracket_b \;\big|\; (\llbracket\vec{e}_i\rrbracket_0, \llbracket\vec{e}_i\rrbracket_1) \leftarrow \mathsf{Gen}(1^\lambda, N;\, i)\big\}_{\lambda\in\mathbb{N}}$$
are computationally indistinguishable.

The $\llbracket\vec{e}_i\rrbracket_b$ output by Gen are called *DPF shares (of $\vec{e}_i$)* or *DPF keys.*

The next subsection recalls the Boyle–Gilboa–Ishai construction for $(2,2)$-DPFs with 1-bit outputs [6]. Readers should take note of the structural relationships among (i) the generation and evaluation of such DPFs, (ii) the parity-segment trees from Section 4, and (iii) the taxonomical treatment of point function–tree nodes from Section 5.1. Jumping ahead, we observe that *DPF shareholders can compute XOR-shared segment parities directly from their respective DPF shares* while incurring only a surprisingly low (asymptotic and concrete) cost and without any need for interaction.

5.2.1. The BGI construction. At the heart of the Boyle–Gilboa–Ishai (BGI) construction is the following elementary observation about pseudorandom generators (PRGs) seeded with (pseudo)randomly-sampled XOR sharings.

Observation 3 (PRGs preserve "zeroness")**.** *Let $\{\mathrm{G}_\lambda\}_{\lambda\in\mathbb{N}}$ be a length-doubling PRG family and consider $(L,R) \leftarrow \mathrm{G}_\lambda([z]_0) \oplus \mathrm{G}_\lambda([z]_1)$. If $z = 0^\lambda$, then $L = R = 0^\lambda$; otherwise, if $z \neq 0^\lambda$, then both $L \neq 0^\lambda$ and $R \neq 0^\lambda$ with a probability overwhelming in λ.*

In other words, either both halves of the output are equal (because the inputs were equal), or neither half is equal (because the inputs were unequal)—at least with a very high probability. The idea from here is to use G_λ as a black box to build what we call a *pseudorandom traversal function*, wherein it is easy to *force* equality for a chosen half of the output while ensuring that "inadvertent equalities" in the other half remain cryptographically rare.

Definition 6. Let $\{\mathrm{G}_\lambda\}_{\lambda\in\mathbb{N}}$ be a length-doubling PRG family. The *pseudorandom traversal function family from* $\{\mathrm{G}_\lambda\}_{\lambda\in\mathbb{N}}$ is the infinite family $\{\widetilde{\mathrm{G}}_\lambda\}_{\lambda\in\mathbb{N}}$ of functions $\widetilde{\mathrm{G}}_\lambda\colon \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2 \times \big(\mathbb{Z}_2^{\lambda-1} \times (\mathbb{Z}_2)^2\big) \to \mathbb{Z}_2^\lambda \times \mathbb{Z}_2^\lambda$ such that
$$\widetilde{\mathrm{G}}_\lambda(\underbrace{s}_{\lambda-1\text{ bits}}, \overbrace{advice}^{1\text{ bit}}, \underbrace{cw}_{(\lambda-1)+2\text{ bits}}) := \begin{cases} \mathrm{G}_\lambda(s\|0) & \text{if } advice = 0\text{, and} \\ \mathrm{G}_\lambda(s\|0) \oplus (cw_L, cw_R) & \text{if } advice = 1, \end{cases}$$
where $cw_L := \overline{cw}\|t_L$ and $cw_R := \overline{cw}\|t_R$ for $cw = (\overline{cw}, t_L, t_R)$.

An ordered couple $((s_0, advice_0, cw), (s_1, advice_1, cw))$ of $\widetilde{\mathrm{G}}_\lambda$ inputs that share a common cw term is called an *input pair*. Intuitively, the $advice_0$ and $advice_1$ bits of an input pair indicate whether or not to "correct" the output of G_λ (via perturbing it by cw) before returning it from $\widetilde{\mathrm{G}}_\lambda$ when the PRG seed is s_0 or s_1, respectively.

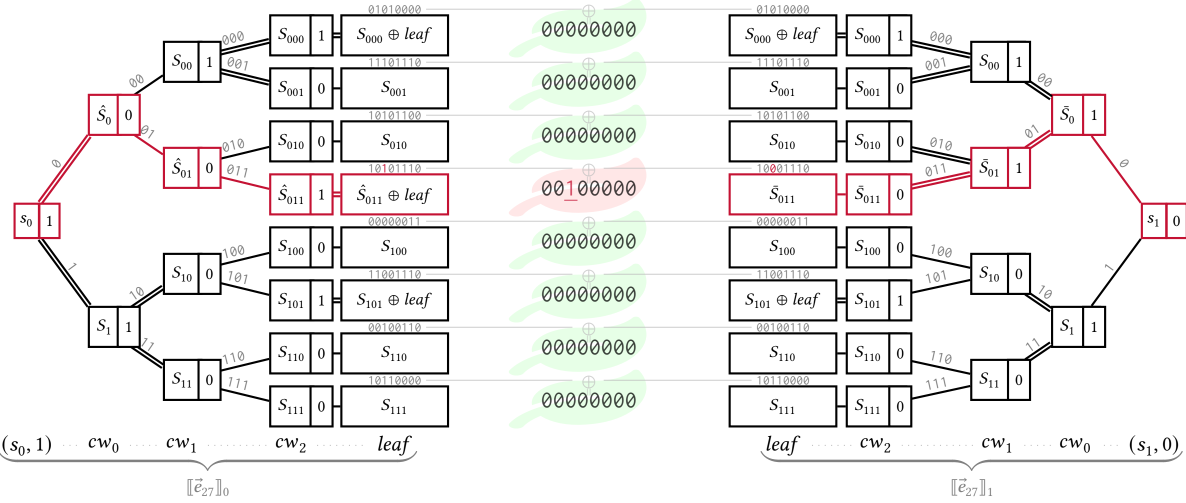


Figure 5: The BGI tree shares induced by $[\![\vec{e}_{27}]\!]$ over $\mathbb{Z}_{2^{3+3}}$.

Taxonomy of input pairs: Every input pair has one of four *types*; namely, setting $(L_0, R_0) \leftarrow \widetilde{G}_\lambda(s_0, advice_0, cw)$ and $(L_1, R_1) \leftarrow \widetilde{G}_\lambda(s_1, advice_1, cw)$, the pair is

1) a 0-*pair* if both $L_0 = L_1$ and $R_0 = R_1$;
2) an L-*pair* if $L_0 \neq L_1$ while $R_0 = R_1$;
3) an R-*pair* if $L_0 = L_1$ while $R_0 \neq R_1$; or
4) a 2-*pair* if neither $L_0 = L_1$ nor $R_0 = R_1$.

We will not use 2-pairs. When the L- versus R- "handedness" of a pair is irrelevant to the discussion (or is a secret), we refer to L-pairs and R-pairs alike as 1-*pairs*.

At a high level, the BGI construction uses pseudorandom traversal function families to construct concise, XOR-shared binary-tree representations of point functions. To see how this works, we first examine how to construct 0- and 1-pairs for a given $\widetilde{G}_\lambda$.

Constructing 0-pairs. Constructing a 0-pair is trivial: Choose $(s_0, advice_0, cw)$ arbitrarily, and then set $s_1 \coloneqq s_0$ and $advice_1 \coloneqq advice_0$. A straightforward calculation confirms that the resulting input pair is indeed a 0-pair.

Constructing 1-pairs. To construct a 1-pair, choose s_0, s_1, and $advice_0$ arbitrarily subject to $s_0 \neq s_1$, compute $(\bar{L}_0, \bar{R}_0) \leftarrow G_\lambda(s_0 \| 0)$ and $(\bar{L}_1, \bar{R}_1) \leftarrow G_\lambda(s_1 \| 0)$, set $advice_1 \coloneqq 1 \oplus advice_0$, $\bar{R} \coloneqq \bar{R}_0 \oplus \bar{R}_1$, and $\bar{L} \coloneqq \bar{L}_0 \oplus \bar{L}_1$, and then parse $\bar{L} = (\tilde{L}, \bar{t}_L) \in \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$ and $\bar{R} = (\tilde{R}, \bar{t}_R) \in \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$.

To make an L-pair, set

$$cw \coloneqq (\tilde{R}, 1 \oplus \bar{t}_L, \bar{t}_R); \tag{4}$$

to instead make an R-pair, set

$$cw \coloneqq (\tilde{L}, \bar{t}_L, 1 \oplus \bar{t}_R). \tag{5}$$

A slightly more involved, albeit fully mechanical, calculation confirms that in both cases the resulting input pair is indeed a 1-pair of the desired handedness.

Chaining 1-pairs. Notice that 0-pairs are agnostic to the values of cw and $advice_b$ (provided $advice_1 = advice_0$ holds), whereas 1-pairs require a very specific choice for cw (i.e., one that depends on s_0, s_1, and the desired handedness) and also that $advice_0 = 1 \oplus advice_1$. We recast part of this as a formal observation, to be used in the next section, as the second lynchpin of our construction.

Observation 4. *If* $((s_0, advice_0, cw), (s_1, advice_1, cw))$ *is a* b-*pair for* $b \in \{0, 1\}$, *then* $advice_0 \oplus advice_1 = b$.

Equations (4) and (5) together ensure the nonzero half of $\widetilde{G}_\lambda(s_0, advice_0, cw) \oplus \widetilde{G}_\lambda(s_1, advice_1, cw)$ has a 1 as its rightmost bit so that parsing that half of $\widetilde{G}_\lambda(s_0, advice_0, cw)$ and $\widetilde{G}_\lambda(s_1, advice_1, cw)$ each as $\mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$ elements yields a pair of values suitable for constructing another 1-pair. This enables the natural construction of 1-*pair chains* consisting of the initial $(s_b, advice_b)$ values linked by an array of cw terms.

BGI share generation. At a high level, the BGI construction assembles a chain of 1-pairs whose handedness reflect the leftmost bits of the distinguished input. Readers should heed the similarities between the construction of this chain and the definition of the 1-path in the corresponding point-function tree (cf. Theorem 3). Suppose the distinguished input is $i \in [0..N)$ with $N = 2^{n+k}$ and $\lambda = 2^k$. The chain begins with a uniformly random L-pair if the leftmost bit of i is 0 and a uniformly random R-pair if it is 1; the next link is an L-pair if the second-leftmost bit of i is 0 and an R-pair if it is 1, and so on until the chain accounts for each of the leftmost n bits of i.

For the remaining k bits, let $i' = i \bmod 2^k$ and let $\vec{e}_{i'}$ be the i'th selection vector of length λ. Suppose $(\bar{L}_0, \bar{R}_0)$ and $(\bar{L}_1, \bar{R}_1)$ are output by the last 1-pair in the chain. If that last pair is an L-pair, then output $leaf \coloneqq \bar{L}_0 \oplus \bar{L}_1 \oplus \vec{e}_{i'}$; otherwise, if it is an R-pair, then output $leaf \coloneqq \bar{R}_0 \oplus \bar{R}_1 \oplus \vec{e}_{i'}$.

Each DPF share $[\![\vec{e}_i]\!]_b$ then consists of (i) the bth share of the 1-pair chain (i.e., $(s_b, advice_b)$ and the array of n correction terms) alongside (ii) the final $leaf$ value.

From 1-pair chains to BGI tree shares. Owing to the pseudorandomness of G_λ, it is computationally infeasible for a shareholder knowing only one of $[\![\vec{e}_i]\!]_0$ or $[\![\vec{e}_i]\!]_1$ to deduce the sequence of L- versus R-ward traversals (i.e., the

leftmost bits of i) reflected in the 1-pair chain. Nevertheless, such a shareholder can "evaluate" the chain for *any* of the 2^n distinct length-n traversal sequences. Consequently, we can think of the 1-pair chain as implicitly defining a (componentwise-)XOR-shared *full binary tree* of height n, which we refer to as the *BGI tree induced by* $[\![\vec{e}_i]\!]$.

If ever the traversal sequence diverges from the binary representation of the distinguished input i, then, by definition, it transits (one share of) a 0-pair. Consequently, if both shareholders evaluate their respective shares on that same sequence, then *the pseudorandom values they produce must coincide from that* 0*-pair onward*—up to and including the leaf node. In particular, we have just argued that (i) corresponding leaves *not* at the end of the 1-pair chain hold XOR-shares of 0^λ; furthermore, due to the way the value *leaf* was constructed, (ii) corresponding leaves that *are* at the end of the 1-pair chain hold XOR-shares of $\vec{e}_{i'}$, with $i' = i \bmod \lambda$ capturing the k rightmost bits of the distinguished input i.

Observation 5. *Let* $(\langle\!|D|\!\rangle_0, \langle\!|D|\!\rangle_1)$ *be the BGI tree shares induced by* $[\![\vec{e}_i]\!]$*. If corresponding nodes in* $\langle\!|D|\!\rangle_0$ *and* $\langle\!|D|\!\rangle_1$ *form a* 0*-pair, then their reconstructed counterpart in* D *is a* 0*-node (or a* 0*-leaf); likewise, if they form a* 1*-pair, then their reconstructed counterpart in* D *is a* 1*-node (or a* 1*-leaf).*

The shareholders can, therefore, "evaluate" their respective DPF shares at any input $j \in [0..2^{n+k})$ to obtain an XOR-sharing $\langle\!|p_i(j)|\!\rangle$ by applying $\widetilde{\mathrm{G}}_\lambda$ repeatedly (with appropriate handedness) until arriving at the leaf, and then extracting the desired bit from that leaf.

