The Sum-Check Protocol over the Monomial Basis, and Other Optimizations

Quang Dao[1,3], Ari Biswas[2,3], Liam Eagen[4], Andrew Milson[5], Shahar Papini[5], and Justin Thaler[6,7]

[1] Carnegie Mellon University
[2] University of Warwick
[3] LayerZero Labs
[4] Ideal Group
[5] Attestable
[6] a16z crypto research
[7] Georgetown University

Abstract. The sum-check protocol underpins SNARKs with the fastest known provers. For an n-variate polynomial g defined over a finite field $\mathbb{F}$, the protocol enables an untrusted prover to convince a verifier of the sum of all evaluations of g over a product set H^n with $H \subset \mathbb{F}$. The standard choice for H^n is the *Boolean hypercube* $\{0,1\}^n$, which serves as a natural interpolating set for multilinear polynomials.

We propose a *projective variant* of the sum-check protocol, obtained by changing the interpolating set from $\{0,1\}^n$ to the *infinity hypercube* $\{0,\infty\}^n$. Under a suitable notion of evaluation at ∞, evaluating a multilinear polynomial at a point in $\{0,\infty\}^n$ directly extracts its corresponding monomial coefficient.

This projective viewpoint is a near-drop-in replacement for applications of sum-check, requiring only local changes to polynomial representations, round identities, and evaluation formulas. It yields a $\approx 10\%$ end-to-end speedup for the sum-check prover on BN254 and on a pseudo-Mersenne 128-bit prime field, against a fair baseline. It eliminates all field subtractions when binding a multilinear polynomial, and for structured polynomials such as equality and less-than, the projective interpolants admit evaluation procedures with fewer field operations. Moreover, the monomial-coefficient form aligns naturally with polynomial commitment schemes like WHIR, removing a basis mismatch that these schemes otherwise need to work around.

Finally, we describe an optimization for sum-check over ≈ 256-bit prime fields. When targeting ≈ 128 bits of security, it suffices to sample challenges from a subset of size $\approx 2^{128}$. We show that a suitable choice of this subset, interpreted as upper-limb values in Montgomery form, yields a $1.92\times$ speedup for field multiplication. Combined with the projective binding formula, this gives a $1.82\times$ speedup for sum-check binding (a key component of fast sum-check proving).

Keywords: SNARKs · Sum-check protocol · Monomial basis · Montgomery multiplication

1 Introduction

The sum-check protocol [27] is a fundamental tool in the design of succinct proof systems. When run interactively, it is an information-theoretically sound method for verifying claims of the form

$$\sum_{(x_1,\ldots,x_n)\in H_1\times\cdots\times H_n} g(x_1,\ldots,x_n) = t,$$

where g is an n-variate polynomial of (individual) degree at most d over a finite field $\mathbb{F}$, and $H_1,\ldots,H_n \subset \mathbb{F}$ are to be summed over. From the verifier's perspective, the protocol *reduces* the expensive task of summing evaluations of g over the product set $H_1 \times \cdots \times H_n$ to the much cheaper task of evaluating g at a single random point in $\mathbb{F}^n$.

Sum-check has become a critical tool in constructing efficient succinct non-interactive arguments (SNARKs). It now underpins SNARKs for arithmetic circuit satisfiability [20,42,43], R1CS [33], Plonkish and CCS [10,35], the fastest lookup arguments [34,36], polynomial commitment schemes [2,17,18,30], folding schemes [7,24,25,8,29], and zero-knowledge virtual machines (zkVMs) like Jolt [3], SP1 Hypercube [38], and Ceno [26].

When used in current SNARKs, sum-check is most often applied to a polynomial g that is a *product* (or a low-degree function) of *multilinear* polynomials. It is also almost always the case that the evaluation domain $H_1 \times H_2 \times \cdots \times H_n$ is chosen to be the Boolean hypercube $\{0,1\}^n$.

In this paper, we argue for the following thesis:

The Boolean hypercube is not the optimal interpolating set for the sum-check protocol.

Instead, we propose a *projective* variant of sum-check, obtained by changing the interpolating set from the Boolean hypercube $\{0,1\}^n$ to the infinity hypercube $\{0,\infty\}^n$, viewed as a subset of the n-fold product of the projective line $\mathbb{P}^1 = \mathbb{F} \cup \{\infty\}$. We refer to this new interpolating set as the *projective Boolean hypercube*.

Here, the evaluation at ∞ of a univariate polynomial means its leading coefficient.[8] For multilinear polynomials, we formalize this in Section 3 via multihomogenization.

We emphasize that this is a change to the *interpolating set*, i.e., the set of evaluation points of g that are summed over by the protocol, not merely to the extrapolation points used during the protocol. In standard sum-check over $\{0,1\}$, in each round i the prover sends a univariate polynomial (the *round polynomial*) of degree at most d in variable X_i, obtained by summing out the remaining variables. The prover may evaluate this round polynomial at additional points outside the interpolating set, such as 2 or ∞, to communicate the degree-d univariate to the verifier; for instance, [4] uses ∞ as such an extrapolation point.

Our change is more fundamental: by making ∞ part of the interpolating set itself, we alter how the polynomial is represented, bound, and evaluated throughout the protocol, and points like 1 that belong to the standard set $H_i = \{0,1\}$ but not the new set $H_i = \{0,\infty\}$ become available as extrapolation points instead.

To make this precise, given a function $f\colon \{0,1\}^n \to \mathbb{F}$, we write $\widetilde{f}$ for its *Boolean multilinear extension*, i.e., the unique multilinear polynomial satisfying $\widetilde{f}(x) = f(x)$ for all $x \in \{0,1\}^n$. We write $\widehat{f}$ for its *multilinear interpolant on the infinity hypercube* $\{0,\infty\}^n$, i.e., the unique multilinear polynomial satisfying $\widehat{f}(b) = f(x)$ for all $x \in \{0,1\}^n$, where $b \in \{0,\infty\}^n$ is obtained from x by replacing each 1 with ∞.

The key observation is that the monomial coefficients of $\widehat{f}$ are *exactly* the truth-table values of f (this observation is formalized in Proposition 3.1). Equivalently, the multilinear Lagrange basis on $\{0,\infty\}^n$ is *exactly* the standard monomial basis. If

$$p(X_1, \ldots, X_n) = \sum_{S \subseteq [n]} a_S \prod_{i \in S} X_i,$$

and $b \in \{0,\infty\}^n$ satisfies $S(b) := \{i \in [n] : b_i = \infty\}$, then $p(b) = a_{S(b)}$.

These facts lead to algebraic simplifications and practical speedups in sum-check proving.

Choice of interpolating set. Why $\{0,\infty\}$ specifically? To answer this, we consider how the choice of interpolating set affects the cost of multilinear interpolation. Each variable in a multilinear polynomial plays one of three roles with respect to a Boolean input bit z: it tracks z (identity), tracks $\neg z$ (complement), or is summed over (ignored). For any two-element interpolating set, at least one of these roles requires an arithmetic operation in the corresponding Lagrange factor. After normalizing the two points, there are three canonical representatives that achieve this minimum, each placing the cost on a different role: $\{0,1\}$ pays for complement, $\{0,\infty\}$ pays for ignored, and $\{1,\infty\}$ pays for identity.

We summarize this tradeoff in the *single-element translation dictionary* (Figure 1). Using the standard Boolean hypercube $\{0,1\}^n$ pays a subtraction for every complement (since the complement of variable $x \in \{0,1\}$ equals $1 - x$).

Our choice $\{0,\infty\}^n$ eliminates this, at the cost of an addition for every ignored variable. To see what we mean by this, for any $\{a,b\} \subset \mathbb{F}$, consider the degree-1 function $\mathrm{comp}(Z) = (b - Z)/(b - a)$: comp maps b to

[8] Equivalently, consider embedding $\mathbb{F}$ into the projective line $\mathbb{P}^1(\mathbb{F})$, which is $\mathbb{F}$ adjoined with infinity. Then consider the involution ($0 \mapsto \infty,\ x \mapsto 1/x$ for all $x \neq 0$). This involution maps a univariate polynomial p to a polynomial p' whose coefficients are reversed in ordering. Thus, the evaluation at ∞ of p is equal to the evaluation at 0 of p', which is the constant term of p', and thus the leading coefficient of p.

Finite $\{a, b\} \subset \mathbb{F}$

Function	Factor
z (identity)	$(Z-a)/(b-a)$
$\neg z$ (complement)	$(b-Z)/(b-a)$
ignored	1

Projective $\{a, \infty\}$

Function	Factor
z (identity)	$Z-a$
$\neg z$ (complement)	1
ignored	$Z-a+1$

	$\{0,1\}$	$\{0,\infty\}$	$\{1,\infty\}$
z (identity)	Z	Z	$Z-1$
$\neg z$ (complement)	$1-Z$	1	1
ignored	1	$1+Z$	Z
non-free entry	$\neg z$	ignored	z

Fig. 1: Single-element translation dictionary for multilinear interpolation. For one formal variable Z and one bit z, the top panels give the general polynomial factor contributed to the interpolant under finite and projective interpolating sets. For any interpolating set, at least one dictionary entry requires an arithmetic operation. The bottom panel shows the three canonical normalized sets that achieve this minimum, each placing the cost on a different entry.

0 and a to 1. Hence, if the input b and the output 1 are both associated with "true", while the input a and the output 0 are both associated with "false", then $\mathrm{comp}(Z)$ maps the true input to false and the false input to true, consistent with logical complementation. But if $b = \infty$ then $\mathrm{comp}(Z)$ simply equals 1.[9] This is what we mean when we say that choosing $a = 0$ and $b = \infty$ eliminates the cost of negation.

The alternative set $\{a, b\} = \{1, \infty\}^n$ instead pays a subtraction for every identity, but makes ignored variables free. (To see why $Z - 1$ is the correct identity factor for $\{1, \infty\}$: it evaluates to 0 at $a = 1$, and its evaluation at $b = \infty$, meaning its leading coefficient, is 1.)

Given this cost structure, the best choice of interpolating set depends on the polynomial structure and the relative cost of addition versus subtraction in the target field; we analyze this further in Section 4.2.

Concrete gains. The following simplifications of the equality polynomial and the sum-check binding step illustrate the concrete gains from choosing $\{0, \infty\}$. The Boolean multilinear extension of the equality function is

$$\widetilde{\mathsf{eq}}((X_1, \ldots, X_n), (Y_1, \ldots, Y_n)) = \prod_{i=1}^{n} \big((1 - X_i)(1 - Y_i) + X_i Y_i\big) = \prod_{i=1}^{n} \big(1 - X_i - Y_i + 2X_i Y_i\big),$$

whereas the corresponding interpolant on the infinity hypercube is

$$\widehat{\mathsf{eq}}((X_1, \ldots, X_n), (Y_1, \ldots, Y_n)) = \prod_{i=1}^{n} (1 + X_i Y_i).$$

Thus each factor drops from a quadratic with two subtractions to a single multiplication and one addition. The same phenomenon appears in the binding step of the standard linear-time sum-check proving algorithm [15,41,39]. Instead of

$$p(r_1, x') = (1 - r_1) \cdot p(0, x') + r_1 \cdot p(1, x') = p(0, x') + r_1 \cdot (p(1, x') - p(0, x')),$$

[9] Indeed, grossly abusing algebra in this informal overview, one can think of the limit of $(b-Z)/(b-a)$ as b approaches ∞ as a constant function 1. Alternatively, one can check that the constant function 1 evaluates to 1 at 0 and to 0 at ∞, when the evaluation-at-∞ of any polynomial of degree at most 1 is defined to be the coefficient of Z (which in the case of $1 = 1 + 0 \cdot Z$, is zero).

we obtain

$$p(r_1, x') = p(0, x') + r_1 \cdot p(\infty, x'), \qquad \text{for every } x' \in \{0, \infty\}^{n-1}.$$

This removes one subtraction per binding step, for a total savings of $d \cdot (2^n - 1)$ subtractions over all rounds when the sum-check instance is a product of d multilinear polynomials.

Practical impact. Sum-check proving time is a major bottleneck in modern SNARKs, and recent works have focused on minimizing the number of field multiplications [4,16,5]. Our variant over the infinity hypercube further eliminates lower-order terms such as field subtractions, which constitute an increasing share of the cost once multiplications have been optimized. In our benchmarks (Section 6.6),[10] this translates to a $\approx \mathbf{10}\%$ end-to-end speedup for the sum-check prover on both BN254 and a 128-bit prime, against a baseline incorporating all known optimizations from [4]. Section C documents the implementation and code-generation details required to realize these gains in Rust.

These savings are more pronounced in the recursion setting, in which one needs to prove the correct execution of the sum-check verifier. In recursive SNARKs based on arithmetic circuits or constraint systems with uniform gate cost (e.g., Plonkish systems), a field addition or subtraction costs the same as a field multiplication, since all operations are native gates. This means that our optimization can save even more, about 33% at the arithmetic-operation level in binding.

1.1 Additional optimization: small challenges in large prime fields

We describe a separate optimization for SNARKs operating over large ($\sim$ 256-bit) prime fields. If we target 128 bits of security, sampling sum-check challenges uniformly from the full field gives a soundness error on the order of 2^{-256}, far smaller than necessary. Instead, we can sample challenges from a $\sim 2^{128}$-sized subset, which still yields negligible ($\sim 2^{-128}$) round-by-round soundness error via Schwartz-Zippel.

A central question is whether this restriction leads to any concrete speedup in sum-check proving. If the challenge is simply embedded into the full field before multiplication, there is no benefit. A faster option is to apply the small-value multiplication technique from prior work [4], but implementing this with Barrett reduction is nontrivial, and its practical performance is close to full field multiplication.

Our solution is different. We interpret the challenge as *already in Montgomery form*, with the 128-bit value placed in the *upper limbs*, leaving the lower limbs zero. This allows a short-circuit in the standard (heavily optimized) Montgomery multiplication routine, eliminating roughly half of the native multiplication operations. In our BN254 benchmark, this yields a 1.92× speedup for chained multiplication; when combined with the projective binding formula, the full binding benchmark improves by 1.82×. We detail this optimization in Section 5.

1.2 Related work

Optimizations to sum-check proving. There is a long line of work on optimizing the sum-check prover. Vu, Setty, Blumberg, and Walfish [41] and Thaler [39] gave linear-time proving algorithms that form the basis of most modern implementations. Cormode, Thaler, and Yi [15] introduced streaming algorithms that trade time for space; Baweja et al. [6] recently improved these to $O(M \log \log M)$ time with sublinear space. Bagad, Domb, and Thaler [5] exploited small-value structure to reduce the number of expensive field multiplications. Dao and Thaler [16] gave optimizations for the equality polynomial via a split-and-factor decomposition. These techniques, along with additional field arithmetic improvements, were consolidated and extended in [4].

Change of evaluation domain. The idea of modifying the sum-check evaluation domain has appeared in several contexts. Gruen [21] introduced the "univariate skip" technique, which replaces multiple rounds of sum-check with a single univariate round over a larger domain, achieving a time-space tradeoff. Baweja et al. [6] introduced a "mixed polynomial sumcheck" protocol (Section 7 of their paper) that replaces

[10] Benchmark code: `https://github.com/quangvdao/monomial-sumcheck-benchmarks`.

multilinear rounds with univariate rounds over multiplicative subgroups of FFT-friendly fields, yielding improved time-space tradeoffs in the PIOP model. Both of these works change the domain to compress or restructure rounds. Our work differs in that we change the *interpolating set* from $\{0,1\}$ to $\{0,\infty\}$, which changes how polynomials are represented and bound rather than how rounds are structured or how the round polynomial is communicated.

2 Preliminaries

Notation. We index vectors from 1, and denote ranges using bracket notation as: $[a,b] := \{a,\ldots,b\}$, $[n] := \{1,\ldots,n\}$, $[< i] := \{1,\ldots,i-1\}$, and $[> i] := \{i+1,\ldots,n\}$ (where n is assumed to be clear from context). We write $\mathsf{nat}(x) := \sum_{i=1}^{n} x_i \cdot 2^{i-1}$ to refer to the natural number corresponding to the binary vector $x \in \{0,1\}^n$. For $i \in \{0,\ldots,2^n-1\}$, we write $\mathsf{bits}(i) \in \{0,1\}^n$ for the binary representation of i, so that $\mathsf{nat}(\mathsf{bits}(i)) = i$. Similarly, we write $\mathsf{bits}_{\mathsf{proj}}(i) \in \{0,\infty\}^n$ for the vector obtained by replacing each 1 in $\mathsf{bits}(i)$ with ∞.

2.1 Finite fields

SNARKs that use elliptic curve-based polynomial commitments need to operate over large prime fields $\mathbb{F}_p$ of roughly 256 bits; examples include the scalar fields of the BN254, secp256k1, or BLS12-381 curves. On 64-bit architectures, elements of such fields are represented as $N = 4$ *limbs* of 64-bit unsigned integers. To accelerate multiplication, field elements are typically stored in *Montgomery form* [28]. An element $a \in \mathbb{F}_p$ is represented as $a' = aR \pmod p$, where $R = 2^{64N}$ is a power of two, chosen so that multiplication or division by R amounts to very cheap bit shifts.

Multiplying two elements a', b' in Montgomery form involves two steps. First, a large-integer multiplication computes $c = a' \cdot b'$, a $2N$-limb integer, at a cost of N^2 native multiplications. Second, a Montgomery reduction computes $cR^{-1} \pmod p$ to return the product to Montgomery form, costing an additional $N^2 + N$ native multiplications. The total cost for a field multiplication is thus approximately $2N^2 + N$ native multiplications, translating to around 80-100 CPU cycles when $N = 4$.

2.2 Multilinear polynomials

We state some facts about multilinear polynomials needed in our paper. See standard references, e.g. [40], for derivations.

Multilinear extension. An n-variate polynomial p is *multilinear* if it has degree at most 1 in each variable. The *multilinear extension* of a function $f : \{0,1\}^n \to \mathbb{F}$ is the *unique* multilinear polynomial p such that $p(x) = f(x)$ for all $x \in \{0,1\}^n$.

In particular, the *equality polynomial* $\widetilde{\mathsf{eq}}$ defined as

$$\widetilde{\mathsf{eq}}(X,Y) = \prod_{i=1}^{n} (X_i \cdot Y_i + (1-X_i)\cdot(1-Y_i)) \tag{1}$$

is the multilinear extension of the *equality function* $(\cdot \stackrel{?}{=} \cdot) : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, as we can verify that

$$\widetilde{\mathsf{eq}}(x,y) = \begin{cases} 1 \text{ if } x = y \\ 0 \text{ otherwise} \end{cases}, \quad \text{for all } x,y \in \{0,1\}^n.$$

Lemma 2.1. *Let $p\colon \mathbb{F}^n \to \mathbb{F}$ be a multilinear polynomial. Then for any $0 \le i \le n$, any $(r_1,\ldots,r_i) \in \mathbb{F}^i$, and any $x' \in \{0,1\}^{n-i}$,*

$$p(r_1,\ldots,r_i,x') = \sum_{y \in \{0,1\}^i} \widetilde{\mathsf{eq}}((r_1,\ldots,r_i),y) \cdot p(y,x'). \tag{2}$$

Setting $i = 1$, Equation (2) simplifies to

$$p(r_1, x') = (1 - r_1) \cdot p(0, x') + r_1 \cdot p(1, x'). \tag{3}$$

Setting $i = n$, we get the *multilinear extension* formula

$$p(r_1, \ldots, r_n) = \sum_{y \in \{0,1\}^n} \widetilde{\mathsf{eq}}(r_1, \ldots, r_n, y) \cdot p(y). \tag{4}$$

Lagrange interpolation. A (univariate) polynomial $s(X)$ of degree d is uniquely determined by its values at $d+1$ distinct evaluation points $U = \{x_0, \ldots, x_d\} \subset \mathbb{F}$ via the *Lagrange interpolation* formula:

$$s(X) = \sum_{i=0}^{d} s(x_i) \cdot \mathcal{L}_{U,i}(X), \text{ where } \mathcal{L}_{U,i}(X) = \prod_{j \neq i} \frac{X - x_j}{x_i - x_j}. \tag{5}$$

In our algorithms, we use a standard variant of Lagrange interpolation that takes into account the "evaluation at infinity" of a polynomial, defined as $s(\infty) :=$ the highest-degree coefficient of $s(X)$.

Lemma 2.2. *Let $s(X) = a \cdot X^d + \cdots \in \mathbb{F}[X]$ be a polynomial of degree d. Then for any distinct evaluation points $x_1, \ldots, x_d \in \mathbb{F}$, setting $U = \{\infty\} \cup \{x_1, \ldots, x_d\}$, we have the identity:*

$$s(X) = a \cdot \prod_{k=1}^{d} (X - x_k) + \sum_{k=1}^{d} s(x_k) \cdot \mathcal{L}_{\{x_i\},k}(X). \tag{6}$$

Thus, we can define $\mathcal{L}_{U,0}(X) := \prod_{k=1}^{d}(X - x_k)$ and $\mathcal{L}_{U,k}(X) := \mathcal{L}_{\{x_i\},k}(X)$ for $k \in [d]$.

2.3 The sum-check protocol

In Figure 2, we describe the sum-check protocol that reduces checking a claim of the form

$$\sum_{x \in \{0,1\}^n} g(x) = C,$$

for a polynomial g of degree at most d in each variable, to checking the evaluation of g at a random point: $g(r) = v$. We assume $d \geq 2$ and that g is a product of d multilinear polynomials $p_1, \ldots, p_d$; by linearity, existing algorithms can be straightforwardly extended if g is any degree-d function of multilinears. We write $\widehat{U}_d := \{\infty\} \cup \{1, \ldots, d-1\}$ for the d evaluation points sent by the prover in each round; the missing evaluation at 0 is derived by the verifier from the round identity.

In each round j, the honest prover sends a univariate polynomial s_j of degree d. As any degree-d univariate polynomial is specified by its evaluations on any set of $d+1$ points, computing $s_j(c)$ for all $c \in \{\infty\} \cup \{0, 1, \ldots, d-1\}$ suffices to uniquely specify s_j.[11] In fact, the prover can omit the evaluation $s_j(0)$, and instead let the verifier derive it from the previous round using the equation $s_j(0) = C_{j-1} - s_j(1)$, which holds for an honest protocol execution.

We use the notion of round-by-round soundness [9], but follow the presentation in [11].

Definition 2.3 (Round-by-round soundness [11, Chapter 23]). *Let Π be an n-round public-coin interactive proof for a relation $\mathcal{R}$. A* state function *for Π is a deterministic function St that maps an instance $\mathfrak{x}$ and a partial interaction transcript τ to a bit $\mathsf{St}(\mathfrak{x}, \tau) \in \{0, 1\}$, satisfying:*

1. $\mathsf{St}(\mathfrak{x}, \perp) = 0$ for the empty transcript $\perp$;

[11] We assume a mapping from $\{0, 1, \ldots, d-1\}$ to distinct field elements. For large enough prime fields, these integers are naturally identified with their corresponding field elements. For binary extension fields, a natural choice is to use those whose binary representation (in the chosen basis) matches that of each integer.

Setup: Assume that the verifier has oracle access to the polynomial g, and knows the value C_0 claimed to equal

$$\sum_{x \in \{0,1\}^n} g(x) = C_0.$$

For each round $i = 1, \ldots, n$**:**

1. $\mathcal{P}$ sends the univariate polynomial $s_i(X)$ claimed to equal

$$\sum_{(x_{i+1},\ldots,x_n) \in \{0,1\}^{n-i}} g(r_1, \ldots, r_{i-1}, X, x_{i+1}, \ldots, x_n).$$

$\mathcal{P}$ does so by sending the evaluations $\{s_i(u) : u \in \widehat{U}_d\}$ to $\mathcal{V}$.

2. $\mathcal{V}$ sends a random challenge $r_i \leftarrow \mathbb{F}$ to $\mathcal{P}$.

3. $\mathcal{V}$ derives $s_i(0) := C_{i-1} - s_i(1)$, then sets $C_i \coloneqq s_i(r_i)$.

After n rounds, we reduce to the claim that

$$g(r_1, \ldots, r_n) = C_n.$$

$\mathcal{V}$ checks this claim by making a single oracle query to g.

Fig. 2: The sum-check protocol.

2. if $\mathsf{St}(\mathfrak{x}, \tau) = 0$ *and* τ' *extends* τ *by one prover message, then* $\mathsf{St}(\mathfrak{x}, \tau') = 0$*;*

3. if τ *is a complete transcript with* $\mathsf{St}(\mathfrak{x}, \tau) = 0$*, then the verifier rejects.*

We say Π *has* round-by-round soundness error ϵ *if there exists a state function* St *such that for every instance* $\mathfrak{x} \notin \mathcal{L}(\mathcal{R})$*, every round* $i \in [n]$*, and every partial transcript* τ *ending with the prover's* i*th message with* $\mathsf{St}(\mathfrak{x}, \tau) = 0$*:*

$$\Pr_{r_i}\big[\, \mathsf{St}(\mathfrak{x},\, \tau \| r_i) = 1 \,\big] \;\leq\; \epsilon,$$

where r_i *is the verifier's uniformly random* i*th challenge. A union bound over* n *rounds shows that the total soundness error is at most* $n\epsilon$*.*

Theorem 2.4. *The sum-check protocol applied to a polynomial* g *of individual degree at most* d *is perfectly complete. It has round-by-round soundness error at most* $d/|\mathbb{F}|$*. Consequently, its total soundness error is at most* $dn/|\mathbb{F}|$*.*

This result goes back to Lund, Fortnow, Karloff, and Nisan [27]; see, e.g., [40] for a textbook proof. The round-by-round formulation is from [9].

Linear-time algorithm for sum-check proving. We recall the algorithm of [41,39], which runs in time linear in the instance size $M = 2^n$. The key idea is that we "bind" the polynomial after each round to this round's verifier challenge r_i. After each round i, the prover uses r_i to compute and cache the partial evaluations $p_k(r_1, \ldots, r_i, x')$ for all $k \in [d]$ and $x' \in \{0,1\}^{n-i}$. This cache, which halves in size each round, allows efficient computation of the next round's polynomial in only $O(d^2 \cdot 2^{n-i})$ time. The total runtime is $O(d^2 \cdot 2^n)$.

The binding step computes, for each multilinear polynomial p_k and each $x' \in \{0,1\}^{n-i}$:

$$p_k(r_i, x') = (1 - r_i) \cdot p_k(0, x') + r_i \cdot p_k(1, x') = p_k(0, x') + r_i \cdot \big(p_k(1, x') - p_k(0, x')\big). \tag{7}$$

As we will see in Section 3, switching the evaluation domain from $\{0, 1\}$ to $\{0, \infty\}$ eliminates the subtraction in this formula.

3 Projective evaluation and evaluation at infinity

In this section we formalize "evaluation at ∞" in a projective setting and extend it to the multivariate (especially multilinear) case that arises in applications of the sum-check protocol.

Univariate case: projective line and homogenization. Let $\mathbb{F}$ be a field. The projective line $\mathbb{P}^1(\mathbb{F})$ consists of homogeneous coordinates $[X : Z]$, with $[X : Z] = [\lambda X : \lambda Z]$ for all $\lambda \in \mathbb{F}^\times$. We embed the affine line by $x \mapsto [x : 1]$ and denote $\infty = [1 : 0]$.

For $p(X) = \sum_{k=0}^{d} a_k X^k$ with $d = \deg p$, define the homogeneous lift (homogenization)

$$\widetilde{p}(X, Z) \stackrel{\text{def}}{=} \sum_{k=0}^{d} a_k \, X^k Z^{d-k}.$$

Then $\widetilde{p}$ is homogeneous of total degree d and satisfies $\widetilde{p}(x, 1) = p(x)$ for all $x \in \mathbb{F}$. We define

$$p(\infty) \stackrel{\text{def}}{=} \widetilde{p}(1, 0) = a_d,$$

so evaluation at ∞ coincides with taking the leading coefficient.

An equivalent affine description uses inversion. Let $J : \mathbb{F}^\times \to \mathbb{F}^\times$ be $J(x) = 1/x$ and set

$$q(T) \stackrel{\text{def}}{=} T^d \, p\big(\tfrac{1}{T}\big) = \sum_{k=0}^{d} a_k T^{d-k}.$$

Then q is a polynomial and $q(0) = a_d = p(\infty)$.

Multilinear, multivariate case: product of projective lines. Let $p(X_1, \ldots, X_n)$ be multilinear:

$$p(X_1, \ldots, X_n) = \sum_{S \subseteq [n]} a_S \prod_{i \in S} X_i.$$

Introduce projective pairs $(X_i : Z_i)$ and define the multihomogenization

$$\widetilde{p}\big((X_i : Z_i)_{i=1}^n\big) \stackrel{\text{def}}{=} \sum_{S \subseteq [n]} a_S \Big(\prod_{i \in S} X_i\Big)\Big(\prod_{i \notin S} Z_i\Big).$$

This $\widetilde{p}$ is degree 1 in each pair (X_i, Z_i) and satisfies $\widetilde{p}((x_i : 1)_i) = p(x_1, \ldots, x_n)$.

Identify $0 = [0 : 1]$ and $\infty = [1 : 0]$ and define the projective Boolean hypercube $\{0, \infty\}^n \subset (\mathbb{P}^1)^n$. For $b = (b_1, \ldots, b_n) \in \{0, \infty\}^n$, let $S(b) = \{i \in [n] : b_i = \infty\}$ and evaluate

$$p(b) \stackrel{\text{def}}{=} \widetilde{p}\big((X_i : Z_i)_{i=1}^n\big), \quad \text{where } (X_i : Z_i) = \begin{cases} (0 : 1) & \text{if } b_i = 0, \\ (1 : 0) & \text{if } b_i = \infty. \end{cases}$$

Proposition 3.1 (Coefficient extraction). *For every $b \in \{0, \infty\}^n$, one has $p(b) = a_{S(b)}$.*

Proof. In $\widetilde{p}$, the summand indexed by $S \subseteq [n]$ contributes $\prod_{i \in S} X_i \prod_{i \notin S} Z_i$. With the above specialization, this equals 1 iff $S = S(b)$ and equals 0 otherwise. The sum collapses to $a_{S(b)}$.