Figure 5 illustrates the BGI tree shares induced by $[\![\vec{e}_{27}]\!]$ over $\mathbb{Z}_{2^{3+3}}$. The single bit set off in a box to the right of each node is that node's *advice* bit; edges are doublestruck (i.e., '‖') if the advice bit of the parent is 1 (a perturbation by *cw* is applied) and singlestruck (i.e., '|') otherwise (no perturbation by *cw* is applied). The 1-pair chain is set off in **red**. In between the two tree shares, we draw the reconstructed leaf nodes including the 1-byte substring of $\vec{e}_i$ that each leaf holds.

We conclude this section by asserting that the BGI construction constitutes a $(2, 2)$-DPF with 1-bit outputs (cf. Definition 5).

Theorem 5 ([5; Theorem 3.3])**.** *If* $\{\mathrm{G}_\lambda\}_{\lambda\in\mathbb{N}}$ *is a length-doubling PRG family, then the BGI construction is a* $(2, 2)$*-DPFs with* 1*-bit outputs. Each DPF share comprises just* $\lambda{\cdot}n + n - 1$ *bits.*

Interested readers can find proof of Theorem 5 and additional details about the BGI construction in Boyle et al.'s manuscript [5; §3.2.2].

6. DPFs as parity-segment trees

The following observation about segment parities over point functions is obvious, yet it is sufficiently central to our technique as to warrant explication.

Observation 6. *Fix* $i \in [0..N)$ *and let* $\vec{e}_i$ *be the* i*th selection vector of length* N*. Then* $\mathrm{parity}(\vec{e}_i[a..b)) = 1$ *if and only if* $i \in [a..b)$*.*

When combined with Theorem 4, Observation 6 leads to the following implication.

Corollary 1 (Point-function tree → parity-segment tree)**.** *The binary-tree representation of a point function doubles as the parity-segment tree for the associated selection vector, with* 0*-nodes implying even, and* 1*-nodes odd, parity.*

Meanwhile, from Observation 4 we know that the joint parity of *advice* bits for 0-pairs is even (0) while for 1-pairs it is odd (1). In conjunction with Observation 5, we get the following analogue of Corollary 1.

Corollary 2 (DPF tree → point-function tree)**.** *A DPF tree pair doubles as an XOR-shared binary-tree representation of the associated point function.*

Finally, we note that all arithmetic operations in the prefix-parity algorithm are linear over $\mathbb{Z}_2$. The confluence of this fact with a transitive application of Corollary 2 followed by Corollary 1 makes the following "Fundamental Theorem of Grotto" inescapable.

Theorem 6 (Fundamental Theorem of Grotto)**.** *BGI shareholders can run the prefix-parity algorithm directly on their respective DPF shares to obtain XOR-sharings of arbitrary segment parities, at a cost of one half-PRG*[4] *evaluation per edge traversed.*

Circling back to piecewise-polynomial function evaluation as discussed in Section 3, Theorem 6 leads to a novel approach for approximating (or exactly evaluating) a large class of nonlinear functions on additively-shared inputs at a *very* low cost relative to prior art.

7. The Grotto framework

We now have all the fundamental building blocks in place. This section describes how we integrated these building blocks to arrive at Grotto, our framework and C++ library for space- and time-efficient $(2 + 1)$-party evaluation of piecewise-polynomial functions (or splines) on $(2, 2)$-additively shared inputs.

The premise. P_0 and P_1 wish to obliviously evaluate some non-linear function $f : \mathbb{R} \to \mathbb{R}$ on input some $(2, 2)$-additively shared fixed-point value $[x]$; that is, they wish to compute $[f(x)]$ from $[x]$. We assume that f is well-approximated by the piecewise-polynomial function described in $\mathcal{F} := (\vec{B}, \vec{P})$ and that P_0 and P_1 each hold $\mathcal{F}$ in plaintext. Here $\vec{B}$ is the ordered list of endpoints for the "pieces" of the approximation and $\vec{P}$ is the correspondingly ordered list of (vectors of coefficients defining) polynomials for approximating within those pieces.

4. A *half-PRG* evaluation is an evaluation of G_λ in which only half of the output is required. For many PRGs, it is possible to compute only the required half at about half the cost of a full, length-doubling evaluation.

Preprocessing phase. In a preprocessing phase, some benevolent third-party (P_2) samples an ($[i]$, $[\![\vec{e}_i]\!]$) pair alongside "Beaver triple–like" values in support of the eventual sign-corrected polynomial evaluation (see Section 7.1), and then it distributes the shares and Beaver triple–like values to P_0 and P_1 and exits the scene.

Online phase. Upon learning $[x]$ in the online phase, P_0 and P_1 use a reconstructed $x - i$ to cyclically shift each endpoint in $\vec{B}$ to the left. Here they are effectively running the protocol of Figure 1, only with $[\![\vec{e}_i]\!]$ in place of $[\vec{e}_i]$ and leveraging the previously described equivalence between rotating $\vec{e}_i$ to the right versus rotating $\vec{B}$ to the left (see Figure 2). We emphasize that shifting $\vec{B}$ instead of $\vec{e}_i$ obviates the need for P_0 and P_1 to evaluate the DPF at every $i \in [0..N)$, a procedure that may be prohibitively costly—or perhaps even computationally infeasible—when N is large [33].

Both parties then run the prefix-parity algorithm on their respective shares of $[\![\vec{e}_i]\!]$ to find XOR-shared prefix parities for each of the above-rotated endpoints, and then they use these XOR-shared prefix parities to construct XOR-shares of the vector of segment parities corresponding to pieces in $\mathcal{F}$. Specifically, the resulting shares reconstruct to the selection vector indicating which polynomial reflected in $\vec{P}$ provides a good approximation to f on input x. From here, the two parties use this vector of parities to obliviously fetch *additive* shares of (plus-or-minus) the appropriate coefficients vector from $\vec{P}$, using the PIR-like process from Section 3.

Finally, the parties use the aforementioned Beaver triple-like values to compute sign-corrected evaluations on input $[x]$ of whatever polynomial f' they obliviously fetched in the preceding step, yielding additive shares of (a good approximation to) the desired evaluation $f(x)$.

7.1. Sign-corrected polynomial evaluation

Grotto could use any of a number of known techniques for oblivious polynomial evaluation. Our implementation supports two such techniques, namely either (i) Horner's method together with Du–Atallah multiplication or (ii) ABY2.0-style D-ary multiplication.

7.1.1. Horner's method. Horner's method is a technique for evaluating polynomials efficiently. To evaluate the degree-d polynomial $a(x) := a_d{\cdot}x^d + a_{d-1}{\cdot}x^{d-1} + \cdots + a_1{\cdot}x + a_0$, Horner's method simply expresses it in the form

$$a_0 + x{\cdot}\big(a_1 + x{\cdot}\big(a_2 + x{\cdot}\big(a_3 + \cdots + x{\cdot}(a_{d-1} + x{\cdot}a_d)\cdots\big)\big)\big),$$

thereby allowing its evaluation via an interleaved sequence of d additions and d multiplications.

For example, to evaluate the quadratic $f(x) = a_2{\cdot}x^2 + a_1{\cdot}x + a_0$, in which the coefficients a_i and indeterminate x are each fixed-point numbers with p fractional bits, on input $x = j$, Horner's method evaluates the expression

$$f(j) = \Big(\Big(\big(((a_2 \cdot j) \gg p) + a_1\big) \cdot j\Big) \gg p\Big) + a_0\,.$$

The arithmetic right-shifts following every multiplication ensure the operands to each addition and the final output all have p fractional bits.

When this approach is instantiated using Du–Atallah multiplication, evaluating a degree-d polynomial requires $2d + 1$ rounds of interaction—specifically, d rounds for the d multiplications interleaved with d rounds for precision reductions, plus one final round that merely sign-corrects the penultimate answer. As the details of the first $2d$ rounds are quite standard—and, in any case, not germane to our main contribution—we omit their detailed exposition, instead focusing our attention on the sign correction in the final round.

Sign-correcting the answer. Implementing the sign-correction is pleasantly easy: Let $(\!|\tilde{e}|\!) = ((\!|\tilde{e}|\!)_0, (\!|\tilde{e}|\!)_1)$ be the XOR-shared vector of segment parities, let U_0 and U_1 respectively denote the sum over $\mathbb{Z}_N$ of all parities in $(\!|\tilde{e}|\!)_0$ and $(\!|\tilde{e}|\!)_1$, and set $u := U_0 - U_1$. By construction, we have $u = \pm 1$ and, moreover, if f' is the (uncorrected) polynomial that $(\!|\tilde{e}|\!)$ selects from $\vec{P}$, then u has the "matching" sign. It, therefore, follows that

$$f(j) \approx u{\cdot}f'(j),$$

which P_0 and P_1 can easily compute a sharing of using one final Du–Atallah multiplication between the sharings $[\pm f'(j)]$ and $[u] = (U_0, -U_1)$.

7.1.2. ABY2.0-style multiplication. The approach based on Horner's method incurs low precomputation costs in exchange for a relatively large (degree-dependent) round complexity. As an alternative that avoids this blowup in round complexity, Grotto also supports polynomial evaluation based on the single-round D-ary multiplication technique used by ABY2.0 [25]. This technique has a noticeably higher pre-processing cost, but it can be substantially more performant in instances where each round of communication incurs Internet round-trip latency.

Our use of ABY2.0-style multiplication follows the original expositions of Patra, Schneider, Suresh, and Yalame [25; §3.1.4] rather faithfully, save for two important optimizations that we introduce specifically to facilitate efficient fixed-point polynomial evaluation. First, because we desire only the final output of a polynomial evaluation (i.e., we are not interested in evaluating the individual monomials), we are able to merge several P_2 terms to noticeably reduce overall precomputation size relative to a naïve application of ABY2.0-style multiplication to the individual monomials. Second, to prevent integer overflows and the need to reduce the fractional bits in intermediate values, we "lift" the coefficients a_i and indeterminate x to a larger ring (namely, to $\mathbb{Z}_{2^{n+k+m}}$ for some $m \in \mathbb{N}$). Once the polynomial has been evaluated in this larger ring, we project the result back into $\mathbb{Z}_{2^{n+k}}$. While it would be possible to support arbitrarily large integer parts with this approach, our implementation in Grotto seeks only to support evaluations that would also succeed using Horner's method (see Section 7.1.1); for instance, if x is a 64-bit integer with 16 fractional bits, then each intermediate value

in the Horner evaluation has 32 fractional bits and, thus, integer parts comprising at most 31 = 64 − 32 − 1 bits. Therefore, for polynomials of degree d, we desire a ring large enough to encode $16(d + 3)$ bits—31 integer bits, 1 sign bit, and $16(d + 1)$ fractional bits—such as $\mathbb{Z}_{2^{n+k+m}}$ for any $m \geq \max(16(d+3)-n-k, 0)$. For optimal performance on 64-bit CPUs, our implementation uses the smallest such m for which $n + k + m$ is a multiple of 64.

Lifting the coefficients. Lifting the a_i is trivial: Since P_0 and P_1 hold the LUT $\vec{P}$ in plaintext, they can lift each coefficient "for free" ahead of the PIR step. At the same time, they adjust the number of fractional bits in each coefficient (by left-shifting in zeros) so that every monomial $a_i{\cdot}x^i$ will use exactly $(d + 1){\cdot}p$ fractional bits; that is, they replace each a_i by $\bar{a}_i := (a_i \ll (d-i){\cdot}p)\|0^{i\cdot p} \in \mathbb{Z}_{2^{n+k+m}}$. The latter fractional-precision adjustments ensure that the decimal points in intermediate values of the polynomial evaluation "line up", allowing them to be added or subtracted non-interactively.

Lifting the indeterminate. Lifting the indeterminate is more cumbersome, as P_0 and P_1 hold only a sharing $[j]$ of the input. The main observation behind our approach is that lifting $[j]$ is actually trivial—*provided we are not fussy about the number fractional bits in the lifted result.* Specifically, to lift $[j]$ from $\mathbb{Z}_{2^{n+k}}$ into $\mathbb{Z}_{2^{n+k+m}}$, P_0 and P_1 each simply append 0^m to the binary representations of their respective shares (and switch from reducing modulo 2^{n+k} to reducing modulo 2^{n+k+m}) so that

$$[j]_0\|0^m + [j]_1\|0^m = j\|0^m \in \mathbb{Z}_{2^{n+k+m}},$$

which is the correct j, only with $p + m$ instead of p fractional bits. From here, an interactive fractional precision reduction (see Section 2.4) suffices to "reset" the number of fractional bits back to the desired p.

Only one question remains unanswered: How do P_0 and P_1 determine $[\mathsf{msb}(j)]$, which is needed in Equation (1), from $[j]$? For this, we look inward, noting that the msb function is just a piecewise-constant (indeed, piecewise-Boolean) function comprising two parts; thus, the parties can calculate $[\pm\mathsf{msb}(j)]$ using *the same DPF shares* as they are using to fetch f' from $\vec{P}$. Because the "polynomials" for the msb function are constant integers, there is no reason to lift them to the larger ring (i.e., there is no "chicken and egg" situation). Then, when subsequently computing the correction term via Equation (1), we use the fact that $\mathsf{msb}(j) \in \{0, 1\}$ so that

$$\big(\pm\mathsf{msb}(j)\big)^2 = \big(-\mathsf{msb}(j)\big)^2 = \mathsf{msb}(j)^2 = \mathsf{msb}(j)$$

holds for all j.

Putting it all together. Instantiating Grotto with ABY2.0-style multiplication yields a three-round protocol for approximations via polynomials of any degree:

Round 1: P_0 and P_1 reconstruct $j - i \bmod 2^{n+k}$ and use a Du–Atallah multiplication to compute the sharing $[Z_0{\cdot}Z_1]$ with $Z_b := \mathsf{msb}([j]_b)$ for $b = 0, 1$ (cf. Equation (1)). Note that P_0 and P_1 respectively know Z_0 and Z_1 in plaintext.

Before proceeding to the next round, each party lifts (and precision-adjusts) the coefficients comprising $\vec{P}$ into $\mathbb{Z}_{2^{n+k+m}}$ and then runs the prefix-parity algorithm to fetch its shares of $[\pm\bar{a}_i]$ and $[\pm Z]$ for $Z := \mathsf{msb}(j)$.

Round 2: P_0 and P_1 use a ternary ABY2.0-style multiplication to compute the sharing $[(Z_0{\cdot}Z_1){\cdot}(\pm Z)^2]$ from $[Z_0{\cdot}Z_1]$ and $[\pm Z]$, and then they use it to (from this point on, non-interactively) compute $[\pm 2^{n+k}]$ using Equation (1).

Before proceeding to the next round, the parties use $[\pm 2^{n+k}]$ as the correction term to lift the shares of the indeterminate $x \in \mathbb{Z}_{2^{n+k}}$ into shares of $\bar{x} \in \mathbb{Z}_{2^{n+k+m}}$.

Round 3: P_0 and P_1 use a $(d+2)$-ary ABY2.0-style multiplication over sharings $[\bar{a}_0], \ldots, [\bar{a}_d]$, $[\bar{x}]$, and $[u]$ (which they compute the same way as they would in Horner's method) to evaluate $[f'(x)] = [u{\cdot}(a_d{\cdot}x^d + \cdots + a_1{\cdot}x + a_0)]$.

In cases where the coefficients and x need not be lifted to avoid overflow, we can skip **Round 2** above, resulting in a somewhat simpler two-round protocol. We include our precise formulae for ABY2.0-style sign-corrected polynomial evaluation and detailed derivations thereof as Appendix E.[5]

8. Implementation & evaluation

To empirically evaluate the performance of our approach, we implemented Grotto as a C++ library. Our implementation uses dpf++ [15] for (2, 2)-DPFs, the GNU multiprecision arithmetic library (GMP) v6.2.1 [13] for multi-limb arithmetic in our ABY2.0-style multiplication, and the C++ version of ALGLIB 3.19.0 [2] for curve fitting in our LUT-generation code.

In addition to implementing the prefix-parity algorithm and associated (2 + 1)-party protocols, our implementation comes equipped with *scores* of "gadgets" (i.e., LUTs and associated machinery) for evaluating common functions, including trigonometric and hyperbolic functions (and their inverses); various logarithms; roots, reciprocals, and reciprocal roots; sign testing and bit counting; and over two dozen of the most common (univariate) activation functions from the deep-learning literature. We also include utilities for generating additional LUTs from arbitrary functions $f\colon \mathbb{R} \to \mathbb{R}$ given as a blackbox.[6] We include as Appendix D a table summarizing selected gadgets (65 in all) supported out-of-the-box by Grotto, including efficiency metrics (polynomial degree, number of parts in $\vec{P}$, and

5. In fact, Appendix E includes derivations for both one- and two-round sign-corrected evaluations (for constant, linear, quadratic, and cubic polynomials). The two-round variants are similar to their one-round counterparts, as described above, except they apply the sign correction as a post-processing step (similar to with Horner's method). This reduces the number of Beaver-like terms that P_2 must send at the expense of one additional communication round; crucially, though, it still decouples the round complexity of polynomial evaluation from the degree of the polynomial under consideration.

6. Simultaneously efficient, accurate, and fully-automated LUT generation is impossible given only blackbox access to f; to work around this, our LUT-generation utility allows the user to provide some "hints" that effectively transform the problem into that of "graybox" LUT generation.

TABLE 2: GROTTO versus LLAMA and EzPC DCFs

Function	Scheme	Input		Polys		Preprocessing		Online		
		bitlength	frac bits	# parts	degree	comp	comm	comp	comm	rounds
isqrt	LLAMA	16	9	10	2	28 ± 4 μs	11.37KiB	60 ± 10 μs	36B	3
	GROTTO	16	9	40	3	2.65 ± 0.02μs	0.38KiB	3.1 ± 0.6μs	74B	3
	GROTTO	64	16	330	3	4.61 ± 0.02μs	1.32KiB	78 ± 1 μs	152B	3
tanh	LLAMA	16	9	12	2	31 ± 6 μs	13.22KiB	60 ± 10 μs	36B	3
	GROTTO	16	9	19	3	2.67 ± 0.07μs	0.38KiB	2.6 ± 0.5μs	74B	3
	GROTTO	64	16	84	3	4.55 ± 0.07μs	1.32KiB	17 ± 3 μs	152B	3
sigmoid	LLAMA	16	9	34	2	60 ± 10 μs	33.05KiB	260 ± 60 μs	36B	3
	GROTTO	16	9	84	3	2.66 ± 0.02μs	0.38KiB	5 ± 7 μs	74B	3
	GROTTO	64	16	98	3	4.61 ± 0.02μs	1.32KiB	21 ± 4 μs	152B	3
log10	GROTTO	16	9	40	3	2.66 ± 0.05μs	0.38KiB	3.2 ± 0.6μs	74B	3
	GROTTO	64	16	550	3	4.58 ± 0.03μs	1.32KiB	186 ± 2 μs	152B	3
sqrt	GROTTO	16	9	39	3	2.66 ± 0.02μs	0.38KiB	6 ± 1 μs	74B	3
	GROTTO	64	16	999	3	4.50 ± 0.04μs	1.32KiB	554 ± 1 μs	152B	3

expected number of half-PRG invocations used by the prefix-parity algorithm) alongside fidelity metrics (maximum error and root-mean-squared approximation error) for each gadget.

Performance benchmarks. The remainder of this section reports our findings from a series of experiments we ran on a workstation equipped with 16GiB of RAM and an Intel Core i7-9700K processor, and running Ubuntu 18.04. We ran all experiments for 100 trials and report here the sample mean alongside the sample standard deviation over those 100 trials. (We express this as *mean* ± *stdev.*) All numbers are reported to one significant figure in the sample standard deviation.

We compare GROTTO against the recent work closest to it, namely LLAMA [14]. (Some details about LLAMA and its relation to GROTTO appear in Section 9.) Table 2 presents single-threaded wall-clock running times *excluding network latency* for a small selection of gadgets. The first three gadgets—*reciprocal square root* (isqrt), *hyperbolic tangent* (tanh), and the *logistic sigmoid* function (sigmoid)—are also provided by the reference implementation of LLAMA, albeit only with 16-bit words. In addition to the three above-mentioned functions, we also benchmark GROTTO on the *square root* (sqrt) and *base-10 logarithm* (log10) functions. We note that sqrt is the costliest function among those listed in Appendix D in terms of expected half-PRG calls.

Our experiments indicate that GROTTO handily outperforms LLAMA on almost every metric: In all instances, its preprocessing time and space requirements are but a fraction of what LLAMA uses; online computation times are consistently lower as well, with even GROTTO on 64-bit words outperforming even LLAMA on 16-bit words for two out of three common gadgets.

In terms of round complexity and online communication, GROTTO is also competitive with LLAMA but cannot compete with DCFs' non-interactive evaluation. In particular, GROTTO incurs the same round complexity as LLAMA, but has a noticeably higher online communication cost, owing to our use of cubic (rather than quadratic) polynomials and our strategy of lifting shares to a larger ring. The upshot of this higher online communication is the capacity for faster and more accurate approximations: GROTTO's approximations for all three functions exhibit a maximum error less than the fixed-point numbers can represent, whereas LLAMA tolerates deviations of up to 4 units in the last place (ULPs) of error.

We stress that all of the times we present explicitly ignore the network communication time. This is done for consistency with the available implementation of LLAMA and to remove the communication overhead which dominates the running time of both LLAMA and GROTTO alike. Besides, typical Internet latency is easily four to five orders of magnitude higher than GROTTO's running time, making the overhead of GROTTO impossible to separate from the variance in the network latency.

9. Related work

GROTTO builds on a line of recent works that use DPFs and the related *distributed comparison functions (DCFs)*, a DPF-adjacent primitive that is specialized for performing efficient comparisons, to evaluate non-linear functions in additive secret sharing–based (2+1)-party protocols.

For instance, Vadapalli, Bayatbabolghani, and Henry [31] used the *spline evaluation via selection vectors* approach as described in Section 3, but with the selection vector shares compressed as DPF shares, to implement both piecewise-linear approximations for the *reciprocal square root* (isqrt) function and a piecewise-constant exact *comparison* (leq) for 16-bit fixed-point numbers in the semi-honest $(2+1)$-party setting. Subsequently, Wagh [33] generalized their approach to work in the fully malicious setting by porting a sketching protocol for finite fields into the ring of integers modulo 2^n. In both cases, the authors restricted attention to $\mathbb{Z}_N$ for N a *small* power of 2 to account for inherently poor ($O(N)$) asymptotics; indeed, Wagh writes that "typical sizes for which [this approach] provides performance comparable to general purpose (sic) MPC are around 20–25 bits" [33; §3].

Boyle, Chandran, Gilboa, Gupta, Ishai, Kumar, and Rathee [3] showed how replacing DPFs with DCFs can lead to vastly improved asymptotics. Indeed, their innovation improves exponentially on the complexity of the Vadapalli

et al. and Wagh, with communication and computation both in $O(P\cdot \lg N)$ to evaluate a spline with P parts. Gupta, Kumaraswamy, Chandran, and Gupta [14] built on Boyle et al.'s ideas in constructing their LLAMA library, a project with similar goals to Grotto. Compared with the methods described in the above two works, Grotto reduces communication cost from $O(P\cdot \lg N)$ to $O(\lg N)$ and computation cost from $O(P\cdot \lg N)$ to $o(P\cdot \lg N)$, while also shrinking the hidden constants.

In terms of applications, one of the most obvious places our work applies is evaluating non-linear activation functions in the context of privacy-preserving neural networks. Recent years have seen a flurry of research in this direction, including a plethora of works in the 2-party [17], [18], [21], [21], [27], $(2+1)$-party [19], [28], 3-party [8], [24], [33]–[35], and 4-party [7], [9], [20] settings.

10. Conclusion

We introduced Grotto, a framework and C++ library for space- and time-efficient $(2+1)$-party piecewise polynomial evaluation on secrets additively shared over $\mathbb{Z}_{2^{n+k}}$. Grotto improves on the state-of-the-art approaches based on DCFs in almost every metric, offering asymptotically superior communication and computation costs with comparable round complexity. At the heart of Grotto is a novel observation about the structure of the most compact DPFs from the literature and our prefix-parity algorithm, which leverages this structure to do with a single DPF what others require many DCFs to do.

Acknowledgements. We acknowledge the support of Cisco Research and the Natural Sciences and Engineering Research Council of Canada (NSERC) through grant RGPIN 2019-04821.

References

[1] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology: Proceedings of CRYPTO 1991*, volume 576 of *LNCS*, pages 420–432, Santa Barbara, CA, USA (August 1991).

[2] Sergey Bochkanov and the ALGLIB Project development team. ALGLIB; version 3.19.0 [computer software]. Available from: https://www.alglib.net/, June 2022.

[3] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Advances in Cryptology: Proceedings of EUROCRYPT 2021 (Part II)*, volume 12697 of *LNCS*, pages 871–900, Zagreb, Croatia (October 2021).

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Advances in Cryptology: Proceedings of EUROCRYPT 2015 (Part II)*, volume 9057 of *LNCS*, pages 337–367, Sofia, Bulgaria (April 2015).

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In *Advances in Cryptology: Proceedings of CRYPTO 2016 (Part I)*, volume 9814 of *LNCS*, pages 509–539, Santa Barbara, CA, USA (August 2016).

[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of CCS 2016*, pages 1292–1303, Vienna, Austria (October 2016).

[7] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2020(2), April 2020.

[8] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: High throughput 3PC over rings with application to secure prediction. In *Proceedings of CCSW@CCS 2019*, pages 81–92, London, UK (November 2019).

[9] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *Proceedings of NDSS 2020*, San Diego, CA, USA (February 2020).

[10] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of FOCS 1995*, pages 41–50, Milwaukee, WI, USA (October 1995).

[11] Wenliang Du and Mikhail J. Atallah. Protocols for secure remote database access with approximate matching. In *E-Commerce Security and Privacy (Part II)*, volume 2 of *Advances in Information Security*, pages 87–111, February 2001.