Corollary 3.2 (Binding identity). *Fix an index i and other variables x'. Writing $p(X_i, x') = \alpha(x') + \beta(x')X_i$, one has*

$$p(0, x') = \alpha(x'), \qquad p(\infty, x') = \beta(x'), \qquad p(r, x') = p(0, x') + r \cdot p(\infty, x') \quad \forall r \in \mathbb{F}.$$

Setup: Assume that the verifier has oracle access to the polynomial g, and knows the value C_0 claimed to equal

$$\sum_{b\in\{0,\infty\}^n} g(b) = C_0.$$

For each round $i = 1, \ldots, n$**:**

1. $\mathcal{P}$ sends the univariate polynomial $s_i(X)$ claimed to equal

$$\sum_{(b_{i+1},\ldots,b_n)\in\{0,\infty\}^{n-i}} g(r_1, \ldots, r_{i-1}, X, b_{i+1}, \ldots, b_n).$$

$\mathcal{P}$ does so by sending the evaluations $\{s_i(u) : u \in \widehat{U}_d\}$ to $\mathcal{V}$.

2. $\mathcal{V}$ sends a random challenge $r_i \leftarrow \mathbb{F}$ to $\mathcal{P}$.
3. $\mathcal{V}$ derives $s_i(0) := C_{i-1} - s_i(\infty)$, then sets $C_i \coloneqq s_i(r_i)$.

After n rounds, we reduce to the claim that

$$g(r_1, \ldots, r_n) = C_n.$$

$\mathcal{V}$ checks this claim by making a single oracle query to g.

Fig. 3: The projective sum-check protocol. Compared to Figure 2, the interpolating set changes from $\{0, 1\}$ to $\{0, \infty\}$, and the verifier derives $s_i(0)$ from the leading coefficient $s_i(\infty)$ rather than from $s_i(1)$.

Basis perspective. Within the multilinear space, the monomials $\{\prod_{i\in S} X_i\}_{S\subseteq[n]}$ are exactly the Lagrange basis for interpolation on $\{0, \infty\}^n$: each monomial evaluates to 1 at its matching point and to 0 at all others. Equivalently, "evaluation on $\{0, \infty\}^n$" and "taking coefficients in the monomial basis" are the same operation.

General individual-degree-d polynomials. For the protocol below, let $f(Y_1, \ldots, Y_m)$ be a polynomial of individual degree at most d. For a mixed input $(y, b) \in \mathbb{F}^k \times \{0, \infty\}^{m-k}$, we interpret $f(y, b)$ by repeated univariate specialization in the last $m - k$ variables. A coordinate $b_j = 0$ means evaluation at $Y_j = 0$, while $b_j = \infty$ means taking the coefficient of Y_j^d. On the monomial basis $\prod_j Y_j^{a_j}$, these operations act independently in each coordinate, so they commute and the resulting value is well defined.

3.1 The projective sum-check protocol

The projective machinery above yields a variant of the sum-check protocol in which the interpolating set is $\{0, \infty\}$ instead of $\{0, 1\}$. The protocol is shown in Figure 3.

The only structural differences from the Boolean protocol (Figure 2) are:

1. The round sums range over $\{0, \infty\}^{n-i}$ instead of $\{0, 1\}^{n-i}$.
2. The verifier's consistency check uses $s_i(0) + s_i(\infty) = C_{i-1}$ (the projective round identity) instead of $s_i(0) + s_i(1) = C_{i-1}$.

The prover's evaluation points $\widehat{U}_d = \{\infty\} \cup \{1, \ldots, d-1\}$ and the binding step $C_i = s_i(r_i)$ are identical. Here the indeterminate X in the round polynomial remains an affine variable, while the suffix variables range over $\{0, \infty\}$ in the projective sense described above. When g is multilinear, the binding step simplifies to $C_i = s_i(0) + r_i \cdot s_i(\infty)$ by Corollary 3.2.

Theorem 3.3 (Projective sum-check). *The projective sum-check protocol (Figure 3) applied to a polynomial g of individual degree at most d is perfectly complete. It has round-by-round soundness error at most $d/|\mathbb{F}|$. Consequently, its total soundness error is at most $dn/|\mathbb{F}|$.*

Proof. Perfect completeness follows by construction of the honest round polynomials. For round-by-round soundness we exhibit a state function in the sense of Definition 2.3. After challenges $r_1, \ldots, r_{i-1} \in \mathbb{F}$ have been fixed, write the honest residual claim as

$$T_{i-1} \coloneqq \sum_{b \in \{0,\infty\}^{n-i+1}} g(r_1, \ldots, r_{i-1}, b),$$

where the suffix variables are evaluated projectively as defined above. Define St on a partial transcript τ ending after the verifier's ith challenge r_i by

$$\mathsf{St}(\mathfrak{x}, \tau) \coloneqq \begin{cases} 1 & \text{if } C_i = T_i, \\ 0 & \text{otherwise,} \end{cases}$$

where C_i is the running claim derived from τ and T_i is the honest residual after challenges $r_1, \ldots, r_i$. On transcripts ending with a prover message, set St equal to its value on the preceding verifier-challenge transcript. We verify the three axioms. The empty transcript has $\mathsf{St} = 0$ because the input is false, so $C_0 \neq T_0$. Prover messages do not change St by definition. If a full transcript has $\mathsf{St} = 0$, then $C_n \neq T_n = g(r_1, \ldots, r_n)$, so the verifier rejects at the final oracle check.

It remains to bound the probability that a verifier challenge transitions St from 0 to 1. Suppose $\mathsf{St} = 0$ right before round i, i.e. $C_{i-1} \neq T_{i-1}$. The honest round polynomial is

$$h_i(X) \coloneqq \sum_{b \in \{0,\infty\}^{n-i}} g(r_1, \ldots, r_{i-1}, X, b) \ \in \mathbb{F}[X].$$

This is a standard univariate polynomial in X of degree at most d: fixing b and specializing each suffix coordinate projectively yields a polynomial in X alone, and summing over b preserves the degree bound. The projective sum over the ith coordinate decomposes as

$$h_i(0) + h_i(\infty) = \sum_{b \in \{0,\infty\}^{n-i}} g(r_1, \ldots, r_{i-1}, 0, b) \ + \sum_{b \in \{0,\infty\}^{n-i}} g(r_1, \ldots, r_{i-1}, \infty, b) = T_{i-1},$$

since $\{0,\infty\}^{n-i+1} = \{0,\infty\} \times \{0,\infty\}^{n-i}$.

The verifier reconstructs a degree-at-most-d polynomial $s_i \in \mathbb{F}[X]$ from the prover's message and the derived value $s_i(0) \coloneqq C_{i-1} - s_i(\infty)$. By construction, $s_i(0) + s_i(\infty) = C_{i-1} \neq T_{i-1} = h_i(0) + h_i(\infty)$, so $q_i \coloneqq s_i - h_i$ is a nonzero polynomial in $\mathbb{F}[X]$ of degree at most d. The verifier samples $r_i \in \mathbb{F}$ uniformly and sets $C_i \coloneqq s_i(r_i)$. The state transitions to 1 only if $C_i = h_i(r_i)$, i.e. $q_i(r_i) = 0$. Since q_i is a nonzero element of $\mathbb{F}[X]$ of degree at most d and r_i is uniform over $\mathbb{F}$, the Schwartz–Zippel lemma gives $\Pr[q_i(r_i) = 0] \leq d/|\mathbb{F}|$. A union bound over n rounds gives the total soundness error $dn/|\mathbb{F}|$.

4 Applications

In this section we describe several applications of the projective sum-check framework introduced in Section 3. Each application exploits the fact that switching the interpolating set from the Boolean hypercube $\{0,1\}^n$ to the infinity hypercube $\{0,\infty\}^n$ simplifies key algebraic operations.

4.1 Faster binding in linear-time sum-check

The dominant cost of the linear-time sum-check proving algorithm (Section 2.3) is the *binding* step: after each round i, the prover updates every cached evaluation using the verifier's challenge r_i. Over the Boolean hypercube, binding requires one subtraction per evaluation (Equation (7)):

$$p_k(r_i, x') = p_k(0, x') + r_i \cdot \big(p_k(1, x') - p_k(0, x')\big).$$

By Corollary 3.2, the projective variant eliminates this subtraction entirely:

$$p_k(r_i, x') = p_k(0, x') + r_i \cdot p_k(\infty, x').$$

Proposition 4.1. *For a sum-check instance involving a product of d multilinear polynomials over n variables, the projective variant saves a total of $d \cdot (2^n - 1)$ field subtractions compared to the standard Boolean-hypercube algorithm, summed across all n rounds.*

Proof. In each round i, the prover binds d multilinear polynomials at 2^{n-i} points each. The Boolean version performs one subtraction per binding; the projective version performs none. Summing over all rounds gives $d \cdot \sum_{i=1}^{n} 2^{n-i} = d \cdot (2^n - 1)$ saved subtractions.

While subtractions are cheaper than multiplications, they are not free. On 256-bit prime fields such as BN254, a subtraction requires a conditional correction for underflow and costs roughly a quarter of a multiplication in latency (Table 1). After prior optimizations have reduced the number of multiplications in sum-check [4], subtractions can still account for 10–20% of the total binding cost. Eliminating them is therefore a meaningful improvement.

4.2 Efficient evaluation of structured polynomials

Several commonly used multilinear polynomials have simpler interpolants on the infinity hypercube $\{0, \infty\}^n$ than their Boolean multilinear extensions on $\{0,1\}^n$.

Equality polynomial. The Boolean interpolant of the equality function is the usual equality polynomial $\widetilde{\mathsf{eq}}$.

$$\widetilde{\mathsf{eq}}(X, Y) = \prod_{i=1}^{n} \big(1 - X_i - Y_i + 2\, X_i \cdot Y_i\big).$$

The corresponding interpolant on the infinity hypercube is

$$\widehat{\mathsf{eq}}(X, Y) = \prod_{i=1}^{n} \big(1 + X_i \cdot Y_i\big).$$

Thus each factor drops from a quadratic with two subtractions to a single multiplication and one addition. This simplification directly benefits any sum-check instance that involves equality constraints, which includes Spartan [33], memory-checking arguments [36], and many folding schemes [24].

Comparison polynomials. The Boolean interpolant of the less-than function $\mathsf{LT}(x, y) = \mathbb{1}[\mathsf{nat}(x) < \mathsf{nat}(y)]$ for $x, y \in \{0,1\}^n$ is

$$\widetilde{\mathsf{LT}}(X, Y) = \sum_{j=1}^{n} \Big(\prod_{i=j+1}^{n} \widetilde{\mathsf{eq}}(X_i, Y_i) \Big) \cdot (1 - X_j) \cdot Y_j.$$

The corresponding interpolant on the infinity hypercube is

$$\widehat{\mathsf{LT}}(X, Y) = \sum_{j=1}^{n} Y_j \Big(\prod_{i=j+1}^{n} (1 + X_i Y_i) \Big) \Big(\prod_{i=1}^{j-1} (1 + X_i)(1 + Y_i) \Big).$$

Here the first product enforces agreement on all more-significant bits, while the second product accounts for the less-significant bits that become irrelevant once the first differing bit has been found. Compared with the Boolean formula, the decisive factor $(1 - X_j)Y_j$ becomes Y_j, and each ignored lower bit contributes a factor $(1 + X_i)(1 + Y_i)$.

Per-pair dictionary. Both simplifications above are instances of the single-element dictionary (Figure 1). For two-operand tables with interleaved variable pairs (X_i, Y_i), each pair-level factor is the product of two single-element translations. For example, $x_i \wedge \neg y_i$ is the product of the identity factor for x_i and the complement factor for y_i, giving $X_i \cdot 1 = X_i$ on $\{0, \infty\}$. Defining the shorthands

$$\Omega_i \;:=\; (1 + X_i)(1 + Y_i), \qquad E_i \;:=\; 1 + X_i Y_i,$$

the complete per-pair translation from Boolean to projective ($\{0, \infty\}$) interpolation is:

$$\begin{aligned} x_i \wedge y_i &\rightsquigarrow X_i Y_i, & x_i \wedge \neg y_i &\rightsquigarrow X_i, \\ x_i \vee y_i &\rightsquigarrow X_i + Y_i + X_i Y_i, & x_i \oplus y_i &\rightsquigarrow X_i + Y_i, \\ \mathbb{1}[x_i = y_i] &\rightsquigarrow E_i, & \mathbb{1}[x_i < y_i] &\rightsquigarrow Y_i, \\ x_i &\rightsquigarrow X_i(1 + Y_i), & y_i &\rightsquigarrow (1 + X_i)Y_i, \\ \text{ignored pair} &\rightsquigarrow \Omega_i. \end{aligned}$$

This dictionary drives all 40 Jolt lookup tables detailed in Section A.

Choice of interpolating set. The alternative projective set $\{1, \infty\}^n$ has Lagrange basis $\{1, Z_i - 1\}$ instead of $\{1, Z_i\}$. Equivalently, it is the shifted coefficient basis generated by $(Z_i - 1)$ in each variable. By Figure 1, this trades every addition for a subtraction and vice versa in the single-element dictionary: identity costs a subtraction $(Z - 1)$ instead of being free, but the ignored factor simplifies from $1 + Z$ to just Z.

At the pair level, the consequences are systematic. XOR remains multiplication-free, becoming $X_i + Y_i - 2$ instead of $X_i + Y_i$ on $\{0, \infty\}$, and the ignored pair simplifies from Ω_i to $X_i Y_i$. However, AND worsens from $X_i Y_i$ to $X_i Y_i - X_i - Y_i + 1$, picking up two extra subtractions. The binding step $p(r, x') = p(1, x') + (r - 1) \cdot p(\infty, x')$ has the same amortized cost as the $\{0, \infty\}$ variant, since the factor $(r - 1)$ is computed once per round.

Which set is optimal depends on the target field. On fields where subtraction carries extra cost relative to addition (e.g., BN254, where subtraction requires a conditional addition to handle underflow), $\{0, \infty\}$ is preferable. For fields where addition and subtraction have equal cost (e.g., MontyField31), the choice is neutral. On fields where addition is more expensive (e.g., 128-bit fields with Solinas reduction, where addition may trigger a carry chain), the shifted set $\{1, \infty\}^n$ is empirically competitive and can be slightly favorable. In our Fp128 experiments, however, the gap after delayed reduction is only a few percent in either direction. We therefore focus on $\{0, \infty\}^n$ throughout. Its basis agrees directly with the ordinary monomial-coefficient basis, whereas $\{1, \infty\}^n$ is better viewed as the shifted basis generated by $(Z_i - 1)$.

4.3 Polynomial commitment compatibility

Several recent polynomial commitment schemes, notably WHIR [2], were originally described in terms of monomial-basis coefficients. Sum-check-based proof systems, however, typically represent multilinear polynomials as evaluations over the Boolean hypercube $\{0, 1\}^n$. Bridging the two representations requires a basis conversion known as the Möbius transform (or multidimensional inclusion-exclusion), at a cost of $O(n \cdot 2^n)$ field operations [19]. Early versions of the WHIR implementation in Plonky3 [14] indeed performed this transform; later releases eliminated it by adapting the internal DFT to operate directly on evaluations, exploiting the multilinear decomposition $P(\alpha, \alpha^2, \ldots) = (1 - \alpha)\, P(0, \alpha^2, \ldots) + \alpha\, P(1, \alpha^2, \ldots)$.

Our use of the infinity hypercube $\{0, \infty\}^n$ resolves this mismatch from the theoretical side. By Proposition 3.1, the Lagrange basis over $\{0, \infty\}^n$ is identical to the standard monomial basis. This means the prover can use a single representation for both sum-check and commitment, eliminating the need for any basis conversion or custom DFT adaptation.