[12] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology: Proceedings of EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658, Copenhagen, Denmark (May 2014).

[13] Torbjörn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library; version 4.6.1 [computer software]. Available from: https://gmplib.org/, November 2020.

[14] Kanav Gupta, Deepak Kumaraswamy, Divya Gupta, and Nishanth Chandran. LLAMA: A low latency math library for secure inference. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2022(4):274–294, October 2022.

[15] Ryan Henry and Adithya Vadapalli. dpf++; version 0.0.1 [computer software]. Available from: https://www.github.com/rh3nry/dpfplusplus, July 2019.

[16] William George Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, January 1819.

[17] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. In *Proceedings of USENIX Security 2022*, pages 809–826, Boston, MA, USA (August 2022).

[18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of USENIX Security 2018*, pages 1651–1669, Baltimore, MD, USA (August 2018).

[19] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of CCS 2020*, pages 1575–1590, Virtual event (October 2020).

[20] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In *Proceedings of USENIX Security 2021*, pages 2651–2668, Virtual event (August 2021).

[21] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *Proceedings of USENIX Security 2020*, pages 2505–2522, Virtual event (August 2020).

[22] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *Proceedings of IEEE S&P 2017*, pages 19–38, San Jose, CA, USA (May 2017).

[23] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology: Proceedings of EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 223–238, Prague, Czech Republic (May 1999).

[24] Arpita Patra and Ajith Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *Proceedings of NDSS 2020*, San Diego, CA, USA (February 2020).

[25] Arpite Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. In *Proceedings of USENIX Security 2021*, pages 2165–2182, Virtual event (August 2021).

[26] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, Cambridge, MA, USA, May 1981.

[27] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SIRNN: A math library for secure RNN inference. In *Proceedings of IEEE S&P 2021*, pages 1003–1020, San Francisco, CA, USA (May 2021).

[28] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis R. Bach. AriaNN: Low-interaction privacy-preserving deep learning via function secret sharing. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2022(1):291–316, January 2022.

[29] Adi Shamir. How to share a secret. *Communications of the ACM (CACM)*, 22(11):612–613, November 1979.

[30] Neil J. A. Sloane and OEIS Foundation Inc. A083652: Sum of lengths of binary expansions of 0 through n. In *The on-line encyclopedia of integer sequences.* [Online; accessed 2022-10-13].

[31] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. You may also like... Privacy: Recommendation systems meet PIR. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2021(4):30–53, October 2021.

[32] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-anonymous messaging with fast audits. In *Proceedings of IEEE S&P 2022*, pages 1953–1970, San Francisco, CA, USA (May 2022).

[33] Sameer Wagh. Pika: Secure computation using function secret sharing over rings. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2022(4):351–377, October 2022.

[34] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2019(3):26–49, March 2019.

[35] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2021(1):188–208, 2021.

Appendix A. Proof of Theorem 1

Theorem 1 (Restatement). *Given a parity-segment tree $T(x)$ of height n and a lexicographically sorted list E of S distinct prefix endpoints, the prefix-parity algorithm traverses at most $Sn - \sum_{i=2}^{S}\lfloor \lg(i-1)\rfloor$ edges to compute all S prefix parities.*

Before proving Theorem 1, we state and prove the following lemma based on the intuition that a “worst-case” instance for the prefix-parity algorithm is one that minimizes the lengths of common prefixes in (the binary representations of) successive endpoints in E.

Lemma 1. *If P is a set of S distinct bitstrings, then*

$$\sum_{x\in P}|x| \ge \sum_{i=1}^{S-1}\lceil \lg(i+1)\rceil, \tag{6}$$

where $|x|$ denotes the length (in bits) of x.

Proof (Sketch). We first note that equality holds in Equation (6) when P consists of the empty string together with the binary representations of all non-negative integers less than $S-1$ [30]. Moreover, substituting one or more bitstrings from P with the binary representations of integers *greater than or equal to* $S-1$ results in a set whose aggregate length is likewise greater than or equal to that of P. ◻

Proof of Theorem 1. The result follows by induction on S.

Base case ($S = 1$): This case is immediate by inspection.

Inductive step: Assume the prefix-parity algorithm traverses at least $(S-1)\,n - \sum_{i=2}^{S-1}\lfloor \lg(i-1)\rfloor$ edges for $S-1$ endpoints and let E be some length-$(S-1)$ sequence of endpoints inducing this worst-case cost. We will construct a worst-case length-S sequence E' by inserting one additional endpoint at an appropriate (sorted) position within E. Specifically, to ensure that the resulting sequence is also a worst-case sequence, it suffices to choose a position among existing endpoints that (globally) minimizes the length of its common prefix with its immediate neighbours in the resulting ordered sequence (thereby maximizing the number of new edges to traverse).
Thus, we construct E' by inserting an arbitrary endpoint whose prefix is one of the (not necessarily unique) shortest prefixes not yet reflected in E. Now, consider the sets P and P' of shortest unique prefixes for E and E', respectively. From Lemma 1, we have

$$\begin{aligned}\sum_{x\in P'}|x| - \sum_{x\in P}|x| &\ge \sum_{i=1}^{S-1}\lceil \lg(i+1)\rceil - \sum_{i=1}^{S-2}\lceil \lg(i+1)\rceil\\ &= \lceil \lg(S-1+1)\rceil\\ &\ge \lfloor \lg S\rfloor .\end{aligned}$$

As all but the last bit of the shortest unique prefix corresponds to an already-traversed edge, this newly added endpoint necessitates traversing at most

$$n - (\lceil \lg S\rceil - 1) \ge n - \lfloor \lg(S-1)\rfloor$$

additional edges, for a total number of edges traversed of at most

$$\begin{aligned}\Big((S-1)\,n - \sum_{i=2}^{S-1}\lfloor \lg(i-1)\rfloor\Big) + \big(n - \lfloor \lg(S-1)\rfloor\big)\\ = Sn - \sum_{i=2}^{S}\lfloor \lg(i-1)\rfloor .\end{aligned}$$ ◻

Appendix B.
Prefix-parity pseudocode

This appendix presents a pseudocode listing for the prefix-parity algorithm (Figure 6). The pseudocode is written using a C-like syntax and is accompanied by some commentary (on the right) that elucidates selected steps in the algorithm (notably including those related to memoization).

```
// Computes the prefix parities for each of the given endpoints
//
// Parameters:
//   bound - a sorted list of segment endpoints
//   T     - a parity-segment tree
//   n,k   - depth of tree, lg(bits per leaf)
//
vector prefix_parities(vector bound, tree T, int n, int k)
{
  vector res          = {}          // to be populated with prefix parities
  vector path         = {T.root}    // memoized current path from the root
  vector direction    = {0}         // traversal directions along path
                                    // 0 is to the left; 1 is to the right
  vector parity       = {0}         // cumulative prefix parity along path
  vector left         = {2**(n-1)}  // leftmost-beneath-right-child along path
  int prev            = ~bound[0]   // *complement* of first shifted bound;
                                    // ensures from=0 in first loop iteration

  for (int i = 0; i < bound.length; i++)
  {
    int from          = clz(bound[i] ^ prev) // common prefix length
    int to            = n           // by default, traverse entire depth

    for (int j = from; j < to; j++) // iterate only over (non-common) suffix
    {
      int next_dir    = (left[j] <= bound[i]) ? 1 : 0 // going right or left?
      int next_path   = T.traverse(path[j], next_dir) // go that direction
      int next_parity = (next_dir == direction[j])    // update running parity
                        ? parity[j]                     // no change
                        : path[j].parity ^ parity[j]    // include/exclude
      int next_left   = (next_dir == 1)               // update leftmost bit
                        ? left[j] + 2**(64-2-j)         // advance right
                        : left[j] - 2**(64-2-j)         // unadvance left
                                    //^^^^^^^^^^^ <- #bits beneath each child

      path[j+1]       = next_path   // memoize node along path
      direction[j+1]  = next_dir    // memoize direction of traversal
      left[j+1]       = next_left   // memoize leftmost-beneath-right-child
      parity[j+1]     = next_parity // memoize cumulative parities

      if (next_left == bound[i])    // early-termination optimization
        to = j                      //   -> halt traversal at current level
    } // inner for loop ends

    res[i] = parity[to]             // record running prefix-parity
    if (to == n)                    // conditionally add partial-leaf parity
    {
      string substr = path[n].substr   // substring associated with leaf
      int prefix_len = bound[i] % 2**k // length of prefix to compute
      res[i] ^= prefix_parity_str(substr, prefix_len)
    }
    prev = bound[i]                 // for computing common prefix length
  } // outer for loop ends
  return res                        // n.b.: *prefix* (not segment) parities!
} // prefix_parities
```

`vector path` memoizes each node along the path from the root to the leaf node whose substring contains the `bound[i]`th bit, sidestepping the need to revisit any node in the common prefix between `bound[i]` and `bound[i+1]` (and, likewise, between `bound[i-1]` and `bound[i]`). If `vector bound` is lexicographically sorted, then this is sufficient to ensure that no edge in the tree is traversed more than once.

Similarly, `vector parity` memoizes running parities along this path. Since the “inclusion–exclusion” decisions used to update running parities depend on traversal direction *changes*, we also use `vector direction` to memoize traversal directions along this path. Meanwhile, `vector left` exists to assist in deciding which direction to go.

`int from` is the index of the first bit *after* the common prefix; i.e., the point starting from which memoized values reflect `prev` but not `bound[i]`. In the first loop iteration, `prev == ~bound[i]`, which ensures that `from=0`. On the subsequent iterations, `from` is set to `clz(bound[i] ^ prev)`, the number of leading zeros in the binary representation of `bound[i] ^ prev` (i.e., the number of prefix bits common to the current and previous bound). `int to=n` at the start of each loop iteration, but it may be reduced by the early-terminate optimization on Lines 41–42.

`int next_dir` indicates whether to traverse to the right (`1`) or left (`0`), depending on whether `parity[j]` currently over- or undershoots `path[j]`. After traversing in that direction on Line 27, Lines 28–33 compute the `next_parity` and the `next_left`. The ternary operator on lines 28–30 carries forward `parity[j]` if there was no change in traversal direction; otherwise, it first “updates” that parity by XORing in the parity of the node just traversed. All four values are memoized on Lines 36–39.

If, by serendipity, `path[j+1]` neither overshoots nor undershoots `bound[i]` at this point, then `parity[j+1]` already equals the desired parity and we can break from the inner loop now—even if we haven’t yet reached a leaf. We break by setting `to=j` so that Lines 45–51 can easily determine whether or not we manually broke from the inner loop.

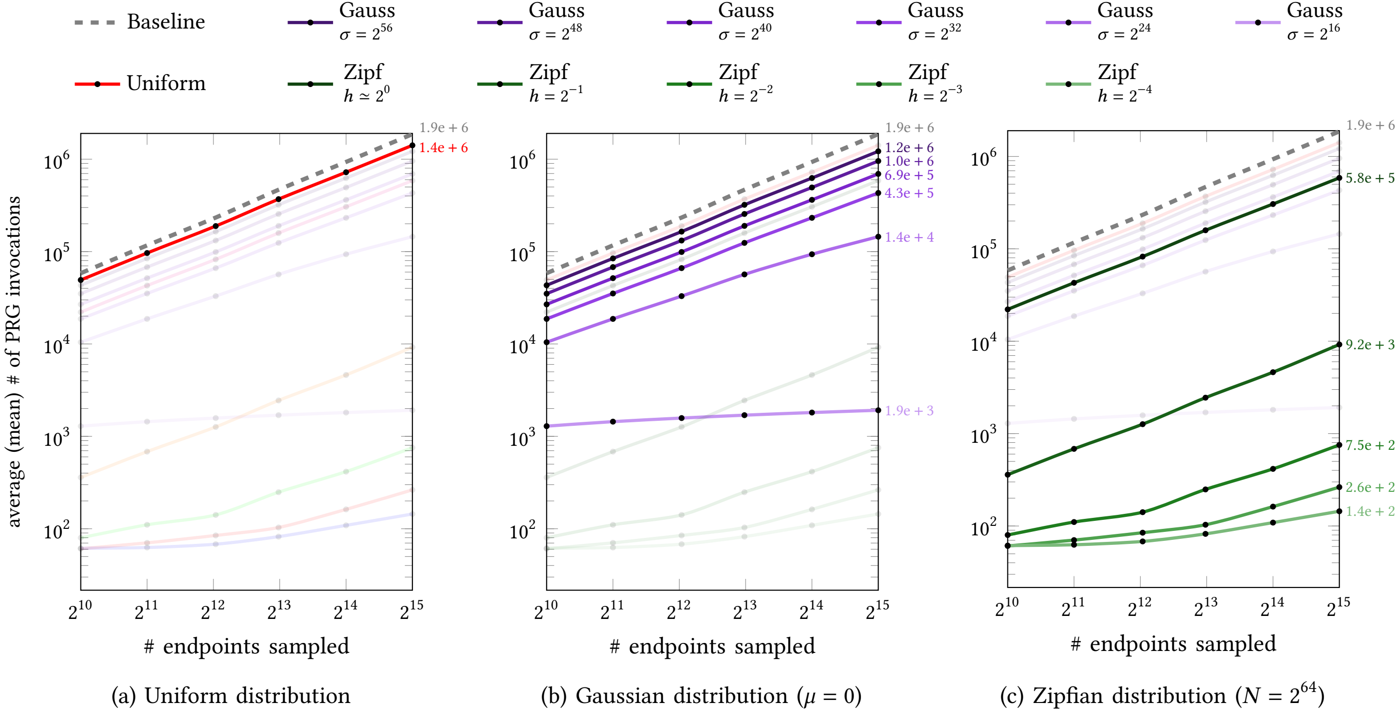
If we did *not* manually break from the inner loop, then we must complete the parity computation by XORing in some prefix of the substring in the leaf node currently stored in `path[n]`.

Figure 6: C-like pseudocode listing for the prefix-parity algorithm (left) with running commentary (right).

Appendix C.
Prefix-parity amortization via memoization

This appendix presents empirically measured total costs for the prefix-parity algorithm with many endpoints, such as when evaluating several complex functions at once. The costs depend heavily on the distribution of the points, with higher endpoint density implying greater per-endpoint savings. We choose three kinds of distributions, namely uniform, Gaussian, and Zipfian.

As expected, the greater the variance of the distribution, the lower the amortization savings that the prefix-parity algorithm enjoys, with the “worst” case occurring when endpoints are sampled uniformly. By contrast, the amortization savings for Zipfian distributions with small parameters are very extreme.



(a) Uniform distribution

(b) Gaussian distribution ($\mu = 0$)

(c) Zipfian distribution ($N = 2^{64}$)

Figure 7: Empirical amortized costs for prefix-parity over points sampled from selected probability distributions.

Appendix D.
Selected gadgets

This appendix lists selected gadgets that are supported by GROTTO out of the box, summarizing polynomial degrees and lookup table sizes for each gadget assuming 64-bit fixed-point arithmetic with 16 fractional bits.

	Gadget	Descriptive name	Formula	Degree	# parts[†]	Max error[†]	RMSE[†]	Expected half-PRGs[†]
trigonometric functions	cos	cosine[¶]	$\cos(x)$	3	64	3.0e − 8	1.6e − 8	586 ± 3
	sin	sine[¶]	$\sin(x)$	3	63	3.0e − 8	1.7e − 8	580 ± 2
	tan	tangent[¶]	$\sin(x)/\cos(x)$	3	608	4.6e − 4[‡]	5.8e − 5[‡]	489 ± 3
	csc	cosecant[¶]	$1/\sin(x)$	3	595	4.6e − 4[‡]	5.8e − 5[‡]	492 ± 3
	sec	secant[¶]	$1/\cos(x)$	3	358	4.6e − 4[‡]	5.5e − 5[‡]	496 ± 3
	cot	cotangent[¶]	$1/\tan(x)$	3	366	2.9e − 4[‡]	3.4e − 5[‡]	486 ± 3
	arccos	arc cosine	$\cos^{-1}(x)$	3	31	6.8e − 5	1.1e − 5	245 ± 2
	arcsin	arc sine	$\sin^{-1}(x)$	3	31	7.9e − 5	1.3e − 5	244 ± 2
	arctan	arc tangent	$\tan^{-1}(x)$	3	179	1.6e − 7	2.4e − 8	2333 ± 4
	arccsc	arc cosecant	$\csc^{-1}(x)$	3	55	4.4e − 4[‡]	3.9e − 5[‡]	744 ± 3
	arcsec	arc secant	$\sec^{-1}(x)$	3	58	4.4e − 4[‡]	3.5e − 5[‡]	755 ± 4
	arccot	arc cotangent	$\cot^{-1}(x)$	3	41	1.5e − 5[‡]	7.2e − 6[‡]	684 ± 3
	arcosh	area hyperbolic cosine	$\ln(x+\sqrt{x^2-1})$	3	439	2.4e − 7	8.7e − 8	11 129 ± 4
	arsinh	area hyperbolic sine	$\ln(x+\sqrt{x^2+1})$	3	753	2.4e − 7	8.9e − 8	22 166 ± 4
	artanh	area hyperbolic tangent	$\frac{1}{2}\ln((1+x)/(1-x))$	3	54	1.5e − 7	9.0e − 8	299 ± 2
logarithms	ln	natural logarithm	$\ln(x)$	3	507	3.0e − 7	7.5e − 8	11 382 ± 5
	log10	common logarithm	$\log_{10}(x)$	3	550	1.7e − 7	4.8e − 8	10 343 ± 5
	log2	binary logarithm	$\lg(x)$	3	193	3.9e − 5	6.5e − 6	4402 ± 5
	ilogb	integer binary log	$\lfloor \lg(x) \rfloor$	0	128	0	0	3306 ± 1
	ilog10	integer base-10 log	$\lfloor \log_{10}(x) \rfloor$	0	40	0	0	1158 ± 5
roots/reciprocals	sqrt	square root	$\sqrt{x}$	3	999	1.2e − 4	2.6e − 5	34 911 ± 4
	cbrt	cube root	$\sqrt[3]{x}$	3	765	1.7e − 5	2.1e − 6	24 326 ± 4
	qtrt	quartic root	$\sqrt[4]{x}$	3	379	1.6e − 5	6.3e − 6	9344 ± 5
	isqrt	reciprocal square root	$1/\sqrt{x}$	3	330	3.5e − 6	1.6e − 7	4174 ± 3
	icbrt	reciprocal cube root	$1/\sqrt[3]{x}$	3	192	2.0e − 5	3.7e − 6	1082 ± 4
	iqtrt	reciprocal quartic root	$1/\sqrt[4]{x}$	3	367	3.5e − 6	1.6e − 7	5862 ± 4
	reciprocal	reciprocal	$1/x$	3	561	4.6e − 5	4.6e − 6	1051 ± 3
erf	erf	Gaussian error function	$(2/\sqrt{\pi})\int_0^x e^{-t^2}\,\mathrm{d}t$	3	70	3.0e − 8	1.7e − 8	626 ± 2
	erfc	complementary erf	$1-\mathrm{erf}(x)$	3	92	4.2e − 8	1.6e − 8	772 ± 2
sign testing	abs	absolute value	$\lvert x \rvert$	1	2	0	0	114 ± 0
	signum	sign number	$\mathrm{sgn}(x)$	0	3	0	0	114 ± 0
	positive	test strictly positive	$[x > 0]$	0	2	0	0	114 ± 0
	negative	test strictly negative	$[x < 0]$	0	2	0	0	114 ± 0
	nonneg	test non-negative	$[x \geq 0]$	0	2	0	0	114 ± 0
	nonpos	test non-positive	$[x \leq 0]$	0	2	0	0	114 ± 0
	zero	test exactly zero	$[x \stackrel{?}{=} 0]$	0	2	0	0	57 ± 0
	nonzero	test non-zero	$[x \neq 0]$	0	2	0	0	57 ± 0
bits	clz	count leading zeros	$\mathrm{clz}(x)$	0	49	0	0	1665 ± 1
	clrsb	count redundant sign bits	$\mathrm{clrsb}(x)$	0	95	0	0	3207 ± 1
activation functions	ReLU	rectified linear unit	$\max(0,x)$	1	2	0	0	114 ± 0
	ReLU6	clipped ReLU	$\min(\max(0,x),6)$	1	3	0	0	128 ± 2
	LeakyReLU	leaky ReLU	$\max(0,x)+\min(0,x/100)$	1	2	0	0	114 ± 0
	ReLU^2	squared ReLU	$\max(0,x^2)$	2	2	0	0	114 ± 0
	GELU	Gaussian error linear unit	$x(1+\mathrm{erf}(x/\sqrt{2}))/2$	3	81	6.0e − 6	2.5e − 7	712 ± 2
	HardELiSH	"hard" ELiSH	$\min(e^x-1,\lvert x \rvert)\max(0,\min(1,\frac{x+1}{2}))$	3	38	4.2e − 8	1.2e − 8	351 ± 2
	Hardshrink	"hard" shrink	$\lvert(\lvert x) < 1\,?\,0 : x$	1	3	0	0	182 ± 1
	Hardsigmoid	"hard" logistic sigmoid	$x < -3\,?\,0 : (x > 3\,?\,1 : (x+3)/6)$	1	3	0	0	128 ± 2
	Hardswish	"hard" swish	$x < -3\,?\,0 : (x > 3\,?\,1 : x\cdot(x+3)/6)$	2	3	0	0	128 ± 1
	Hardtanh	"hard" hyperbolic tangent	$x < -1\,?\,-1 : (x > 1\,?\,1 : x)$	1	3	0	0	126 ± 2
	Softplus	"soft" plus	$\ln(1+e^x)$	3	94	1.2e − 7	2.2e − 8	960 ± 2
	Softminus	"soft" minus	$x-\mathrm{Softplus}(x)$	3	93	1.2e − 7	2.2e − 8	946 ± 2
	Softsign	"soft" sign	$x/(1+\lvert x \rvert)$	3	182	6.2e − 7	3.9e − 8	2247 ± 3
	Softshrink	"soft" shrink	$\mathrm{sgn}(x)\cdot\max(\lvert(\lvert x)-1,0)$	1	3	0	0	125 ± 1
	ELU	exponential linear unit	$\max(\alpha e^x-1,x)$	3	46	3.0e − 8	1.7e − 8	496 ± 3
	sigmoid	logistic sigmoid	$1/(1+e^{-x})$	3	98	1.1e − 7	1.4e − 8	991 ± 2
	SiLU	sigmoid linear unit	$x\,\mathrm{sigmoid}(x)$	3	108	1.2e − 7	2.6e − 8	1046 ± 2
	CELU	continuously differentiable ELU	$\max(0,x)+\min(0,e^x-1)$	3	47	3.0e − 8	1.7e − 8	507 ± 2
	ELiSH	exponential-linear squashing	$x < 0\,?\,\frac{x}{1+e^{-x}} : \frac{e^x-1}{1+e^{-x}}$	3	115	1.2e − 7	2.4e − 8	1093 ± 3
	Mish	Misra's swish	$x\tanh(\mathrm{Softplus}(x))$	3	106	1.2e − 7	1.8e − 8	1005 ± 3
	LeCunTanh	LeCun's hyperbolic tangent	$1.7159\tanh(\frac{2}{3}x)$	3	89	4.2e − 8	2.3e − 8	860 ± 2
	TanhExp	tanh-exponential	$x\tanh(e^x)$	3	100	6.0e − 8	1.1e − 8	931 ± 3
	TanhShrink	tanh shrink	$x-\tanh(x)$	3	99	8.4e − 8	5.9e − 8	852 ± 2
	Serf	log-softplus error	$x\,\mathrm{erf}(\mathrm{Softplus}(x))$	3	102	5.9e − 8	1.5e − 8	943 ± 2
	logsigmoid	natural ogarithm of sigmoid	$\ln\mathrm{sigmoid}(x)$	3	67	4.1e − 5	7.2e − 6	682 ± 2
	tanh	hyperbolic tangent	$(e^x-e^{-x})/(e^x+e^{-x})$	3	84	3.0e − 8	1.7e − 8	770 ± 3

[†]Assuming 64-bit fixed-point arithmetic using 16-bits to represent the fractional part.
[‡]This function has one or more poles; error measurements exclude points with a distance less than 1.5e − 3 from a pole.
[¶]This is a periodic function for which only the principle domain is included in the LUT

Appendix E. Formulae for (2+1)-PC sign-corrected polynomial evaluation

E.1. One-round ABY2.0-like evaluation

E.1.1. For constant polynomials.

Precomputation: P_2 samples U and A_0, computes $[U]$, $[A_0]$, $[U{\cdot}A_0]$, and shares all five them among P_0 and P_1.

Round 1: P_b sends $[\bar{u}]_b = [u]_b + [U]_b$ and $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $[y]_b = \bar{u}{\cdot}\bar{a}_0{\cdot}b - \bar{u}{\cdot}[A_0]_0 - \bar{a}_0{\cdot}[U]_0 + [U{\cdot}A_0]_0$

The result is $[y] = \bar{u}{\cdot}\bar{a}_0 - \bar{u}{\cdot}[A_0] - \bar{a}_0{\cdot}[U] + [U{\cdot}A_0]$

This is derived by:

$$\begin{aligned}
[y] &= [y]_0 + [y]_1 \\
&= \bar{u}{\cdot}\bar{a}_0 - \bar{u}{\cdot}[A_0] - \bar{a}_0{\cdot}[U] + [U{\cdot}A_0] \\
&= (\bar{u} - [U]){\cdot}(\bar{a}_0 - [A_0]) \\
&= u{\cdot}a_0
\end{aligned}$$

E.1.2. For linear polynomials.

Precomputation: P_2 samples A_0, A_1, U, and X, computes $[U{\cdot}X]$, $[U{\cdot}A_1]$, $[A_1{\cdot}X - A_0]$, $[U{\cdot}(A_1{\cdot}X - A_0)]$, and shares all eight among P_0 and P_1.