4.4 Recursion and IVC

The benefits of the projective variant are amplified in the recursive (or IVC) setting, where a prover must prove the correct execution of the sum-check verifier itself. In this context, all field operations are performed "natively" within the circuit, so a field addition or subtraction typically costs the same as a field multiplication.

In the standard binding formula, each binding step involves three field operations: one subtraction, one multiplication, and one addition $(p(0, x') + r \cdot (p(1, x') - p(0, x')))$. The projective formula requires only two: one multiplication and one addition $(p(0, x') + r \cdot p(\infty, x'))$. Since all field operations have equal cost in the recursive setting, reducing three operations to two saves 33% of the binding cost, compared with about 10-20% in the native setting.

5 Smaller challenges for 256-bit fields

In this section, we describe an optimization for sum-check over large prime fields of roughly 256 bits, such as the scalar fields of BN254 or BLS12-381. The key observation is that when targeting $\lambda \approx 128$ bits of security, it suffices for the sum-check challenges to be sampled from a subset of size $\approx 2^\lambda$, rather than from the full field of size $\approx 2^{256}$. The question is: which subset leads to the largest speedup?

Our answer is to sample challenges as λ-bit unsigned integers (occupying two 64-bit limbs on most architectures), interpreted as a k-limb big integer with the two *lower* limbs set to zero, and the entire number treated as already being in Montgomery form. This allows a short-circuit in the standard Montgomery multiplication algorithm, eliminating roughly half of the native multiplication operations. In our BN254 benchmark, this yields a $\mathbf{1.92}\times$ speedup for chained multiplication, close to the theoretical $2\times$ ceiling.

One caveat is that the number of usable bits must be slightly truncated (e.g., 125 bits for BN254, 127 bits for BLS12-381) to satisfy the assumptions of the Montgomery reduction algorithm on the size of intermediate products. Precisely, we define this challenge set as follows:

Definition 5.1 (Upper-limb challenge set). *Let p be a k-limb prime with limb width w, and let $\ell \le k$ be the number of zeroed low limbs. Define*

$$S \;:=\; \big\{\, a \in \mathbb{F}_p : \widetilde{a} \bmod 2^{w\cdot\ell} = 0 \,\big\}.$$

For BN254 with $k = 4$, $w = 64$, and $\ell = 2$, the Montgomery image of S consists of all 4-limb integers whose two least significant limbs are zero. The top 3 bits of the high limb are also zeroed to prevent overflow during big-integer multiplication, giving $|S| = 2^{128-3} = 2^{125}$. For BLS12-381, the same layout with $\ell = 2$ and 1 overflow bit gives $|S| = 2^{127}$.

Fiat-Shamir instantiation. To sample a challenge from S via Fiat-Shamir, the transcript hash is truncated to λ bits (e.g., $\lambda = 125$ for BN254), the result is interpreted as a λ-bit unsigned integer, left-shifted by $w \cdot \ell$ bits into the upper limbs of a k-limb integer, and the entire value is treated as already in Montgomery form. This deterministically maps transcript outputs to elements of S.

5.1 Review of Montgomery multiplication via CIOS

Let k be the number of limbs used to represent field elements and let w denote the maximum number of bits per limb. For BN254 in the Jolt zkVM, $k = 4$ and $w = 64$. Setting $R = 2^{w\cdot k}$, for any $a \in \mathbb{F}$, we denote its Montgomery representation by $\widetilde{a} = aR \bmod p$, where p is the prime modulus. Given $\widetilde{a}, \widetilde{b} \in \mathbb{F}$, the Coarsely Integrated Operand Scanning (CIOS) algorithm [23] gives an efficient procedure to compute $\widetilde{c} = (ab)R \bmod p$ without explicit long division. We briefly review the algorithm to illustrate the benefits of our optimization.

The algorithm proceeds in two phases. In phase one, we compute $c = \widetilde{a}\widetilde{b} = (ab)R^2$ using textbook limb-by-limb multiplication. This step uses k^2 multiplication instructions to multiply two w-bit integers.

In the next phase, we multiply c by R^{-1} and reduce modulo p to get the desired answer $\widetilde{c}$. This is done limb by limb, without ever performing explicit long division. Writing c in terms of its limbs as in Equation (8), we compute cR^{-1} by multiplying by r^{-1} (where $r = 2^w$) a total of k times sequentially.

$$c \equiv (c_{2k-1}r^{2k-1} + c_{2k-2}r^{2k-2} + \ldots + c_1 r + c_0) \bmod p \tag{8}$$

$$cr^{-1} \equiv (c_{2k-1}r^{2k-1} + c_{2k-2}r^{2k-2} + \ldots + c_1 r)r^{-1} + c_0 r^{-1} \bmod p \tag{9}$$

Observe that $c_0 r^{-1} \equiv (c_0 + m_0 p)r^{-1} \bmod p$, where m_0 is chosen so that $c_0 + m_0 p \equiv 0 \bmod r$. Solving gives $m_0 = -p^{-1}c_0 \equiv \mu c_0 \bmod r$, where $\mu = -p^{-1} \bmod r$. Since we work modulo r, it suffices to use $m_0^{(0)} = \mu_0 c_0$

discarding any carries, where μ_0 and $m_0^{(0)}$ are the least significant limbs of μ and m_0 respectively. Computing $m_0^{(0)} p$ and adding it to c gives $c^{(1)}$, which clears the least significant limb:

$$c^{(1)} r^{-1} \equiv (c_{2k-1}^{(1)} r^{2k-2} + \ldots + c_1^{(1)}) \bmod p. \tag{10}$$

The division by r in (10) is computed by a right bitshift. Repeating this subroutine k times yields:

$$cR^{-1} \equiv cr^{-k} \equiv c_{2k-1}^{(k)} r^{k-1} + \ldots + c_k^{(k)} \bmod p. \tag{11}$$

Assuming $\widetilde{a}, \widetilde{b}$ are both strictly less than p, the output is guaranteed to be less than $2p$, so a single conditional subtraction completes the computation.

The second phase uses $k+1$ multiplication instructions per reduction step. With k iterative reduction steps, the total cost of a Montgomery multiplication is $2k^2 + k$ native multiplications. For BN254, this equals $2 \cdot 16 + 4 = 36$ native multiplications.

5.2 Savings from upper-limb challenges

Now suppose $\widetilde{b} \xleftarrow{\$} S$, where S is the set of field elements whose Montgomery representation has the two least significant limbs equal to zero. This gives two sources of savings:

1. **Phase 1 (multiplication):** Since the lower two limbs of $\widetilde{b}$ are zero, half of the k^2 limb-by-limb multiplications can be skipped. For $k = 4$, this saves 8 native multiplications.
2. **Phase 2 (reduction):** During the first two reduction steps, the lowest limb of the intermediate product c is already zero, so $m_0^{(0)} = m_1^{(0)} = 0$ and the corresponding multiplications $m_i^{(0)} \cdot p$ can be skipped. This saves $8 + 2 = 10$ additional native multiplications.

In total, for $k = 4$, we save 18 out of 36 native multiplications, halving the cost of field multiplication when one operand is an upper-limb challenge.

5.3 Soundness

Proposition 5.2. *Let $S \subset \mathbb{F}$ be the set of challenges described above, with $|S| \geq 2^\lambda$. Then the sum-check protocol with challenges sampled from S has round-by-round soundness error at most $d/|S| \leq d/2^\lambda$, where d is the maximum individual degree of the polynomial being summed.*

Proof. The same state-function argument as in Theorem 3.3 applies, except that the verifier samples its challenge from S rather than from all of $\mathbb{F}$. When the state is 0 (the running claim differs from the honest residual), the prover's round polynomial differs from the honest one by a nonzero univariate of degree at most d. The state transitions to 1 only if the sampled challenge is a root of this difference polynomial. By Schwartz-Zippel, a uniformly random element of S is such a root with probability at most $d/|S|$. Since $|S| \geq 2^\lambda$, this is at most $d/2^\lambda$, which is negligible for typical parameters ($d \leq 33$, $\lambda = 125$).

For BN254, we use $\lambda = 125$ (the top 3 bits of the high limb are zeroed to prevent overflow during the big-integer multiplication), giving $|S| = 2^{125}$ and a round-by-round soundness error of at most $33/2^{125} \approx 2^{-120}$.

Corollary 5.3 (Total soundness budget). *Consider a sum-check instance with n rounds in which the polynomial has individual degree at most d, with challenges sampled uniformly from S. By a union bound over all rounds, the total soundness error is at most $nd/|S| \leq nd/2^\lambda$.*

For BN254 with $n = 20$, $d = 3$ (degree-2 $\times$ eq), and $\lambda = 125$, this gives $60/2^{125} \approx 2^{-119}$. When B independent sum-check instances are run with independently sampled challenges from S, the combined soundness error is at most $B \cdot nd/2^\lambda$ by a further union bound. For example, with $B = 10$ batched instances at the above parameters, the total error is $\approx 2^{-116}$, leaving a gap of 12 bits to a 128-bit security target. This gap can be closed via grinding (Section 5.3) at negligible cost.

Grinding for tighter security margins. The remaining gap to 128-bit security can be closed by a standard proof-of-work grinding step [37]. After computing all sum-check messages, the prover finds a nonce η such that $H(\mathsf{transcript}\|\eta)$ has at least γ leading zero bits, and includes η in the proof; the verifier checks this condition before sampling challenges. Because a cheating prover must re-grind for every modified commitment, this multiplies the attack cost by 2^γ, reducing the soundness error by a factor of $2^{-\gamma}$. The honest prover pays only 2^γ hashes, performed once outside the sum-check hot loop. In the example above, $\gamma = 12$ suffices to reach 2^{-128} total error.

5.4 Empirical speedup

We benchmark two operations to evaluate the practical impact of upper-limb challenges, both over the BN254 scalar field on an Apple M4 Max:

1. **Chained multiplication:** Computing $a \cdot r_1 \cdot r_2 \cdots r_k$ where each r_i is an upper-limb challenge. This directly measures the multiplication speedup. We observe a $\mathbf{1.92}\times$ speedup compared to standard field multiplication.
2. **Projective sum-check binding:** The binding operation $p(0, x') + r \cdot p(\infty, x')$ where r is an upper-limb challenge. Compared against the same projective binding loop with a full-field challenge, the end-to-end binding time improves by $\mathbf{1.56}\times$, isolating the effect of the specialized multiplication inside the projective kernel.

These numbers are consistent with the theoretical savings: halving the 36 native multiplications produces the nearly $2\times$ gain in chained multiplication, while the smaller $1.56\times$ binding gain reflects the additions, loads, and stores that the upper-limb specialization does not change. When combined with the projective representation and compared against the Boolean/full-field baseline, Section 6 measures a $\mathbf{1.82}\times$ full-loop binding speedup.

6 Evaluation

Our evaluation aims to answer the following questions:

1. How much does the projective sum-check variant improve binding performance and full-domain table construction over the Boolean-hypercube variant?
2. How much does the upper-limb challenge optimization (Section 5) speed up field multiplication?
3. What is the combined impact on sum-check binding throughput?
4. What is the end-to-end speedup for the full sum-check prover?

Experimental setup. All benchmarks are run on a MacBook Pro with an Apple M4 Max chip (16 cores: 12 performance at up to 4.51 GHz and 4 efficiency, 64 GB unified memory). We use the Criterion [22] benchmarking framework for statistically significant microbenchmarks. All code is compiled with thin link-time optimization (LTO), which merges compilation units at link time and enables cross-crate inlining of field arithmetic. All microbenchmarks are run single-threaded on a performance core. We leave multi-architecture and multi-threaded evaluation to future work; the gains reported here are specific to this single-threaded, ARM-based setup. Because several of the smallest differences are codegen-sensitive, we also validated suspicious rows with focused reruns and assembly inspection; see Section C. Our benchmark suite is publicly available at `https://github.com/quangvdao/monomial-sumcheck-benchmarks`.

6.1 Field arithmetic microbenchmarks

We begin by establishing the relative cost of field addition, subtraction, and multiplication across fields commonly used in SNARK implementations. These ratios determine the concrete impact of the arithmetic simplifications studied below. We benchmark the following fields and implementations:

Table 1: Per-operation cost of field arithmetic (ns), measured on Apple M4 Max. Latency: serial accumulation chain (4096 ops). Throughput: register-resident batch (4–16 elements, 1024 ops total), batch size tuned per field to avoid register spilling.

	Latency (ns/op)			**Throughput (ns/op)**		
Field	add	sub	mul	add	sub	mul
MontyField31	0.71	0.69	2.84	0.17	0.18	0.50
MontyField31 Ext4	1.40	1.39	9.19	0.22	0.23	2.45
MontyField31 Ext5 (bin.)	1.40	1.38	9.74	0.24	0.25	2.78
MontyField31 Ext5 (tri.)	1.43	1.39	7.38	0.30	0.31	3.41
Fp128	1.16	0.96	5.42	0.57	0.37	1.75
BN254 $\mathbb{F}_r$	0.77	3.72	13.0	2.13	5.92	10.1
GF(2^{128})	0.46	0.45	3.04	0.05	0.05	2.21

- **MontyField31** ($p \approx 2^{31}$): a 31-bit prime stored in Montgomery form as a single `u32`. Addition and subtraction use branchless conditional reduction (add, subtract p, then `umin`). Multiplication uses a 64-bit widening multiply followed by a single-step Montgomery reduction. We benchmark BabyBear ($p = 2^{31} - 2^{27} + 1$); KoalaBear ($p = 2^{31} - 2^{24} + 1$) uses the identical `MontyField31` code path and has the same cost. Implementation from Plonky3 [31].
- **MontyField31 degree-4 and degree-5 extensions**: binomial extensions $\mathbb{F}_p[X]/(X^d - w)$ for $d \in \{4, 5\}$. Addition and subtraction are vectorized via NEON `uint32x4_t` intrinsics: all four (or four + one scalar) components are processed in a single SIMD instruction, using the same branchless `umin` reduction as the base field. Multiplication uses explicit schoolbook-style polynomial multiplication with widening NEON multiplies and per-lane Montgomery reduction. Both BabyBear and KoalaBear support degree-4 and degree-5 extensions. BabyBear uses a binomial irreducible $X^d - w$; KoalaBear uses a binomial for degree 4 and the quintic trinomial $X^5 + X^2 - 1$ for degree 5 (since no binomial exists for this field and degree). In our implementation the latter (KoalaBear) yields lower serial multiplication latency, though not better throughput.
- **Fp128** ($p = 2^{128} - 275$): a 128-bit pseudo-Mersenne prime stored in canonical form as two `u64` limbs. Addition and subtraction use 128-bit carry/borrow arithmetic with conditional correction. Multiplication uses a 2×2 schoolbook expansion to 256 bits, then Solinas-style folding: the upper limbs are reduced by multiplying with $c = 275$ and folding back.
- **BN254** $\mathbb{F}_r$: a 254-bit prime stored in Montgomery form as four `u64` limbs. Addition and subtraction use 256-bit multi-limb carry/borrow with conditional correction. Multiplication uses the CIOS variant of Montgomery multiplication with $4 \times 4 = 16$ widening 64-bit multiplies. Implementation from arkworks [12].
- **GF(2^{128})** (GHASH): the binary extension field $\mathbb{F}_2[X]/(X^{128} + X^7 + X^2 + X + 1)$, stored in a NEON `uint64x2_t` register. Addition and subtraction are both XOR (identical cost, characteristic 2). Multiplication uses carry-less polynomial multiplication (`PMULL` on AArch64) followed by Barrett reduction. Implementation from binius64 [13].

Table 1 reports per-operation latency and throughput. Latency is measured via a serial accumulation chain of 4096 operations, capturing the true per-step cost in sequential computations like sum-check binding. Throughput is measured via a register-resident batch with in-place passes (1024 operations total), isolating arithmetic cost from memory effects and capturing the peak rate when operations are independent. The batch size is tuned per field so the working set stays in registers: 16 elements for types ≤ 20 bytes (MontyField31 variants, Fp128, GF(2^{128})), and 4 for BN254 (32 bytes, GPR arithmetic).

Several patterns emerge from Table 1. First, multiplication is consistently 4–17× more expensive than the cheaper of addition or subtraction across all fields, confirming that eliminating even one multiplication from

Table 2: Binding cost comparison: Boolean hypercube vs. projective ($\{0, \infty\}$), per polynomial per round, over BN254 $\mathbb{F}_r$ with a full-field challenge.

Operation	**Boolean**	**Projective**
Multiplications per binding	1	1
Additions per binding	1	1
Subtractions per binding	1	0
Per-element latency (ns)	14.5	12.1

Table 3: Field multiplication speedup when one operand is an upper-limb challenge, over BN254 $\mathbb{F}_r$.

Metric	**Standard**	**Upper-limb**
Native multiplications	36	18
Chained multiply latency (ns)	10.9	5.66
Multiplication speedup	1.00×	**1.92**×

the binding loop yields a meaningful speedup. Second, the MontyField31 extension fields achieve nearly the same add/sub latency (≈ 1.4 ns regardless of degree), because the NEON backend processes all components in a single vector instruction; only the throughput numbers reveal the expected degree-dependent cost. Third, the prime-characteristic 128-bit and 256-bit fields exhibit an add/sub latency asymmetry caused by differing conditional-reduction costs: for BN254, subtraction is $\approx 5\times$ more expensive than addition, while for Fp128 addition is $\approx 1.2\times$ more expensive than subtraction. In GF(2^{128}), addition and subtraction are identical (both vector XOR), so replacing one with the other has no effect. The monomial-basis binding formula uses only multiplication and addition (no subtraction), which is advantageous for BN254. Note that BN254 add/sub throughput appears worse than latency; this reflects branch-misprediction penalties in conditional reduction with random inputs, which are absent from the predictable serial chain.

6.2 Projective vs. Boolean-hypercube binding

We benchmark the core binding operation, which dominates the per-round cost of the linear-time sum-check prover. The Boolean-hypercube version computes $p(r, x') = p(0, x') + r \cdot (p(1, x') - p(0, x'))$ for each x', while the projective version computes $p(r, x') = p(0, x') + r \cdot p(\infty, x')$.

Table 2 summarizes the per-binding operation count and measured latency. The projective variant eliminates one field subtraction per binding step. In the serial dependency-chain benchmark, this reduces the per-element latency from 14.5 ns to 12.1 ns, a **1.20**× speedup. This matches the operation counts: removing one BN254 subtraction from a one-multiply, one-add, one-subtract loop yields a modest but measurable arithmetic win.

6.3 Upper-limb challenge multiplication

Table 3 reports the speedup from using upper-limb challenges (Section 5.2). The measured **1.92**× chained-multiply speedup is close to the theoretical prediction of 2×, which comes from halving the native multiplication count from 36 to 18.

6.4 Full-domain table construction

The sum-check prover builds full-domain evaluation tables for the equality polynomial $\mathsf{eq}(r, \cdot)$ and the less-than polynomial $\mathsf{LT}(r, \cdot)$ via an iterative doubling procedure over 2^n entries. The projective variant changes the per-entry recurrence:

Table 4: Full-domain table construction time (ms) for $n = 20$ variables (2^{20} entries), over BN254 $\mathbb{F}_r$ and BabyBear degree-4 extension.

	BN254 $\mathbb{F}_r$		**BB Ext4**	
Table	Boolean	Projective	Boolean	Projective
EQ	22.5	11.6 (1.94×)	3.42	3.24 (1.06×)
LT	18.9	14.2 (1.33×)	3.06	2.72 (1.13×)

Table 5: In-place binding over 2^{20} coefficient pairs, BN254 $\mathbb{F}_r$. Speedups are relative to the Boolean/full-field baseline. Each Criterion iteration restores the buffer via `memcpy`; reported times subtract the copy overhead (≈ 1 ms).

	Full-field challenge	**Upper-limb challenge**
Boolean	16.8 ms	11.7 ms (**1.44**×)
Projective	14.6 ms (**1.15**×)	9.26 ms (**1.82**×)

- **EQ table.** The Boolean recurrence splits each entry e into $(e(1-r_i),\ e \cdot r_i)$, costing 1 multiplication and 1 subtraction per entry per round. The projective recurrence splits into $(e,\ e \cdot r_i)$: the left half is a free copy, saving the subtraction entirely.
- **LT table.** The Boolean recurrence computes $y = x \cdot r_i$ and $x \mathrel{+}= r_i - y$, costing 1 multiplication, 1 addition, and 1 subtraction per entry per round. The projective recurrence computes $y = x \cdot r_i$ and $x \mathrel{+}= r_i \cdot P$ where $P = \prod_{m>i}(1 + r_m)$ is a precomputed scalar, costing 1 multiplication and 1 addition (no subtraction).

Table 4 reports the construction time for $n = 20$. Over BN254, the projective EQ table is 1.94× faster, reflecting the elimination of the subtraction from a tight per-entry recurrence. The LT table achieves a 1.33× speedup. Over BabyBear Ext4, where subtraction is much cheaper (≈ 1.4 ns, vectorized via NEON), the savings are smaller but still measurable.

6.5 Combined impact

The projective variant and the upper-limb challenge optimization are complementary: the former eliminates subtractions from the binding formula and table construction, while the latter speeds up the remaining multiplication.

Table 5 reports in-place binding throughput, following the memory layout used by Jolt's `bound_poly_var_top`: a contiguous buffer of $2N$ field elements is split into halves, and each binding step updates the first half in place. Since binding is destructive, each Criterion iteration restores the buffer via `memcpy`; the table reports binding-only times after subtracting the measured copy overhead. The two optimizations are complementary: switching to the projective representation eliminates subtractions (**1.15**×), while the upper-limb challenge specialization speeds up multiplication (**1.44**×). The **1.82**× row is a direct measurement of the combined binding loop, not the product of the earlier isolated speedups. Combined, the two changes yield a **1.82**× speedup over the Boolean/full-field baseline.

6.6 End-to-end sum-check prover

We benchmark the full sum-check prover loop to measure the end-to-end impact of the projective representation. We consider two standard settings, following the benchmarks of [4]:

- **Degree-2:** prove $\sum_{x \in \{0,1\}^n} f(x) \cdot g(x) = v$ for multilinear f, g. Each round produces a degree-2 univariate, evaluated at $\{0, 1, \infty\}$. Both variants evaluate and bind in separate passes (as in a real prover, where

Table 6: End-to-end sum-check prover time (ms) for $n = 20$ variables, single-threaded. Boolean and projective use eager reduction. Delayed and delayed projective use delayed reduction, and the × eq rows additionally use split-eq from [4]. Bold marks the fastest variant within each block.

	Degree-2				**Degree-2 × eq**			
Field	Bool.	Proj.	Delayed	Del. Proj.	Bool.	Proj.	Delayed	Del. Proj.
BN254 $\mathbb{F}_r$	93.4	85.9	50.7	**46.1**	112.1	102.1	65.2	**59.4**
BN254 $\mathbb{F}_r$ + opt. chal.	68.7	65.7	41.2	**38.1**	78.4	71.4	55.9	**53.7**
Fp128	17.5	15.5	12.3	**11.1**	21.6	20.3	16.2	**15.4**
BB Ext4	12.9	12.4	11.5	**11.0**	15.8	15.0	12.3	**12.1**
BB Ext5	18.6	18.0	17.4	**15.5**	27.0	26.3	18.6	**18.4**
KoalaBear Ext5	24.7	23.6	17.5	**16.3**	30.1	27.7	20.5	**20.0**
GF(2^{128})	5.28	5.19	3.89	**3.72**	5.91	5.93	**4.90**	4.95

the verifier's challenge is derived between the two). The Boolean version incurs 4 subtractions per pair (2 for the round message, 2 for binding); the projective version trades these for 2 additions (computing $f(1) = f(0) + f(\infty)$ and $g(1) = g(0) + g(\infty)$ during the evaluation pass), with no subtractions in binding.

– **Degree-2 × eq:** prove $\sum_x f(x) \cdot g(x) \cdot \mathsf{eq}(r, x) = v$. We apply the split-eq optimization of [4]: the equality polynomial is factored into a scalar and a precomputed suffix table, so no eq table is cloned or bound per round. The resulting quotient is degree 2, evaluated at $\{1, \infty\}$ per pair, with $q(0)$ derived from the running claim. As in the degree-2 setting, the Boolean version incurs 4 subtractions per pair; the projective version trades these for 2 additions (computing $f(1)$ and $g(1)$ during the evaluation pass).

For all fields, the baseline incorporates delayed modular reduction from [4], which amortizes a single reduction per accumulator per round; we apply the same technique to the projective variant.

Table 6 separates the effect of delayed reduction from the effect of the projective basis, and includes dedicated rows for BN254 with optimized challenges and for KoalaBear Ext5. Across all fields, moving from the eager kernels to the delayed kernels gives the largest absolute improvement. The largest gains remain on the prime fields, about 1.8× on BN254 degree-2 and about 1.4× on Fp128 degree-2. For BabyBear extensions, the delayed kernels also precompute the packed constant-multiply layout for each round challenge, which reduces the binding cost. The dedicated BN254 upper-limb row confirms that the challenge restriction of Section 5 is complementary to the basis choice, lowering the degree-2 delayed projective kernel from 46.1 ms to 38.1 ms and the degree-2 × eq delayed projective kernel from 59.4 ms to 53.7 ms.

Within a fixed reduction regime, the projective basis remains most helpful on subtraction-expensive prime fields. On BN254 it improves degree-2 by about 1.09× in the eager regime and about 1.10× in the delayed regime, and it also improves the delayed degree-2 × eq kernel by about 1.10×. On Fp128 it improves degree-2 by about 1.13× eagerly and about 1.11× with delayed reduction, while degree-2 × eq still sees smaller but positive gains in both regimes. On BB Ext4, the projective basis continues to give small wins, especially in the eager kernels. On BB Ext5, projective wins in both the eager and delayed regimes, while delayed reduction still contributes most of the absolute speedup.

KoalaBear Ext5 follows the same pattern once the delayed kernels precompute the packed round-challenge layout per round, reaching 16.3 ms for degree-2 and 20.0 ms for degree-2 × eq. The earlier regression was an implementation artifact rather than a limitation of delayed reduction in that field. In GF(2^{128}), where addition and subtraction are both XOR, the basis change is effectively neutral and the remaining differences stay within a narrow noisy band.

We omit the Fp128-only $\{1, \infty\}^n$ variants from the main table for clarity. They remain competitive, but they do not change the qualitative picture. Except for the dedicated BN254 upper-limb row, these timings do not include the upper-limb challenge optimization (Section 5).

Limitations. The projective basis is not a uniform win. Its benefit depends on the relative cost of subtraction, multiplication, and code generation in the target field and kernel. Delayed reduction contributes most of the absolute speedup in every field we tested. The projective basis is most compelling on prime fields such as BN254 and Fp128, modest but consistent on the small extension fields, and essentially neutral on $\mathrm{GF}(2^{128})$. Section C records the main implementation lessons behind these field-by-field differences and the corresponding benchmark methodology. The upper-limb challenge optimization (Section 5) is specific to ≈ 256-bit prime fields and does not apply to small or binary extension fields.

Jolt integration. We have integrated the upper-limb challenge optimization into the Jolt zkVM, requiring changes to the transcript (challenge sampling) and the field multiplication routine. Integrating the projective sum-check variant requires additional changes to the polynomial representation (storing monomial coefficients instead of Boolean-hypercube evaluations), the binding routine, the lookup tables (Section A), the batched-sumcheck dummy-round machinery, and the Stage 8 Dory scaling factors; see Section B. The projective integration is ongoing.

Acknowledgments

The authors would like to thank the developers and maintainers of plonky3 [31], Jolt [1], arkworks [12], and binius64 [13] for providing the optimized field arithmetic implementations used in the benchmarks.

Disclosures

Justin Thaler is a Research Partner at a16z crypto and is an investor in various blockchain-based platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see `https://www.a16z.com/disclosures/`).

Generative AI Usage

GPT and Claude were used to assist with implementation, benchmark setup, data extraction, LaTeX formatting, and parts of the writing in the technical sections. The authors have carefully reviewed every word and every experimental result, and take full responsibility for all writing and results reported in this paper.

References

1. Jolt codebase (2025), `https://github.com/a16z/jolt`
2. Arnon, G., Chiesa, A., Fenzi, G., Yogev, E.: WHIR: Reed–solomon proximity testing with super-fast verification. Cryptology ePrint Archive, Paper 2024/1586 (2024), `https://eprint.iacr.org/2024/1586`
3. Arun, A., Setty, S., Thaler, J.: Jolt: Snarks for virtual machines via lookups. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 3–33. Springer (2024)
4. Bagad, S., Dao, Q., DeStefano, Z., Domb, Y., Thaler, J.: Speeding up sum-check proving. Cryptology ePrint Archive, Paper 2025/1117 (2025), `https://eprint.iacr.org/2025/1117`
5. Bagad, S., Domb, Y., Thaler, J.: The sum-check protocol over fields of small characteristic. Cryptology ePrint Archive, Paper 2024/1046 (2024), `https://eprint.iacr.org/2024/1046`
6. Baweja, A., Chiesa, A., Fedele, E., Fenzi, G., Mishra, P., Mopuri, T., Zitek-Estrada, A.: Time-space trade-offs for sumcheck. Cryptology ePrint Archive, Paper 2025/1473 (2025), `https://eprint.iacr.org/2025/1473`
7. Boneh, D., Chen, B.: LatticeFold: A lattice-based folding scheme and its applications to succinct proof systems. Cryptology ePrint Archive, Paper 2024/257 (2024), `https://eprint.iacr.org/2024/257`
8. Boneh, D., Chen, B.: Latticefold+: Faster, simpler, shorter lattice-based folding for succinct proof systems. Cryptology ePrint Archive (2025)
9. Canetti, R., Chen, Y., Holmgren, J., Lombardi, A., Rothblum, G.N., Rothblum, R.D., Wichs, D.: Fiat-Shamir: from practice to theory (2019)
10. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: Plonk with linear-time prover and high-degree custom gates (2023)