Round 1: P_b sends $[\bar{u}]_b = [u]_b + [U]_b$, $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$, $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$, and $[\bar{x}]_b = [x]_b + [X]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $[y]_b = \bar{u}{\cdot}(b{\cdot}(\bar{a}_1{\cdot}\bar{x} + \bar{a}_0) - \bar{a}_1{\cdot}[X]_b - \bar{x}{\cdot}[A_1]_b + [A_1{\cdot}X - A_0]_b) - (\bar{a}_1{\cdot}\bar{x} + \bar{a}_0){\cdot}[U]_b + \bar{a}_1{\cdot}[U{\cdot}X]_b + \bar{x}{\cdot}[U{\cdot}A_1]_b - [U{\cdot}(A_1{\cdot}X - A_0)]_b$

The result is $[y] = \bar{u}{\cdot}((\bar{a}_1{\cdot}\bar{x} + \bar{a}_0) - \bar{a}_1{\cdot}[X] - \bar{x}{\cdot}[A_1] + [A_1{\cdot}X - A_0]) - (\bar{a}_1{\cdot}\bar{x} + \bar{a}_0){\cdot}[U] + \bar{a}_1{\cdot}[U{\cdot}X] + \bar{x}{\cdot}[U{\cdot}A_1] - [U{\cdot}(A_1{\cdot}X - A_0)]$

This is derived by:

$$\begin{aligned}
[y] &= [y]_0 + [y]_1 \\
&= \bar{u}{\cdot}(\bar{a}_1{\cdot}\bar{x} + \bar{a}_0 - \bar{a}_1{\cdot}[X] - \bar{x}{\cdot}[A_1] + [A_1{\cdot}X - A_0]) \\
&\quad - (\bar{a}_1{\cdot}\bar{x} + \bar{a}_0){\cdot}[U] + \bar{a}_1{\cdot}[U{\cdot}X] + \bar{x}{\cdot}[U{\cdot}A_1] \\
&\quad - [U{\cdot}(A_1{\cdot}X - A_0)] \\
&= \bar{u}{\cdot}\bar{a}_1{\cdot}\bar{x} - \bar{u}{\cdot}\bar{a}_1{\cdot}[X] - \bar{u}{\cdot}\bar{x}{\cdot}[A_1] - \bar{a}_1{\cdot}\bar{x}{\cdot}[U] \\
&\quad + \bar{u}{\cdot}[A_1{\cdot}X] + \bar{a}_1{\cdot}[U{\cdot}X] + \bar{x}{\cdot}[U{\cdot}A_1] - [U{\cdot}A_1{\cdot}X] \\
&\quad + \bar{u}{\cdot}\bar{a}_0 - \bar{u}{\cdot}[A_0] - \bar{a}_0{\cdot}[U] + [U{\cdot}A_0] \\
&= \bar{u}{\cdot}\bar{a}_1{\cdot}\bar{x} - \bar{u}{\cdot}\bar{a}_1{\cdot}[X] - \bar{u}{\cdot}\bar{x}{\cdot}[A_1] - \bar{a}_1{\cdot}\bar{x}{\cdot}[U] \\
&\quad + \bar{u}{\cdot}[A_1{\cdot}X] + \bar{a}_1{\cdot}[U{\cdot}X] + \bar{x}{\cdot}[U{\cdot}A_1] - [U{\cdot}A_1{\cdot}X] \\
&\quad + u{\cdot}a_0{\cdot}2^p \\
&= (\bar{u} - [U]){\cdot}(\bar{a}_1 - [A_1]){\cdot}(\bar{x} - [X]) + u{\cdot}a_0{\cdot}2^p \\
&= u{\cdot}a_1{\cdot}x + u{\cdot}a_0{\cdot}2^p \\
&= u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p)
\end{aligned}$$

E.1.3. For quadratic polynomials.

Precomputation: P_2 samples A_0, A_1, A_2, U, and X, computes $[X^2]$, $[U{\cdot}A_2]$, $[U{\cdot}X]$, $[U{\cdot}X^2]$, $[2{\cdot}A_2{\cdot}X - A_1]$, $[A_2{\cdot}X^2 - A_1{\cdot}X + A_0]$, $[U{\cdot}(2{\cdot}A_2{\cdot}X - A_1)]$, $[U{\cdot}(A_2{\cdot}X^2 - A_1{\cdot}X + A_0)]$, and shares all thirteen among P_0 and P_1.

Round 1: P_b sends $[\bar{u}]_b = [u]_b + [U]_b$, $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$, $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$, $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$, and $[\bar{x}]_b = [x]_b + [X]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $[y]_b = \bar{u}{\cdot}(b{\cdot}(\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x} + \bar{a}_0) - \bar{x}^2{\cdot}[A_2]_b + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1]_b - (\bar{a}_1 + 2{\cdot}\bar{a}_2{\cdot}\bar{x}){\cdot}[X]_b + \bar{a}_2{\cdot}[X^2]_b - [A_2{\cdot}X^2 - A_1{\cdot}X + A_0]_b) - [U]_b{\cdot}(\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x} + \bar{a}_0) + \bar{x}^2{\cdot}[U{\cdot}A_2]_b - \bar{x}{\cdot}[U{\cdot}(2{\cdot}A_2{\cdot}X - A_1)]_b + (\bar{a}_1 + 2{\cdot}\bar{a}_2{\cdot}\bar{x}){\cdot}[U{\cdot}X]_b - \bar{a}_2{\cdot}[U{\cdot}X^2]_b + [U{\cdot}(A_2{\cdot}X^2 - A_1{\cdot}X + A_0)]_b$

The result is $[y] = \bar{u}{\cdot}((\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x} + \bar{a}_0) - \bar{x}^2{\cdot}[A_2] + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1] - (\bar{a}_1 + 2{\cdot}\bar{a}_2{\cdot}\bar{x}){\cdot}[X] + \bar{a}_2{\cdot}[X^2] - [A_2{\cdot}X^2 - A_1{\cdot}X + A_0]) - [U]{\cdot}(\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x} + \bar{a}_0) + \bar{x}^2{\cdot}[U{\cdot}A_2] - \bar{x}{\cdot}[U{\cdot}(2{\cdot}A_2{\cdot}X - A_1)] + (\bar{a}_1 + 2{\cdot}\bar{a}_2{\cdot}\bar{x}){\cdot}[U{\cdot}X] - \bar{a}_2{\cdot}[U{\cdot}X^2] + [U{\cdot}(A_2{\cdot}X^2 - A_1{\cdot}X + A_0)]$

This is derived by:

$$\begin{aligned}
[y] &= [y]_0 + [y]_1 \\
&= \bar{u}{\cdot}((\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x} + \bar{a}_0) - (2{\cdot}\bar{x}{\cdot}\bar{a}_2 + \bar{a}_1){\cdot}[X] \\
&\qquad + \bar{a}_2{\cdot}[X^2] - \bar{x}^2{\cdot}[A_2] + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1] \\
&\qquad - [A_2{\cdot}X^2 - A_1{\cdot}X + A_0]) \\
&\qquad - (\bar{a}_2 + \bar{a}_1{\cdot}\bar{x} + \bar{a}_0){\cdot}\bar{x}^2[U] \\
&\qquad + (2{\cdot}\bar{x}{\cdot}\bar{a}_2 + \bar{a}_1){\cdot}[U{\cdot}X] - \bar{a}_2{\cdot}[U{\cdot}X^2] \\
&\qquad + \bar{x}^2{\cdot}[U{\cdot}A_2] - \bar{x}{\cdot}[U{\cdot}(2{\cdot}A_2{\cdot}X - A_1)] \\
&\qquad + [U{\cdot}(A_2{\cdot}X^2 - A_1{\cdot}X + A_0)] \\
&= \bar{u}{\cdot}(\bar{a}_2{\cdot}\bar{x}^2 - 2{\cdot}\bar{x}{\cdot}\bar{a}_2{\cdot}[X] + \bar{a}_2{\cdot}[X^2] \\
&\qquad - \bar{x}^2{\cdot}[A_2] + 2{\cdot}\bar{x}{\cdot}[A_2{\cdot}X] - [A_2{\cdot}X^2]) \\
&\qquad - \bar{a}_2{\cdot}\bar{x}^2[U] + 2{\cdot}\bar{x}{\cdot}\bar{a}_2{\cdot}[U{\cdot}X] \\
&\qquad - \bar{a}_2{\cdot}[U{\cdot}X^2] + \bar{x}^2{\cdot}[U{\cdot}A_2] - 2{\cdot}\bar{x}{\cdot}[U{\cdot}A_2{\cdot}X] \\
&\qquad + [U{\cdot}A_2{\cdot}X^2] + \bar{u}{\cdot}((\bar{a}_1{\cdot}\bar{x} + \bar{a}_0) - \bar{a}_1{\cdot}[X] \\
&\qquad - \bar{x}{\cdot}[A_1] + [A_1{\cdot}X - A_0]) - (\bar{a}_1{\cdot}\bar{x} + \bar{a}_0){\cdot}[U] \\
&\qquad + \bar{a}_1{\cdot}[U{\cdot}X] + \bar{x}{\cdot}[U{\cdot}A_1] - [U{\cdot}(A_1{\cdot}X - A_0)] \\
&= \bar{u}{\cdot}(\bar{a}_2{\cdot}\bar{x}^2 - 2{\cdot}\bar{x}{\cdot}\bar{a}_2{\cdot}[X] + \bar{a}_2{\cdot}[X^2] \\
&\qquad - \bar{x}^2{\cdot}[A_2] + 2{\cdot}\bar{x}{\cdot}[A_2{\cdot}X] - [A_2{\cdot}X^2]) \\
&\qquad - \bar{a}_2{\cdot}\bar{x}^2[U] + 2{\cdot}\bar{x}{\cdot}\bar{a}_2{\cdot}[U{\cdot}X] - \bar{a}_2{\cdot}[U{\cdot}X^2] \\
&\qquad + \bar{x}^2{\cdot}[U{\cdot}A_2] - 2{\cdot}\bar{x}{\cdot}[U{\cdot}A_2{\cdot}X] + [U{\cdot}A_2{\cdot}X^2] \\
&\qquad + u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p){\cdot}2^p \\
&= (\bar{u}{\cdot}\bar{a}_2{\cdot}\bar{x}^2 - 2{\cdot}\bar{u}{\cdot}\bar{x}{\cdot}\bar{a}_2{\cdot}[X] + \bar{u}{\cdot}\bar{a}_2{\cdot}[X^2] \\
&\qquad - \bar{u}{\cdot}\bar{x}^2{\cdot}[A_2] + 2{\cdot}\bar{u}{\cdot}\bar{x}{\cdot}[A_2{\cdot}X] - \bar{u}{\cdot}[A_2{\cdot}X^2]) \\
&\qquad - \bar{a}_2{\cdot}\bar{x}^2[U] + 2{\cdot}\bar{x}{\cdot}\bar{a}_2{\cdot}[U{\cdot}X] - \bar{a}_2{\cdot}[U{\cdot}X^2] \\
&\qquad + \bar{x}^2{\cdot}[U{\cdot}A_2] - 2{\cdot}\bar{x}{\cdot}[U{\cdot}A_2{\cdot}X] + [U{\cdot}A_2{\cdot}X^2] \\
&\qquad + u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p){\cdot}2^p \\
&= (\bar{u} - [U]){\cdot}(\bar{a}_2 - [A_2]){\cdot}(\bar{x} - [X])^2 \\
&\qquad + u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p){\cdot}2^p \\
&= u{\cdot}a_2{\cdot}x^2 + u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p){\cdot}2^p \\
&= u{\cdot}\left(a_2{\cdot}x^2 + a_1{\cdot}x{\cdot}2^p + a_0{\cdot}2^{2p}\right)
\end{aligned}$$

E.1.4. For cubic polynomials.

TABLE 3: Comparing round complexity and communication cost to evaluate polynomials using $(2+1)$-party protocols. The communication cost is expressed as the number of values exchanged as part of the Beaver triple–like from P_2 (in the preprocessing phase) and between P_0 and P_1 (in the online phase). The bitlength of each term depends on the polynomial degree and fractional precision; see Section 7.1.2 for details on how to calculate them.

Protocol	Polynomial degree	Round complexity	Communication: Preprocessing	Communication: Online
One-round ABY2.0	1	1	8	4
	2	1	13	5
	3	1	18	6
Two-round ABY2.0-like ABY2.0	1	2	6	4
	2	2	9	5
	3	2	13	6
Horner's method	1	2	6	4
	2	3	8	5
	3	4	10	6

Precomputation: P_2 samples A_0, A_1, A_2, A_3, U, and X, computes $[X^2]$, $[X^3]$, $[3\cdot A_3\cdot X - A_2]$, $[3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1]$, $[A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0]$, $[U\cdot X]$, $[U\cdot X^2]$, $[U\cdot X^3]$, $[U\cdot A_3]$, $[U\cdot(3\cdot A_3\cdot X - A_2)]$, $[U\cdot(3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1)]$, $[U\cdot(A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0)]$, and shares all eighteen among P_0 and P_1.

Round 1: P_b sends $[\bar{u}]_b = [u]_b + [U]_b$, $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$, $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$, $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$, $[\bar{a}_3]_b = [a_3]_b + [A_3]_b$, and $[\bar{x}]_b = [x]_b + [X]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $[y]_b = \bar{u}\cdot\big(b\cdot(\bar{a}_3\cdot\bar{x}^3 + \bar{a}_2\cdot\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0) - (3\cdot\bar{a}_3\cdot\bar{x}^2 + 2\cdot\bar{a}_2\cdot\bar{x} + \bar{a}_1)\cdot[X]_b + (3\cdot\bar{a}_3\cdot\bar{x} + \bar{a}_2)\cdot[X^2]_b - \bar{a}_3\cdot[X^3]_b - \bar{x}^3\cdot[A_3]_b + \bar{x}^2\cdot[3\cdot A_3\cdot X - A_2]_b - \bar{x}\cdot[3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1]_b + [A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0]_b\big) - [U]_b\cdot(\bar{a}_3\cdot\bar{x}^3 + \bar{a}_2\cdot\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0) + (3\cdot\bar{a}_3\cdot\bar{x}^2 + 2\cdot\bar{a}_2\cdot\bar{x} + \bar{a}_1)\cdot[U\cdot X]_b - (3\cdot\bar{a}_3\cdot\bar{x} + \bar{a}_2)\cdot[U\cdot X^2]_b + \bar{a}_3\cdot[U\cdot X^3]_b + \bar{x}^3\cdot[U\cdot A_3]_b - \bar{x}^2\cdot[U\cdot(3\cdot A_3\cdot X - A_2)]_b + \bar{x}\cdot[U\cdot(3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1)]_b - [U\cdot(A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0)]_b$

The result is $[y] = \bar{u}\cdot\big((\bar{a}_3\cdot\bar{x}^3 + \bar{a}_2\cdot\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0) - (3\cdot\bar{a}_3\cdot\bar{x}^2 + 2\cdot\bar{a}_2\cdot\bar{x} + \bar{a}_1)\cdot[X] + (3\cdot\bar{a}_3\cdot\bar{x} + \bar{a}_2)\cdot[X^2] - \bar{a}_3\cdot[X^3] - \bar{x}^3\cdot[A_3] + \bar{x}^2\cdot[3\cdot A_3\cdot X - A_2] - \bar{x}\cdot[3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1] + [A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0]\big) - [U]\cdot(\bar{a}_3\cdot\bar{x}^3 + \bar{a}_2\cdot\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0) + (3\cdot\bar{a}_3\cdot\bar{x}^2 + 2\cdot\bar{a}_2\cdot\bar{x} + \bar{a}_1)\cdot[U\cdot X] - (3\cdot\bar{a}_3\cdot\bar{x} + \bar{a}_2)\cdot[U\cdot X^2] + \bar{a}_3\cdot[U\cdot X^3] + \bar{x}^3\cdot[U\cdot A_3] - \bar{x}^2\cdot[U\cdot(3\cdot A_3\cdot X - A_2)] + \bar{x}\cdot[U\cdot(3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1)] - [U\cdot(A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0)]$

This is derived by:

$$
\begin{aligned}
[y] &= [y]_0 + [y]_1 \\
&= \bar{u}\cdot((\bar{a}_3\cdot\bar{x}^3 + \bar{a}_2\cdot\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0) \\
&\qquad - (3\cdot\bar{a}_3\cdot\bar{x}^2 + 2\cdot\bar{a}_2\cdot\bar{x} + \bar{a}_1)\cdot[X] + (3\cdot\bar{a}_3\cdot\bar{x} + \bar{a}_2)\cdot[X^2] \\
&\qquad - \bar{a}_3\cdot[X^3] - \bar{x}^3\cdot[A_3] + \bar{x}^2\cdot[3\cdot A_3\cdot X - A_2] \\
&\qquad - \bar{x}\cdot[3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1] \\
&\qquad + [A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0]) \\
&\qquad - (\bar{a}_3\cdot\bar{x}^3 + \bar{a}_2\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0)\cdot[U] \\
&\qquad + (3\cdot\bar{a}_3\cdot\bar{x}^2 + 2\cdot\bar{a}_2\cdot\bar{x} + \bar{a}_1)\cdot[U\cdot X] + \bar{x}^3\cdot[U\cdot A_3] \\
&\qquad - \bar{x}^2\cdot[U\cdot(3\cdot A_3\cdot X - A_2)] \\
&\qquad + \bar{x}\cdot[U\cdot(3\cdot A_3\cdot X^2 - 2\cdot A_2\cdot X + A_1)] \\
&\qquad - [U\cdot(A_3\cdot X^3 - A_2\cdot X^2 + A_1\cdot X - A_0)] \\
&= \bar{u}\cdot(\bar{a}_3\cdot\bar{x}^3 - 3\cdot\bar{a}_3\cdot\bar{x}^2\cdot[X] + 3\cdot\bar{a}_3\cdot\bar{x}\cdot[X^2] - \bar{a}_3\cdot[X^3] - \bar{x}^3\cdot[A_3] \\
&\qquad + 3\cdot\bar{x}^2\cdot[A_3\cdot X] - 3\cdot\bar{x}\cdot[A_3\cdot X^2] + [A_3\cdot X^3]) - \bar{a}_3\cdot\bar{x}^3\cdot[U] \\
&\qquad + 3\cdot\bar{a}_3\cdot\bar{x}^2\cdot[U\cdot X] - 3\cdot\bar{a}_3\cdot\bar{x}\cdot[U\cdot X^2] + \bar{a}_3\cdot[U\cdot X^3] \\
&\qquad + \bar{x}^3\cdot[U\cdot A_3] - 3\cdot\bar{x}^2\cdot[U\cdot A_3\cdot X] + 3\cdot\bar{x}\cdot[U\cdot A_3\cdot X^2] \\
&\qquad - [U\cdot A_3\cdot X^3] + \bar{u}\cdot((\bar{a}_2\cdot\bar{x}^2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0) \\
&\qquad - (2\cdot\bar{x}\cdot\bar{a}_2 + \bar{a}_1)\cdot[X] + \bar{a}_2\cdot[X^2] \\
&\qquad - \bar{x}^2\cdot[A_2] + \bar{x}\cdot[2\cdot A_2\cdot X - A_1] - [A_2\cdot X^2 - A_1\cdot X + A_0]) \\
&\qquad - (\bar{a}_2 + \bar{a}_1\cdot\bar{x} + \bar{a}_0)\cdot\bar{x}^2[U] + (2\cdot\bar{x}\cdot\bar{a}_2 + \bar{a}_1)\cdot[U\cdot X] \\
&\qquad - \bar{a}_2\cdot[U\cdot X^2] + \bar{x}^2\cdot[U\cdot A_2] - \bar{x}\cdot[U\cdot(2\cdot A_2\cdot X - A_1)] \\
&\qquad + [U\cdot(A_2\cdot X^2 - A_1\cdot X + A_0)] \\
&= \bar{u}\cdot(\bar{a}_3\cdot\bar{x}^3 - 3\cdot\bar{a}_3\cdot\bar{x}^2\cdot[X] + 3\cdot\bar{a}_3\cdot\bar{x}\cdot[X^2] - \bar{a}_3\cdot[X^3] - \bar{x}^3\cdot[A_3] \\
&\qquad + 3\cdot\bar{x}^2\cdot[A_3\cdot X] - 3\cdot\bar{x}\cdot[A_3\cdot X^2] + [A_3\cdot X^3]) \\
&\qquad - \bar{a}_3\cdot\bar{x}^3\cdot[U] + 3\cdot\bar{a}_3\cdot\bar{x}^2\cdot[U\cdot X] - 3\cdot\bar{a}_3\cdot\bar{x}\cdot[U\cdot X^2] \\
&\qquad + \bar{a}_3\cdot[U\cdot X^3] + \bar{x}^3\cdot[U\cdot A_3] - 3\cdot\bar{x}^2\cdot[U\cdot A_3\cdot X] \\
&\qquad + 3\cdot\bar{x}\cdot[U\cdot A_3\cdot X^2] - [U\cdot A_3\cdot X^3] \\
&\qquad + u\cdot\left(a_2\cdot x^2 + a_1\cdot x\cdot 2^p + a_0\cdot 2^{2p}\right)\cdot 2^p \\
&= (\bar{u} - [U])\cdot(\bar{a}_3\cdot\bar{x}^3 - 3\cdot\bar{a}_3\cdot\bar{x}^2\cdot[X] + 3\cdot\bar{a}_3\cdot\bar{x}\cdot[X^2] - \bar{a}_3\cdot[X^3] \\
&\qquad - \bar{x}^3\cdot[A_3] + 3\cdot\bar{x}^2\cdot[A_3\cdot X] - 3\cdot\bar{x}\cdot[A_3\cdot X^2] + [A_3\cdot X^3]) \\
&\qquad + u\cdot\left(a_2\cdot x^2 + a_1\cdot x\cdot 2^p + a_0\cdot 2^{2p}\right)\cdot 2^p \\
&= (\bar{u} - [U])\cdot(\bar{a}_3 - [A_3])\cdot(\bar{x} - [X])^3 \\
&\qquad + u\cdot\left(a_2\cdot x^2 + a_1\cdot x\cdot 2^p + a_0\cdot 2^{2p}\right)\cdot 2^p \\
&= u\cdot a_3\cdot x^3 + u\cdot\left(a_2\cdot x^2 + a_1\cdot x\cdot 2^p + a_0\cdot 2^{2p}\right)\cdot 2^p \\
&= u\cdot\left(a_3\cdot x^3 + a_2\cdot x^2\cdot 2^p + a_1\cdot x\cdot 2^{2p} + a_0\cdot 2^{3p}\right)
\end{aligned}
$$

E.2. Two-round ABY2.0-like evaluation

E.2.1. For constant polynomials. Since sign-correcting a constant polynomial only requires a single multiplication as described in section E.1.1, a Two-round ABY2.0-like evaluation is not required.

E.2.2. For linear polynomials.

Precomputation: P_2 samples A_1, U, X, and Y, computes $[A_1{\cdot}X]$, $[U{\cdot}Y]$, and shares all six among P_0 and P_1.

Round 1: P_b sends $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$ and $[\bar{x}]_b = [x]_b + [X]_b$ to P_{1-b}, and vice versa.

Round 2: P_b sends $[\bar{y}]_b = [y]_b + [Y]_b$ and $[\bar{u}]_b = [u]_b + [U]_b$ to P_{1-b}, and vice versa. Here $[y]_b = b{\cdot}(\bar{a}_1{\cdot}\bar{x}) - \bar{a}_1{\cdot}[X]_b - \bar{x}{\cdot}[A_1]_b + [A_1{\cdot}X]_b + [a_0]{\cdot}2^p$.

Output: P_b outputs $[u{\cdot}y]_b = b{\cdot}\bar{x}{\cdot}\bar{y} - \bar{x}{\cdot}[X]_b - \bar{y}{\cdot}[Y]_b + [X{\cdot}Y]_b$

The result of the first round of online communication is $[y] = (\bar{a}_1{\cdot}\bar{x}) - \bar{a}_1{\cdot}[X] - \bar{x}{\cdot}[A_1] + [A_1{\cdot}X] + [a_0]{\cdot}2^p$

This is derived by:

$$\begin{aligned}[y] &= [y]_0 + [y]_1 \\ &= \bar{a}_1{\cdot}\bar{x} - \bar{x}{\cdot}[A_1] - \bar{a}_1{\cdot}[X] + [A_1{\cdot}X] + [a_0]{\cdot}2^p \\ &= (\bar{a}_1 - [A_1]){\cdot}(\bar{x} - [X]) + [a_0]{\cdot}2^p \\ &= a_1{\cdot}x + a_0{\cdot}2^p\end{aligned}$$

The final Du-Atallah multiplication produces, $u{\cdot}[y] = u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p)$, the sign corrected result.

E.2.3. For quadratic polynomials.

Precomputation: P_2 samples A_1, A_2, U, X, and Y, computes $[X^2]$, $[2{\cdot}A_2{\cdot}X - A_1]$, $[A_2{\cdot}X^2 - A_1{\cdot}X]$, $[U{\cdot}Y]$, and shares all nine among P_0 and P_1.

Round 1: P_b sends $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$, $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$, and $[\bar{x}]_b = [x]_b + [X]_b$ to P_{1-b}, and vice versa.

Round 2: P_b sends $[\bar{y}]_b = [y]_b + [Y]_b$ and $[\bar{u}]_b = [u]_b + [U]_b$ to P_{1-b}, and vice versa. Here $[y]_b = b{\cdot}(\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - \bar{x}^2{\cdot}[A_2]_b + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1]_b - (\bar{a}_1 + 2{\cdot}\bar{a}_2{\cdot}\bar{x}){\cdot}[X]_b + \bar{a}_2{\cdot}[X^2]_b - [A_2{\cdot}X^2 - A_1{\cdot}X]_b + [a_0]_b{\cdot}2^{2p}$.