11. Chiesa, A., Yogev, E.: Building Cryptographic Proofs from Hash Functions (2024), https://github.com/hash-based-snargs-book
12. arkworks contributors: arkworks zksnark ecosystem (2022), https://arkworks.rs
13. contributors, T.B.: binius64: Binary field arithmetic library (2025), https://github.com/binius-zk/binius64
14. Coratger, T., Wambsgans, T., et al.: whir-p3: WHIR implementation in Plonky3. https://github.com/tcoratger/whir-p3 (2025), basis conversion removed in PRs #81, #173, #394
15. Cormode, G., Thaler, J., Yi, K.: Verifying computations with streaming interactive proofs. Proc. VLDB Endow. **5**(1), 25–36 (2011). https://doi.org/10.14778/2047485.2047488, http://www.vldb.org/pvldb/vol5/p025_grahamcormode_vldb2012.pdf
16. Dao, Q., Thaler, J.: More optimizations to sum-check proving. Cryptology ePrint Archive, Paper 2024/1210 (2024), https://eprint.iacr.org/2024/1210
17. Diamond, B.E., Posen, J.: Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784 (2023), https://eprint.iacr.org/2023/1784, https://eprint.iacr.org/2023/1784
18. Diamond, B.E., Posen, J.: Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Paper 2024/504 (2024), https://eprint.iacr.org/2024/504, https://eprint.iacr.org/2024/504
19. Fino, B.J., Algazi, V.R.: Unified matrix treatment of the fast walsh-hadamard transform. IEEE Transactions on Computers **C-25**, 1142–1146 (1976), https://api.semanticscholar.org/CorpusID:13252360
20. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. Journal of the ACM (JACM) **62**(4), 1–64 (2015)
21. Gruen, A.: Some improvements for the piop for zerocheck. Cryptology ePrint Archive (2024)
22. Heisler, B.: Criterion.rs: Statistics-driven micro-benchmarking in rust. https://github.com/criterion-rs/criterion.rs (2024)
23. Koç, c.K., Acar, T., Kaliski, B.S.: Analyzing and comparing montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996)
24. Kothapalli, A., Setty, S.: Hypernova: Recursive arguments for customizable constraint systems. In: Annual International Cryptology Conference. pp. 345–379. Springer (2024)
25. Kothapalli, A., Setty, S.: NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive, Paper 2024/1606 (2024), https://eprint.iacr.org/2024/1606
26. Liu, T., Zhang, Z., Zhang, Y., Hu, W., Zhang, Y.: Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. Journal of Cryptology **38**(2), 17 (2025)
27. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. In: FOCS (Oct 1990)
28. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985). https://doi.org/10.2307/2007970, https://doi.org/10.2307/2007970
29. Nguyen, W., Setty, S.: Neo: Lattice-based folding scheme for ccs over small fields and pay-per-bit commitments. Cryptology ePrint Archive (2025)
30. Novakovic, A., Angeris, G.: Ligerito: A small and concretely fast polynomial commitment scheme (2025)
31. Polygon Labs: Plonky3: A toolkit for polynomial iops. https://github.com/Plonky3/Plonky3 (2024)
32. Posen, J.: Perspectives on sumcheck batching. https://hackmd.io/s/HyxaupAAA (2024), accessed: 2025
33. Setty, S.: Spartan: Efficient and general-purpose zksnarks without trusted setup. In: Annual International Cryptology Conference. pp. 704–737. Springer (2020)
34. Setty, S., Thaler, J.: Twist and shout: Faster memory checking arguments via one-hot addressing and increments. Cryptology ePrint Archive, Paper 2025/105 (2025), https://eprint.iacr.org/2025/105
35. Setty, S., Thaler, J., Wahby, R.: Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552 (2023), https://eprint.iacr.org/2023/552
36. Setty, S., Thaler, J., Wahby, R.: Unlocking the lookup singularity with lasso. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 180–209. Springer (2024)
37. StarkWare: ethSTARK documentation. Cryptology ePrint Archive, Report 2021/582 (2021), https://eprint.iacr.org/2021/582
38. Succinct: Sp1 hypercube: Proving ethereum in real-time. https://blog.succinct.xyz/sp1-hypercube/ (2025)
39. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: CRYPTO (2013)
40. Thaler, J.: Proofs, arguments, and zero-knowledge. Foundations and Trends in Privacy and Security **4**(2–4), 117–660 (2022)
41. Vu, V., Setty, S., Blumberg, A.J., Walfish, M.: A hybrid architecture for verifiable computation (2013)
42. Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup (2018)
43. Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., Song, D.: Libra: Succinct zero-knowledge proofs with optimal prover computation (2019)

A Jolt Lookup Tables Over the Projective Hypercube

In this appendix we record the lookup-table polynomials used by the current Jolt codebase, and we rewrite them after changing the interpolating set from $\{0,1\}$ to $\{0,\infty\}$. The discrete lookup-table semantics do not change; only the interpolating set does.

A.1 Boolean and projective interpolation

Let $T\colon \{0,1\}^m \to \mathbb{F}$ be any lookup table. We write $\widetilde{T}$ for its standard Boolean-hypercube multilinear extension:

$$\widetilde{T}(Z_1,\dots,Z_m) = \sum_{u\in\{0,1\}^m} T(u) \prod_{i:u_i=1} Z_i \prod_{i:u_i=0} (1-Z_i).$$

We write $\widehat{T}$ for the multilinear polynomial interpolating the same discrete table on $\{0,\infty\}^m$:

$$\widehat{T}(Z_1,\dots,Z_m) = \sum_{u\in\{0,1\}^m} T(u) \prod_{i:u_i=1} Z_i.$$

By Proposition 3.1, $\widehat{T}$ is exactly the coefficient form of the table, and $\widehat{T}(\mathsf{bits}_{\mathsf{proj}}(u)) = T(u)$ for every Boolean input u.

For the two-word tables, Jolt uses interleaved variables

$$X_0, Y_0, \dots, X_{w-1}, Y_{w-1},$$

where $i = 0$ is the most significant bit pair, exactly as in the code's 'evaluate_mle' routines. We use the shorthands Ω_i and E_i and the per-pair translation dictionary from Section 4.2. For raw index bits $Z_0,\dots,Z_{2w-1}$, an ignored bit contributes a factor $1 + Z_j$.

A.2 Complete inventory

The current Jolt tree defines 40 concrete lookup-table variants. Grouped into parameterized families, they are:

- RangeCheck, RangeCheckAligned, UpperWord, LowerHalfWord, SignExtendHalfWord, HalfwordAlignment, and WordAlignment.
- And, Andn, Or, Xor, Equal, NotEqual, UnsignedLessThan, UnsignedGreaterThanEqual, LessThanEqual, SignedLessThan, SignedGreaterThanEqual, and Movsign.
- ValidUnsignedRemainder, ValidDiv0, VirtualChangeDivisor, VirtualChangeDivisorW, and MulUNoOverflow.
- Pow2, Pow2W, ShiftRightBitmask, VirtualRev8W, VirtualSRL, VirtualSRA, VirtualROTR, and VirtualROTRW.
- The family $\mathsf{VirtualXORROT}^{(\rho)}$ for $\rho \in \{16, 24, 32, 63\}$ and the family $\mathsf{VirtualXORROTW}^{(\rho)}$ for $\rho \in \{7, 8, 12, 16\}$.

A.3 Closed-form paired families

In this subsection, w denotes the operand width, and all formulas use the paired variables X_i, Y_i above.

Bitwise word tables. Each of these tables takes two w-bit words x, y and returns the w-bit integer obtained by applying a bitwise operation: And computes $x \wedge y$, Andn computes $x \wedge \neg y$, Or computes $x \vee y$, and Xor

computes $x \oplus y$. The current Jolt code implements the Boolean-hypercube MLEs

$$\begin{aligned}
\widetilde{T}_{\mathsf{And}} &= \sum_{i=0}^{w-1} 2^{w-1-i} X_i Y_i, \\
\widetilde{T}_{\mathsf{Andn}} &= \sum_{i=0}^{w-1} 2^{w-1-i} X_i (1 - Y_i), \\
\widetilde{T}_{\mathsf{Or}} &= \sum_{i=0}^{w-1} 2^{w-1-i} (X_i + Y_i - X_i Y_i), \\
\widetilde{T}_{\mathsf{Xor}} &= \sum_{i=0}^{w-1} 2^{w-1-i} \big((1 - X_i) Y_i + X_i (1 - Y_i)\big).
\end{aligned}$$

The corresponding projective interpolants are

$$\begin{aligned}
\widehat{T}_{\mathsf{And}} &= \sum_{i=0}^{w-1} 2^{w-1-i} X_i Y_i \prod_{j \neq i} \Omega_j, \\
\widehat{T}_{\mathsf{Andn}} &= \sum_{i=0}^{w-1} 2^{w-1-i} X_i \prod_{j \neq i} \Omega_j, \\
\widehat{T}_{\mathsf{Or}} &= \sum_{i=0}^{w-1} 2^{w-1-i} (X_i + Y_i + X_i Y_i) \prod_{j \neq i} \Omega_j, \\
\widehat{T}_{\mathsf{Xor}} &= \sum_{i=0}^{w-1} 2^{w-1-i} (X_i + Y_i) \prod_{j \neq i} \Omega_j.
\end{aligned}$$

The qualitative simplification is exactly the single-bit one. The local factors $X_i(1-Y_i)$ and $(1-X_i)Y_i$ become X_i and Y_i, and every bit pair that was irrelevant to a given output bit contributes a simple factor Ω_j.

Equality and order. These tables return Boolean indicators, except for Movsign. Equal and NotEqual test word equality. The remaining four test unsigned or signed ordering ($<$, $\geq$, $\leq$); note that LessThanEqual is unsigned despite the name. Movsign returns $2^w - 1$ (all bits set) if the MSB of the left operand is 1, and 0 otherwise.

Define

$$\widetilde{\mathsf{EQ}}_w(X, Y) := \prod_{i=0}^{w-1} \big(X_i Y_i + (1 - X_i)(1 - Y_i)\big), \qquad \widehat{\mathsf{EQ}}_w(X, Y) := \prod_{i=0}^{w-1} E_i.$$

Thus the equality table changes from

$$\widetilde{T}_{\mathsf{Equal}} = \widetilde{\mathsf{EQ}}_w$$

to

$$\widehat{T}_{\mathsf{Equal}} = \widehat{\mathsf{EQ}}_w.$$

The Boolean MLE of unsigned less-than in Jolt is

$$\widetilde{\mathsf{LT}}^{\mathrm{u}}_w(X, Y) := \sum_{i=0}^{w-1} (1 - X_i) Y_i \prod_{j<i} \big(X_j Y_j + (1 - X_j)(1 - Y_j)\big).$$

The projective interpolant of the same table is

$$\widehat{\mathsf{LT}}^{\mathrm{u}}_w(X, Y) := \sum_{i=0}^{w-1} Y_i \prod_{j<i} E_j \prod_{k>i} \Omega_k.$$

The remaining comparison tables are then

$$\widetilde{T}_{\mathsf{NotEqual}} = 1 - \widetilde{\mathsf{EQ}}_w, \qquad \widehat{T}_{\mathsf{NotEqual}} = \prod_{i=0}^{w-1} \Omega_i - \widehat{\mathsf{EQ}}_w,$$

$$\widetilde{T}_{\mathsf{UnsignedGreaterThanEqual}} = 1 - \widetilde{\mathsf{LT}}^{\mathrm{u}}_w, \qquad \widehat{T}_{\mathsf{UnsignedGreaterThanEqual}} = \prod_{i=0}^{w-1} \Omega_i - \widehat{\mathsf{LT}}^{\mathrm{u}}_w,$$

$$\widetilde{T}_{\mathsf{LessThanEqual}} = \widetilde{\mathsf{LT}}^{\mathrm{u}}_w + \widetilde{\mathsf{EQ}}_w, \qquad \widehat{T}_{\mathsf{LessThanEqual}} = \widehat{\mathsf{LT}}^{\mathrm{u}}_w + \widehat{\mathsf{EQ}}_w.$$

For signed less-than, Jolt's Boolean MLE is

$$\widetilde{T}_{\mathsf{SignedLessThan}} = X_0 - Y_0 + \widetilde{\mathsf{LT}}^{\mathrm{u}}_w(X, Y).$$

The unsigned-LT summation runs over all w bit indices, including the MSB. The correction $X_0 - Y_0$ adjusts for the fact that bit 0 is the sign bit: when signs differ, $X_0 - Y_0$ contributes ± 1 while the unsigned-LT prefix-equality factor at $i = 0$ cancels, producing the correct signed answer. The projective interpolant is more naturally written in semantic form as

$$\widehat{T}_{\mathsf{SignedLessThan}} = X_0 \prod_{j=1}^{w-1} \Omega_j + E_0 \sum_{i=1}^{w-1} Y_i \prod_{1 \le j < i} E_j \prod_{k > i} \Omega_k.$$

Equivalently, the first term handles the sign-mismatch case $(x_0, y_0) = (1, 0)$, and the second term handles the equal-sign case. The signed greater-than-or-equal table is the complement:

$$\widetilde{T}_{\mathsf{SignedGreaterThanEqual}} = 1 - \widetilde{T}_{\mathsf{SignedLessThan}}, \qquad \widehat{T}_{\mathsf{SignedGreaterThanEqual}} = \prod_{i=0}^{w-1} \Omega_i - \widehat{T}_{\mathsf{SignedLessThan}}.$$

Finally, Movsign depends only on the sign bit of the left operand. Its Boolean MLE is

$$\widetilde{T}_{\mathsf{Movsign}} = (2^w - 1) X_0,$$

while its projective interpolant is

$$\widehat{T}_{\mathsf{Movsign}} = (2^w - 1) X_0 (1 + Y_0) \prod_{j=1}^{w-1} \Omega_j.$$

Remainder and division guards. ValidUnsignedRemainder(r, d) returns 1 iff the divisor $d = 0$ or the remainder $r < d$ as unsigned words (canonical remainder range). ValidDiv0(d, q) returns 1 iff $d \neq 0$, or $d = 0$ and $q = 2^w - 1$ (the RISC-V "divide by zero" quotient). VirtualChangeDivisor(x, y) returns the divisor y, except at the overflow corner $(x, y) = (\mathsf{INT\_MIN}, -1)$ where the output is 1. VirtualChangeDivisorW is the word-sized (32-bit) variant. MulUNoOverflow(z) returns 1 iff the high w bits of the $2w$-bit index z are all zero.

For ValidUnsignedRemainder, Jolt interprets X as the remainder and Y as the divisor. The Boolean MLE is

$$\widetilde{T}_{\mathsf{ValidUnsignedRemainder}} = \widetilde{\mathsf{LT}}^{\mathrm{u}}_w(X, Y) + \prod_{i=0}^{w-1} (1 - Y_i),$$

and the projective interpolant is

$$\widehat{T}_{\mathsf{ValidUnsignedRemainder}} = \widehat{\mathsf{LT}}^{\mathrm{u}}_w(X, Y) + \prod_{i=0}^{w-1} (1 + X_i).$$

The second term is the divisor-zero indicator: $\prod(1 - Y_i) = 1$ iff every $y_i = 0$, and its coefficient form over each (X_i, Y_i) pair is $(1 + X_i)$ since this constraint does not involve X_i.

For ValidDiv0, Jolt interprets X as the divisor and Y as the quotient. The Boolean MLE is

$$\widetilde{T}_{\mathsf{ValidDiv0}} = 1 - \prod_{i=0}^{w-1}(1-X_i) + \prod_{i=0}^{w-1}(1-X_i)Y_i,$$

and the projective interpolant is

$$\widehat{T}_{\mathsf{ValidDiv0}} = \prod_{i=0}^{w-1}\Omega_i - \prod_{i=0}^{w-1}(1+Y_i) + \prod_{i=0}^{w-1}Y_i.$$

For VirtualChangeDivisor, Jolt returns the divisor word, except at the exceptional point $(\mathsf{INT\_MIN}, -1)$ where the output is 1. Its Boolean MLE is

$$\widetilde{T}_{\mathsf{VirtualChangeDivisor}} = \sum_{i=0}^{w-1} 2^{w-1-i}Y_i + (2-2^w)X_0\prod_{i=1}^{w-1}(1-X_i)\prod_{i=0}^{w-1}Y_i.$$

Its projective interpolant is

$$\widehat{T}_{\mathsf{VirtualChangeDivisor}} = \sum_{i=0}^{w-1} 2^{w-1-i}(1+X_i)Y_i\prod_{j\neq i}\Omega_j + (2-2^w)X_0Y_0\prod_{i=1}^{w-1}Y_i.$$

For the word-sized family VirtualChangeDivisorW, let $h := w/2$, and let the active indices be $I_W := \{h, \ldots, w-1\}$. The Boolean MLE extracted from the code is

$$\widetilde{T}_{\mathsf{VirtualChangeDivisorW}} = \sum_{i\in I_W} 2^{w-1-i}Y_i + (2^w-2^h)Y_h + (2-2^w)X_h\prod_{i=h+1}^{w-1}(1-X_i)\prod_{i=h}^{w-1}Y_i.$$

The projective interpolant is

$$\widehat{T}_{\mathsf{VirtualChangeDivisorW}} = \sum_{i\in I_W} 2^{w-1-i}(1+X_i)Y_i\prod_{j\neq i}\Omega_j + (2^w-2^h)(1+X_h)Y_h\prod_{j\neq h}\Omega_j + (2-2^w)X_hY_h\prod_{i=h+1}^{w-1}Y_i\prod_{j<h}\Omega_j.$$

Parameterized XOR-rotate families. $\mathsf{VirtualXORROT}^{(\rho)}(x,y)$ computes $\mathsf{XOR}(x,y)$ and then rotates the result right by ρ positions within the w-bit word. $\mathsf{VirtualXORROTW}^{(\rho)}$ is the halfword variant, acting on the low $h = w/2$ bits.

For the full-word family $\mathsf{VirtualXORROT}^{(\rho)}$, Jolt's Boolean MLE is

$$\widetilde{T}_{\mathsf{VirtualXORROT}^{(\rho)}} = \sum_{i=0}^{w-1} 2^{w-1-((i+\rho) \bmod w)}\big((1-X_i)Y_i + X_i(1-Y_i)\big),$$

and the projective interpolant is

$$\widehat{T}_{\mathsf{VirtualXORROT}^{(\rho)}} = \sum_{i=0}^{w-1} 2^{w-1-((i+\rho) \bmod w)}(X_i+Y_i)\prod_{j\neq i}\Omega_j.$$

For the word-sized family $\mathsf{VirtualXORROTW}^{(\rho)}$, with active indices $I_W := \{h, \ldots, w-1\}$ and $h := w/2$, the Boolean and projective formulas are

$$\widetilde{T}_{\mathsf{VirtualXORROTW}^{(\rho)}} = \sum_{i\in I_W} 2^{h-1-((i-h+\rho) \bmod h)}\big((1-X_i)Y_i + X_i(1-Y_i)\big),$$

$$\widehat{T}_{\mathsf{VirtualXORROTW}^{(\rho)}} = \sum_{i\in I_W} 2^{h-1-((i-h+\rho) \bmod h)}(X_i+Y_i)\prod_{j\neq i}\Omega_j.$$

The current concrete instances are $\rho \in \{16, 24, 32, 63\}$ for $\mathsf{VirtualXORROT}^{(\rho)}$ (matching BLAKE2b's right-rotation amounts for 64-bit words) and $\rho \in \{7, 8, 12, 16\}$ for $\mathsf{VirtualXORROTW}^{(\rho)}$ (matching BLAKE3's right-rotation amounts for 32-bit words). Both families use the same right-rotation structure: the exponent $h - 1 - ((j + \rho) \bmod h)$ (with $j = i - h$ for the halfword variant) maps input index j to output index $(j + \rho) \bmod h$.

A.4 Closed-form raw-index families

For the remaining families it is cleaner to work with the raw index bits

$$Z_0, \ldots, Z_{2w-1},$$

again in most-significant-bit-first order. We write

$$\Omega_I(Z) := \prod_{i \in I} (1 + Z_i).$$

Range, extraction, and alignment tables. These tables operate on raw $2w$-bit indices (not interleaved pairs). $\mathsf{RangeCheck}$ extracts the second w-bit word $Z_w, \ldots, Z_{2w-1}$ as an unsigned integer. $\mathsf{RangeCheckAligned}$ does the same with the LSB forced to 0 (even alignment). $\mathsf{UpperWord}$ extracts the first w-bit word $Z_0, \ldots, Z_{w-1}$. $\mathsf{LowerHalfWord}$ extracts the low h-bit half of the second word. $\mathsf{SignExtendHalfWord}$ sign-extends that h-bit half to a full w-bit two's-complement word. $\mathsf{HalfwordAlignment}$ returns 1 iff the index is even, and $\mathsf{WordAlignment}$ returns 1 iff the index is a multiple of 4.

The Boolean MLEs extracted from the code are

$$\begin{aligned}
\widetilde{T}_{\mathsf{RangeCheck}} &= \sum_{i=0}^{w-1} 2^{w-1-i} Z_{w+i}, \\
\widetilde{T}_{\mathsf{RangeCheckAligned}} &= \sum_{i=0}^{w-2} 2^{w-1-i} Z_{w+i}, \\
\widetilde{T}_{\mathsf{UpperWord}} &= \sum_{i=0}^{w-1} 2^{w-1-i} Z_i, \\
\widetilde{T}_{\mathsf{LowerHalfWord}} &= \sum_{i=0}^{h-1} 2^{h-1-i} Z_{w+h+i}, \\
\widetilde{T}_{\mathsf{SignExtendHalfWord}} &= \sum_{i=0}^{h-1} 2^{h-1-i} Z_{w+h+i} + (2^w - 2^h) Z_{w+h}, \\
\widetilde{T}_{\mathsf{HalfwordAlignment}} &= 1 - Z_{2w-1}, \\
\widetilde{T}_{\mathsf{WordAlignment}} &= (1 - Z_{2w-2})(1 - Z_{2w-1}),
\end{aligned}$$

where $h := w/2$. Their projective interpolants are

$$\begin{aligned}
\widehat{T}_{\mathsf{RangeCheck}} &= \sum_{i=0}^{w-1} 2^{w-1-i} Z_{w+i} \Omega_{[0,2w-1]\setminus\{w+i\}}(Z),\\
\widehat{T}_{\mathsf{RangeCheckAligned}} &= \sum_{i=0}^{w-2} 2^{w-1-i} Z_{w+i} \Omega_{[0,2w-1]\setminus\{w+i\}}(Z),\\
\widehat{T}_{\mathsf{UpperWord}} &= \sum_{i=0}^{w-1} 2^{w-1-i} Z_{i} \Omega_{[0,2w-1]\setminus\{i\}}(Z),\\
\widehat{T}_{\mathsf{LowerHalfWord}} &= \sum_{i=0}^{h-1} 2^{h-1-i} Z_{w+h+i} \Omega_{[0,2w-1]\setminus\{w+h+i\}}(Z),\\
\widehat{T}_{\mathsf{SignExtendHalfWord}} &= \sum_{i=0}^{h-1} 2^{h-1-i} Z_{w+h+i} \Omega_{[0,2w-1]\setminus\{w+h+i\}}(Z) + (2^w - 2^h) Z_{w+h} \Omega_{[0,2w-1]\setminus\{w+h\}}(Z),\\
\widehat{T}_{\mathsf{HalfwordAlignment}} &= \Omega_{[0,2w-2]}(Z),\\
\widehat{T}_{\mathsf{WordAlignment}} &= \Omega_{[0,2w-3]}(Z).
\end{aligned}$$

Powers of two, masks, overflow, and byte reversal. $\mathsf{Pow2}(z) = 2^{z \bmod w}$, reading the shift amount from the last $k := \log_2 w$ bits of the index. $\mathsf{Pow2W}(z) = 2^{z \bmod 32}$, using a fixed 5-bit modulus for 32-bit shifts. $\mathsf{ShiftRightBitmask}(z) = 2^w - 2^s$ where s is the same shift amount as $\mathsf{Pow2}$ (a bitmask with the top s bits set). $\mathsf{VirtualRev8W}$ reverses the byte order within each 32-bit half of a 64-bit value.

Let $k := \log_2 w$, and let $U_t := Z_{2w-1-t}$ for $t = 0, \ldots, k-1$. The Boolean MLEs for $\mathsf{Pow2}$ and $\mathsf{Pow2W}$ are

$$\widetilde{T}_{\mathsf{Pow2}} = \prod_{t=0}^{k-1} \big(1 + (2^{2^t} - 1) U_t\big), \qquad \widetilde{T}_{\mathsf{Pow2W}} = \prod_{t=0}^{4} \big(1 + (2^{2^t} - 1) Z_{2w-1-t}\big).$$

The projective interpolants are

$$\widehat{T}_{\mathsf{Pow2}} = \Omega_{[0,2w-k-1]}(Z) \prod_{t=0}^{k-1} \big(1 + 2^{2^t} U_t\big),$$

$$\widehat{T}_{\mathsf{Pow2W}} = \Omega_{[0,2w-6]}(Z) \prod_{t=0}^{4} \big(1 + 2^{2^t} Z_{2w-1-t}\big).$$

Since $\mathsf{ShiftRightBitmask}$ satisfies $T(s) = 2^w - 2^s$, its Boolean and projective forms are

$$\widetilde{T}_{\mathsf{ShiftRightBitmask}} = 2^w - \widetilde{T}_{\mathsf{Pow2}}, \qquad \widehat{T}_{\mathsf{ShiftRightBitmask}} = 2^w \Omega_{[0,2w-1]}(Z) - \widehat{T}_{\mathsf{Pow2}}.$$

The table $\mathsf{MulUNoOverflow}$ checks that the upper half of the raw $2w$-bit product is zero. Its Boolean MLE is

$$\widetilde{T}_{\mathsf{MulUNoOverflow}} = \prod_{i=0}^{w-1} (1 - Z_i),$$

and its projective interpolant is

$$\widehat{T}_{\mathsf{MulUNoOverflow}} = \Omega_{[w,2w-1]}(Z).$$

The index set is $[w, 2w-1]$ (not $[0, w-1]$) because the coefficient form $\widehat{T} = \sum_{u:u_0=\cdots=u_{w-1}=0} \prod_{i:u_i=1} Z_i$ places all its monomials in the lower-word variables $Z_w, \ldots, Z_{2w-1}$. Finally, let π be the bit permutation induced by reversing the bytes in each 32-bit half, namely $abcd : efgh \mapsto dcba : hgfe$ at the byte level. Then

$$\widetilde{T}_{\mathsf{VirtualRev8W}} = \sum_{t=0}^{63} 2^{\pi(t)} Z_t, \qquad \widehat{T}_{\mathsf{VirtualRev8W}} = \sum_{t=0}^{63} 2^{\pi(t)} Z_t \Omega_{[0,2w-1]\setminus\{t\}}(Z).$$

A.5 Recurrence-defined shift and rotate tables

The remaining virtual shift and rotate tables are most compactly described by the exact recurrences that appear in the code. $\mathsf{VirtualSRL}(x, y)$ is the logical right shift of x by the shift amount encoded in y. $\mathsf{VirtualSRA}(x, y)$ is the arithmetic right shift, filling vacated positions from the sign bit of x. $\mathsf{VirtualROTR}(x, y)$ is the right rotation of x by the encoded amount. $\mathsf{VirtualROTRW}$ is the 32-bit variant, rotating only the low $w/2$ bits. Their projective counterparts are then obtained by coefficient interpolation of the same discrete recurrence values. Moreover, efficient $O(w)$ projective recurrences can be derived from the Boolean ones using the following observation.

Lemma A.1 (Projective recurrence transformation). *Let $T_i\colon \{0,1\}^{2i} \to \mathbb{F}$ be a sequence of truth tables satisfying*

$$T_{i+1}(u', x_i, y_i) = \alpha(x_i, y_i) \cdot T_i(u') + \beta(x_i, y_i)$$

for functions $\alpha, \beta\colon \{0,1\}^2 \to \mathbb{F}$. Let $\widehat{R}_i = \sum_{u\in\{0,1\}^{2i}} T_i(u) \prod_{j:u_j=1} Z_j$ denote the projective interpolant of T_i. Then

$$\widehat{R}_{i+1} = \widehat{\alpha}(X_i, Y_i) \cdot \widehat{R}_i + \widehat{\beta}(X_i, Y_i) \cdot \prod_{j<i} \Omega_j,$$

where $\widehat{\alpha}(X, Y) = \sum_{(a,b)\in\{0,1\}^2} \alpha(a, b)\, X^a Y^b$ and likewise for $\widehat{\beta}$.

Proof. Substituting the truth-table recurrence into the definition of $\widehat{R}_{i+1}$ and separating the sums:

$$\begin{aligned}\widehat{R}_{i+1} &= \sum_{(u',x_i,y_i)} \big[\alpha(x_i, y_i)\, T_i(u') + \beta(x_i, y_i)\big] \prod Z^{u'}\, X_i^{x_i} Y_i^{y_i} \\ &= \Big(\sum_{x_i,y_i} \alpha(x_i, y_i)\, X_i^{x_i} Y_i^{y_i}\Big) \cdot \widehat{R}_i \;+\; \Big(\sum_{x_i,y_i} \beta(x_i, y_i)\, X_i^{x_i} Y_i^{y_i}\Big) \cdot \sum_{u'} \prod Z^{u'}.\end{aligned}$$

The first parenthetical is $\widehat{\alpha}(X_i, Y_i)$ by definition, the second is $\widehat{\beta}(X_i, Y_i)$, and the final factor is $\sum_{u'\in\{0,1\}^{2i}} \prod_{j:u'_j=1} Z_j = \prod_{j<i} \Omega_j$.

VirtualSRL. Define

$$R_0 := 0, \qquad R_{i+1} := R_i(1 + Y_i) + X_i Y_i \qquad (0 \le i < w).$$

Then the Boolean MLE is $\widetilde{T}_{\mathsf{VirtualSRL}} = R_w$. Applying Lemma A.1 with $\alpha(x_i, y_i) = 1 + y_i$ and $\beta(x_i, y_i) = x_i y_i$ gives $\widehat{\alpha} = (1 + X_i)(1 + 2Y_i)$ and $\widehat{\beta} = X_i Y_i$. The projective recurrence is therefore

$$\widehat{R}_0 := 0, \qquad \widehat{R}_{i+1} := (1 + X_i)(1 + 2Y_i)\, \widehat{R}_i + X_i Y_i \prod_{j<i} \Omega_j \qquad (0 \le i < w),$$

and the projective interpolant is $\widehat{T}_{\mathsf{VirtualSRL}} = \widehat{R}_w$.