Output: P_b outputs $[u{\cdot}y]_b = b{\cdot}\bar{x}{\cdot}\bar{y} - \bar{x}{\cdot}[X]_b - \bar{y}{\cdot}[Y]_b + [X{\cdot}Y]_b$

The result of the first round of online communication is $[y] = (\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - \bar{x}^2{\cdot}[A_2] + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1] - (\bar{a}_1 + 2{\cdot}\bar{a}_2{\cdot}\bar{x}){\cdot}[X] + \bar{a}_2{\cdot}[X^2] - [A_2{\cdot}X^2 - A_1{\cdot}X] + [a_0]{\cdot}2^{2p}$

This value is derived by:

$$\begin{aligned}[y] &= [y]_0 + [y]_1 \\ &= (\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - (2{\cdot}\bar{a}_2{\cdot}\bar{x} + \bar{a}_1){\cdot}[X] + \bar{a}_2{\cdot}[X^2] - \bar{x}^2{\cdot}[A_2] \\ &\quad + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1] - [A_2{\cdot}X^2 - A_1{\cdot}X] + [a_0]{\cdot}2^{2p} \\ &= (\bar{a}_2{\cdot}\bar{x}^2 - 2{\cdot}\bar{a}_2{\cdot}\bar{x}{\cdot}[X] + \bar{a}_2{\cdot}[X^2] - \bar{x}^2{\cdot}[A_2] + 2{\cdot}\bar{x}{\cdot}[A_2{\cdot}X] \\ &\quad - [A_2{\cdot}X^2]) + \bar{a}_1{\cdot}\bar{x} - \bar{x}{\cdot}[A_1] - \bar{a}_1{\cdot}[X] + [A_1{\cdot}X] \\ &\quad + [a_0]{\cdot}2^{2p} \\ &= (\bar{a}_2{\cdot}\bar{x}^2 - 2{\cdot}\bar{a}_2{\cdot}\bar{x}{\cdot}[X] + \bar{a}_2{\cdot}[X^2] - \bar{x}^2{\cdot}[A_2] + 2{\cdot}\bar{x}{\cdot}[A_2{\cdot}X] \\ &\quad - [A_2{\cdot}X^2]) + (a_1{\cdot}x + a_0{\cdot}2^p){\cdot}2^p \\ &= (\bar{a}_2 - [A_2]){\cdot}(\bar{x} - [X])^2 + (a_1{\cdot}x + a_0{\cdot}2^p){\cdot}2^p \\ &= a_2{\cdot}x^2 + a_1{\cdot}x{\cdot}2^p + a_0{\cdot}2^{2p}\end{aligned}$$

The final Du-Atallah multiplication produces the sign-corrected result $u{\cdot}y = u{\cdot}(a_2{\cdot}x^2 + a_1{\cdot}x{\cdot}2^{2p} + a_0{\cdot}2^p)$

E.2.4. For cubic polynomials.

Precomputation: P_2 samples A_1, A_2, A_3, U, X, and Y, computes $[X^2]$, $[X^3]$, $[3{\cdot}A_3{\cdot}X - A_2]$, $[3{\cdot}A_3{\cdot}X^2 - 2{\cdot}A_2{\cdot}X + A_1]$, $[A_3{\cdot}X^3 - A_2{\cdot}X^2 + A_1{\cdot}X]$, $[U{\cdot}Y]$, and shares all thirteen among P_0 and P_1.

Round 1: P_b sends $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$, $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$, $[\bar{a}_3]_b = [a_3]_b + [A_3]_b$, and $[\bar{x}]_b = [x]_b + [X]_b$ to P_{1-b}, and vice versa.

Round 2: P_b sends $[\bar{y}]_b = [y]_b + [Y]_b$ and $[\bar{u}]_b = [u]_b + [U]_b$ to P_{1-b}, and vice versa. Here $[y]_b = b{\cdot}(\bar{a}_3{\cdot}\bar{x}^3 + \bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - (3{\cdot}\bar{a}_3{\cdot}\bar{x}^2 + 2{\cdot}\bar{a}_2{\cdot}\bar{x} + \bar{a}_1){\cdot}[X]_b + (3{\cdot}\bar{a}_3{\cdot}\bar{x} + \bar{a}_2){\cdot}[X^2]_b - \bar{a}_3{\cdot}[X^3]_b - \bar{x}^3{\cdot}[A_3]_b + \bar{x}^2{\cdot}[3{\cdot}A_3{\cdot}X - A_2]_b - \bar{x}{\cdot}[3{\cdot}A_3{\cdot}X^2 - 2{\cdot}A_2{\cdot}X + A_1]_b + [A_3{\cdot}X^3 - A_2{\cdot}X^2 + A_1{\cdot}X]_b + [a_0]_b{\cdot}2^{3p}$.

Output: P_b outputs $[u{\cdot}y]_b = b{\cdot}\bar{x}{\cdot}\bar{y} - \bar{x}{\cdot}[X]_b - \bar{y}{\cdot}[Y]_b + [X{\cdot}Y]_b$

The result of the first round of online communication is $[y] = (\bar{a}_3{\cdot}\bar{x}^3 + \bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - (3{\cdot}\bar{a}_3{\cdot}\bar{x}^2 + 2{\cdot}\bar{a}_2{\cdot}\bar{x} + \bar{a}_1){\cdot}[X] + (3{\cdot}\bar{a}_3{\cdot}\bar{x} + \bar{a}_2){\cdot}[X^2] - \bar{a}_3{\cdot}[X^2] + \bar{x}^2{\cdot}[3{\cdot}A_3{\cdot}X - A_2] - \bar{x}{\cdot}[3{\cdot}A_3{\cdot}X^2 - 2{\cdot}A_2{\cdot}X + A_1] + [A_3{\cdot}X^3 - A_2{\cdot}X^2 + A_1{\cdot}X] + [a_0]{\cdot}2^{3p}$.

This value is derived by:

$$\begin{aligned}[y] &= [y]_0 + [y]_1 \\ &= (\bar{a}_3{\cdot}\bar{x}^3 + \bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - (3{\cdot}\bar{a}_3{\cdot}\bar{x}^2 + 2{\cdot}\bar{a}_2{\cdot}\bar{x} + \bar{a}_1){\cdot}[X] \\ &\quad + (3{\cdot}\bar{a}_3{\cdot}\bar{x} + \bar{a}_2){\cdot}[X^2] - \bar{a}_3{\cdot}[X^3] - \bar{x}^3{\cdot}[A_3] \\ &\quad + \bar{x}^2{\cdot}[3{\cdot}A_3{\cdot}X - A_2] - \bar{x}{\cdot}[3{\cdot}A_3{\cdot}X^2 - 2{\cdot}A_2{\cdot}X + A_1] \\ &\quad + [A_3{\cdot}X^3 - A_2{\cdot}X^2 + A_1{\cdot}X] + [a_0]{\cdot}2^{3p} \\ &= (\bar{a}_3{\cdot}\bar{x}^3 - 3{\cdot}\bar{a}_3{\cdot}\bar{x}^2{\cdot}[X] + 3{\cdot}\bar{a}_3{\cdot}\bar{x}{\cdot}[X^2] - \bar{a}_3{\cdot}[X^3] \\ &\quad - \bar{x}^3{\cdot}[A_3] + 3{\cdot}\bar{x}^2{\cdot}[A_3{\cdot}X] - 3{\cdot}\bar{x}{\cdot}[A_3{\cdot}X^2] + [A_3{\cdot}X^3]) \\ &\quad + (\bar{a}_2{\cdot}\bar{x}^2 + \bar{a}_1{\cdot}\bar{x}) - (2{\cdot}\bar{a}_2{\cdot}\bar{x} + \bar{a}_1){\cdot}[X] + \bar{a}_2{\cdot}[X^2] \\ &\quad - \bar{x}^2{\cdot}[A_2] + \bar{x}{\cdot}[2{\cdot}A_2{\cdot}X - A_1] - [A_2{\cdot}X^2 - A_1{\cdot}X] \\ &\quad + [a_0]{\cdot}2^{3p} \\ &= (\bar{a}_3{\cdot}\bar{x}^3 - 3{\cdot}\bar{a}_3{\cdot}\bar{x}^2{\cdot}[X] + 3{\cdot}\bar{a}_3{\cdot}\bar{x}{\cdot}[X^2] - \bar{a}_3{\cdot}[X^3] \\ &\quad - \bar{x}^3{\cdot}[A_3] + 3{\cdot}\bar{x}^2{\cdot}[A_3{\cdot}X] - 3{\cdot}\bar{x}{\cdot}[A_3{\cdot}X^2] + [A_3{\cdot}X^3]) \\ &\quad + (a_2{\cdot}x^2 + a_1{\cdot}x{\cdot}2^p + a_0{\cdot}2^{2p}){\cdot}2^p \\ &= (\bar{a}_3 - [A_3]){\cdot}(\bar{x} - [X])^3 + (a_2{\cdot}x^2 + a_1{\cdot}x{\cdot}2^p + a_0{\cdot}2^{2p}){\cdot}2^p \\ &= a_3{\cdot}x^3 + a_2{\cdot}x^2{\cdot}2^p + a_1{\cdot}x{\cdot}2^{2p} + a_0{\cdot}2^{3p}\end{aligned}$$

The final Du-Atallah multiplication produces the sign-corrected result $u{\cdot}y = u{\cdot}(a_3{\cdot}x^3 + a_2{\cdot}x^2{\cdot}2^p + a_1{\cdot}x{\cdot}2^{2p} + a_0{\cdot}2^{3p})$

E.3. Horner's Method evaluation

E.3.1. For linear polynomials. Identical to the two-round ABY2.0-like linear evaluation.

Precomputation: P_2 samples A_1, U, X, and Y, computes $[A_1 \cdot X]$ and $[U \cdot Y]$, and shares all six among P_0 and P_1.

Round 1: Du-Atallah multiply $a_1 \cdot x$. To do this, P_b sends $\bar{a}_1 = [a_1] + [A_1]$ and $\bar{x} = [x] + [X]$ to P_{1-b}, and vice versa.

Round 2: Du-Atallah multiply $u \cdot y_1$ where $y_1 = a_1 \cdot x + a_0{\cdot}2^p$. To do this, P_b sends $[\bar{u}]_b = [u]_b + [U]_b$ and $[\bar{y}]_b = [y]_b + [Y]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $u \cdot y_1 = u{\cdot}(a_1{\cdot}x + a_0{\cdot}2^p)$

Total:

- 6 precomputed values
- 2 rounds
- 4 values sent online by each of P_0 and P_1

E.3.2. For quadratic polynomials.

Precomputation: P_2 samples A_2, U, X, Y_1, and Y_2, computes $[A_2 \cdot X]$, $[Y_1 \cdot X]$, and $[U \cdot Y_2]$, and shares all eight among P_0 and P_1.

Round 1: Du-Atallah multiply $a_2 \cdot x$. To do this, P_b sends $\bar{a}_2 = [a_2] + [A_2]$ and $\bar{x} = [x] + [X]$ to P_{1-b}, and vice versa.

Round 2: Du-Atallah multiply $[y_1] \cdot [x]$ where $y_1 = a_2 \cdot x + a_1 \cdot 2^p$. To do this, P_b sends $[\bar{y_1}] = [y_1] + [Y_1]$ to P_{1-b}, and vice versa.

Round 3: Du-Atallah multiply $u \cdot y_2$ where $y_2 = y_1 \cdot x + a_0 \cdot 2^{2p} = (a_2 \cdot x + a_1 \cdot 2^p) \cdot x + a_0 \cdot 2^{2p}$. To do this, P_b sends $[\bar{u}]_{b:} = [u]_{b:} + [U]_b$ and $[\bar{y}]_b = [y]_b + [Y]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $u \cdot y = u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p})$

Result: $u \cdot ((a_2 \cdot x + a_1) \cdot x + a_0)$

Total:

- 8 precomputed values
- 3 rounds
- 5 values sent online by each of P_0 and P_1

E.3.3. For cubic polynomials.

Precomputation: P_2 samples A_3, U, X, Y_1, Y_2, and Y_3, computes $[A_3 \cdot X]$, $[Y_1 \cdot X]$, $[Y_2 \cdot X]$, and $[U \cdot Y_3]$, and shares all ten among P_0 and P_1.

Round 1: Du-Atallah multiply $a_3 \cdot x$. To do this, P_b sends $\bar{a}_3 = [a_3] + [A_3]$ and $\bar{x} = [x] + [X]$ to P_{1-b}, and vice versa.

Round 2: Du-Atallah multiply $[y_1] \cdot [x]$ where $y_1 = a_3 \cdot x + a_2 \cdot 2^p$. To do this, P_b sends $[\bar{y_1}] = [y_1] + [Y_1]$ to P_{1-b}, and vice versa.

Round 3: Du-Atallah multiply $[y_2] \cdot [x]$ where $y_2 = (a_3 \cdot x + a_2 \cdot 2^p) \cdot x + a_1 \cdot 2^{2p}$. To do this, P_b sends $[\bar{y_2}] = [y_2] + [Y_2]$ to P_{1-b}, and vice versa.

Round 4: Du-Atallah multiply $u \cdot y_3$ where $y_2 = y_2 \cdot x + a_0 \cdot 2^{3p} = \left((a_3 \cdot x + a_2 \cdot 2^p) \cdot x + a_1 \cdot 2^{2p}\right) \cdot x + a_0 \cdot 2^{3p}$. To do this, P_b sends $[\bar{u}]_b = [u]_b + [U]_b$ and $[\bar{y}]_b = [y]_b + [Y]_b$ to P_{1-b}, and vice versa.

Output: P_b outputs $u \cdot y = u \cdot (a_3 \cdot x^3 + a_2 \cdot x^2 \cdot 2^p + a_1 \cdot x \cdot 2^{2p} + a_0 \cdot 2^{3p})$.

Result: $u \cdot (y_2 \cdot x + a_0 = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0)$

Total:

- 10 precomputed values
- 4 rounds
- 6 values sent online by each of P_0 and P_1