VirtualSRA. Using the same R_i, define

$$S(X, Y) := \sum_{i=1}^{w-1} 2^i (1 - Y_i).$$

Then the Boolean MLE is $\widetilde{T}_{\mathsf{VirtualSRA}} = R_w + X_0 S(X, Y)$. The projective interpolant is $\widehat{T}_{\mathsf{VirtualSRA}} = \widehat{R}_w + \widehat{S}_w$, where $\widehat{R}_w$ is as above and

$$\widehat{S}_w \;=\; X_0(1 + Y_0) \prod_{k=1}^{w-1} (1 + X_k) \cdot \sum_{i=1}^{w-1} 2^i \prod_{\substack{k=1 \\ k\neq i}}^{w-1} (1 + Y_k)$$

is the projective interpolant of $x_0 S(x, y)$. The sum is evaluable in $O(w)$ time using prefix-suffix products of the $(1 + Y_k)$ factors.

VirtualROTR and VirtualROTRW. Define

$$P_0 := 1, \qquad A_0 := 0, \qquad B_0 := 0,$$

and for $0 \le i < w$ set

$$A_{i+1} := A_i(1+Y_i) + X_iY_i, \qquad B_{i+1} := B_i + 2^{w-1-i}X_i(1-Y_i)P_i, \qquad P_{i+1} := P_i(1+Y_i).$$

Then the Boolean MLE of VirtualROTR is $\widetilde{T}_{\mathsf{VirtualROTR}} = A_w + B_w$. The projective recurrences, derived by the same technique as Lemma A.1, are

$$\widehat{A}_0 := 0, \qquad \widehat{P}_0 := 1, \qquad \widehat{B}_0 := 0,$$

with

$$\begin{aligned}
\widehat{A}_{i+1} &:= (1+X_i)(1+2Y_i)\,\widehat{A}_i + X_iY_i \prod_{j<i} \Omega_j, \\
\widehat{P}_{i+1} &:= (1+X_i)(1+2Y_i)\,\widehat{P}_i, \\
\widehat{B}_{i+1} &:= \Omega_i\,\widehat{B}_i + 2^{w-1-i}X_i\,\widehat{P}_i,
\end{aligned}$$

and the projective interpolant is $\widehat{T}_{\mathsf{VirtualROTR}} = \widehat{A}_w + \widehat{B}_w$. The $\widehat{B}$ recurrence follows because the truth-table identity $T^B_{i+1}(u', x_i, y_i) = T^B_i(u') + 2^{w-1-i}x_i(1-y_i)T^P_i(u')$ yields coefficient $\widehat{\alpha}_{BB} = \Omega_i$ for the $\widehat{B}_i$ term (since T^B_i has constant multiplier 1) and coefficient $2^{w-1-i}X_i$ for the $\widehat{P}_i$ term (since $x_i(1-y_i)$ has projective interpolant X_i).

For VirtualROTRW, the same projective recurrence is run on the active index set $I_W := \{h, \ldots, w-1\}$, where $h := w/2$, initializing $(\widehat{A}, \widehat{P}, \widehat{B}) = (0, 1, 0)$ at $i = h$. Write $\widehat{A}_{I_W}$, $\widehat{B}_{I_W}$ for the resulting accumulators. Then the Boolean MLE is $\widetilde{T}_{\mathsf{VirtualROTRW}} = A_{I_W} + B_{I_W}$, and the projective interpolant is $\widehat{T}_{\mathsf{VirtualROTRW}} = (\widehat{A}_{I_W} + \widehat{B}_{I_W}) \prod_{j<h} \Omega_j$, where the trailing product accounts for the ignored variable pairs.

A.6 Exact prefix-suffix decompositions

The linear *combine* map used by Jolt does not change when we replace $\{0, 1\}$ by $\{0, \infty\}$. What changes is only the interpolation used for each primitive prefix and suffix polynomial. Below, $P_\star$ denotes the corresponding prefix evaluation and $S_\star$ denotes the corresponding suffix evaluation.

- And, Andn, Or, and Xor use suffixes [One, And], [One, NotAnd], [One, Or], and [One, Xor], with combine formulas $P_{\mathsf{And}}S_{\mathsf{One}} + S_{\mathsf{And}}$, $P_{\mathsf{Andn}}S_{\mathsf{One}} + S_{\mathsf{NotAnd}}$, $P_{\mathsf{Or}}S_{\mathsf{One}} + S_{\mathsf{Or}}$, and $P_{\mathsf{Xor}}S_{\mathsf{One}} + S_{\mathsf{Xor}}$.
- Equal uses suffixes [Eq] and combine formula $P_{\mathsf{Eq}}S_{\mathsf{Eq}}$.
- NotEqual uses suffixes [One, Eq] and combine formula $S_{\mathsf{One}} - P_{\mathsf{Eq}}S_{\mathsf{Eq}}$.
- UnsignedLessThan uses suffixes [One, LessThan] and combine formula $P_{\mathsf{LessThan}}S_{\mathsf{One}} + P_{\mathsf{Eq}}S_{\mathsf{LessThan}}$.
- UnsignedGreaterThanEqual uses suffixes [One, LessThan] and combine formula $S_{\mathsf{One}} - P_{\mathsf{LessThan}}S_{\mathsf{One}} - P_{\mathsf{Eq}}S_{\mathsf{LessThan}}$.
- LessThanEqual uses suffixes [One, LessThan, Eq] and combine formula $P_{\mathsf{LessThan}}S_{\mathsf{One}} + P_{\mathsf{Eq}}S_{\mathsf{LessThan}} + P_{\mathsf{Eq}}S_{\mathsf{Eq}}$.
- SignedLessThan uses suffixes [One, LessThan] and combine

$$P_{\mathsf{LeftOperandMsb}}S_{\mathsf{One}} - P_{\mathsf{RightOperandMsb}}S_{\mathsf{One}} + P_{\mathsf{LessThan}}S_{\mathsf{One}} + P_{\mathsf{Eq}}S_{\mathsf{LessThan}}.$$

- SignedGreaterThanEqual uses suffixes [One, LessThan] and combine

$$S_{\mathsf{One}} + P_{\mathsf{RightOperandMsb}}S_{\mathsf{One}} - P_{\mathsf{LeftOperandMsb}}S_{\mathsf{One}} - P_{\mathsf{LessThan}}S_{\mathsf{One}} - P_{\mathsf{Eq}}S_{\mathsf{LessThan}}.$$

- Movsign uses suffixes [One] and combine formula $(2^w - 1)P_{\mathsf{LeftOperandMsb}}S_{\mathsf{One}}$.

- RangeCheck uses suffixes [One, LowerWord] and combine formula $P_{\mathsf{LowerWord}} S_{\mathsf{One}} + S_{\mathsf{LowerWord}}$.
- RangeCheckAligned uses suffixes [One, LowerWord, Lsb] and combine formula $P_{\mathsf{LowerWord}} S_{\mathsf{One}} + S_{\mathsf{LowerWord}} - P_{\mathsf{Lsb}} S_{\mathsf{Lsb}}$.
- UpperWord uses suffixes [One, UpperWord] and combine formula $P_{\mathsf{UpperWord}} S_{\mathsf{One}} + S_{\mathsf{UpperWord}}$.
- LowerHalfWord uses suffixes [One, LowerHalfWord] and combine formula $P_{\mathsf{LowerHalfWord}} S_{\mathsf{One}} + S_{\mathsf{LowerHalfWord}}$.
- SignExtendHalfWord uses suffixes [One, LowerHalfWord, SignExtensionUpperHalf] and combine

$$P_{\mathsf{LowerHalfWord}} S_{\mathsf{One}} + S_{\mathsf{LowerHalfWord}} + P_{\mathsf{SignExtensionUpperHalf}} S_{\mathsf{SignExtensionUpperHalf}}.$$

- HalfwordAlignment uses suffixes [One, Lsb] and combine formula $S_{\mathsf{One}} - P_{\mathsf{Lsb}} S_{\mathsf{Lsb}}$.
- WordAlignment uses suffixes [TwoLsb] and combine formula $P_{\mathsf{TwoLsb}} S_{\mathsf{TwoLsb}}$.
- ValidUnsignedRemainder uses suffixes [One, LessThan, RightOperandIsZero] and combine

$$P_{\mathsf{RightOperandIsZero}} S_{\mathsf{RightOperandIsZero}} + P_{\mathsf{LessThan}} S_{\mathsf{One}} + P_{\mathsf{Eq}} S_{\mathsf{LessThan}}.$$

- ValidDiv0 uses suffixes [One, LeftOperandIsZero, DivByZero] and combine

$$S_{\mathsf{One}} - P_{\mathsf{LeftOperandIsZero}} S_{\mathsf{LeftOperandIsZero}} + P_{\mathsf{DivByZero}} S_{\mathsf{DivByZero}}.$$

- VirtualChangeDivisor uses suffixes [One, RightOperand, ChangeDivisor] and combine

$$P_{\mathsf{RightOperand}} S_{\mathsf{One}} + S_{\mathsf{RightOperand}} + P_{\mathsf{ChangeDivisor}} S_{\mathsf{ChangeDivisor}}.$$

- VirtualChangeDivisorW uses suffixes [One, RightOperandW, ChangeDivisorW, SignExtensionRightOperand] and combine

$$P_{\mathsf{RightOperandW}} S_{\mathsf{One}} + S_{\mathsf{RightOperandW}} + P_{\mathsf{ChangeDivisorW}} S_{\mathsf{ChangeDivisorW}} + P_{\mathsf{SignExtensionRightOperand}} S_{\mathsf{SignExtensionRightOperand}}.$$

- MulUNoOverflow uses suffixes [OverflowBitsZero] and combine formula $P_{\mathsf{OverflowBitsZero}} S_{\mathsf{OverflowBitsZero}}$.
- Pow2 and Pow2W use suffixes [Pow2] and [Pow2W], with combine formulas $P_{\mathsf{Pow2}} S_{\mathsf{Pow2}}$ and $P_{\mathsf{Pow2W}} S_{\mathsf{Pow2W}}$.
- ShiftRightBitmask uses suffixes [One, Pow2] and combine formula $2^w S_{\mathsf{One}} - P_{\mathsf{Pow2}} S_{\mathsf{Pow2}}$.
- VirtualRev8W uses suffixes [One, Rev8W] and combine formula $P_{\mathsf{Rev8W}} S_{\mathsf{One}} + S_{\mathsf{Rev8W}}$.
- VirtualSRL uses suffixes [RightShift, RightShiftHelper] and combine $P_{\mathsf{RightShift}} S_{\mathsf{RightShiftHelper}} + S_{\mathsf{RightShift}}$.
- VirtualSRA uses suffixes [One, RightShift, RightShiftHelper, SignExtension] and combine

$$P_{\mathsf{RightShift}} S_{\mathsf{RightShiftHelper}} + S_{\mathsf{RightShift}} + P_{\mathsf{LeftOperandMsb}} S_{\mathsf{SignExtension}} + P_{\mathsf{SignExtension}} S_{\mathsf{One}}.$$

- VirtualROTR uses suffixes [RightShiftHelper, RightShift, LeftShift, One] and combine

$$P_{\mathsf{RightShift}} S_{\mathsf{RightShiftHelper}} + S_{\mathsf{RightShift}} + P_{\mathsf{LeftShiftHelper}} S_{\mathsf{LeftShift}} + P_{\mathsf{LeftShift}} S_{\mathsf{One}}.$$

- VirtualROTRW uses suffixes [RightShiftWHelper, RightShiftW, LeftShiftW, One] and combine

$$P_{\mathsf{RightShiftW}} S_{\mathsf{RightShiftWHelper}} + S_{\mathsf{RightShiftW}} + P_{\mathsf{LeftShiftWHelper}} S_{\mathsf{LeftShiftW}} + P_{\mathsf{LeftShiftW}} S_{\mathsf{One}}.$$

- VirtualXORROT$^{(\rho)}$ uses suffixes [One, XorRot$^{(\rho)}$] and combine formula $P_{\mathsf{XorRot}^{(\rho)}} S_{\mathsf{One}} + S_{\mathsf{XorRot}^{(\rho)}}$.
- VirtualXORROTW$^{(\rho)}$ uses suffixes [One, XorRotW$^{(\rho)}$] and combine formula $P_{\mathsf{XorRotW}^{(\rho)}} S_{\mathsf{One}} + S_{\mathsf{XorRotW}^{(\rho)}}$.

A.7 Takeaway

The Boolean formulas implemented in Jolt are already highly structured. Moving to $\{0, \infty\}$ preserves the same discrete tables and the same linear prefix-suffix combine maps, but it systematically replaces every Boolean factor $(1 - Z)$ by either the monomial 1 or an ignored-variable factor $(1 + Z)$. This is why equality collapses to $\prod_i (1 + X_i Y_i)$, why XOR loses its $-2X_iY_i$ term, and why the remaining families can be written exactly by adding the appropriate ignored-variable factors to the bits that do not participate in a given term.

The per-pair savings vary across suffix types. The largest gains arise in XOR-based suffixes (Xor and all VirtualXORROT variants), where the expanded Boolean factor $X_i + Y_i - 2X_iY_i$ collapses to the pure addition $X_i + Y_i$, saving one multiplication per pair. The equality suffix similarly benefits: the expanded Boolean factor $1 - X_i - Y_i + 2X_iY_i$ becomes $1 + X_iY_i$, retaining the same multiplication count but replacing two subtractions with one addition. The less-than suffix saves one multiplication per pair, as $(1 - X_j)Y_j$ becomes Y_j. The smallest gains arise in the And suffix, where the core factor X_iY_i is unchanged and the only additional cost is the ignored-pair factor $\Omega_j = (1 + X_j)(1 + Y_j)$, which is absent from the Boolean form (ignored variables sum to 1 over $\{0, 1\}$).

Not every table benefits uniformly. Single-operand extraction suffixes (e.g., LowerWord, RightOperand) acquire an extra multiplication per pair: the Boolean factor Y_i becomes $(1 + X_i)Y_i$ in the projective form. For tables like RangeCheck whose combine formula only involves such suffixes, the suffix evaluation cost increases slightly. In practice this is offset by the binding-step savings, but it means the net gain is smaller for extraction tables than for arithmetic or comparison tables.

The sub-to-add conversion is also field-dependent. In fields where addition and subtraction have symmetric cost (e.g., small Montgomery primes), the conversion is neutral. In fields where subtraction is much cheaper than addition (e.g., Fp128 with Solinas reduction), the conversion can slightly erode the gains. Conversely, in fields where subtraction is expensive (e.g., BN254, where the conditional borrow makes subtraction several times costlier than addition), the conversion provides a significant additional benefit. These suffix-level savings are on top of the generic binding speedup from Section 6.

B Beyond Lookup Tables: Integrating Projective Sum-Check into Jolt

The lookup-table rewrite of Section A is a self-contained change: it replaces the polynomial formulas inside each table while leaving the surrounding sum-check machinery untouched. A full integration of the projective variant into Jolt, however, requires changes at several other protocol layers, because the current implementation hard-codes the Boolean round identity $H(0) + H(1) = \mathsf{claim}$ in places beyond the lookup tables. This appendix catalogs the main integration surfaces.

B.1 Batched sum-check and dummy rounds

Jolt batches many sum-check instances of different sizes into a single protocol execution. The standard technique, known as "front-loaded" batching [32], runs all instances in lockstep for $n_{\max}$ rounds, where $n_{\max}$ is the number of variables of the longest instance. An instance with only $n < n_{\max}$ variables is inactive for its first $n_{\max} - n$ rounds; during those rounds, its polynomial is constant and contributes no information.

The Boolean dummy-round convention. In the current implementation, each inactive ("dummy") round sends the constant univariate $H(X) = \mathsf{claim}/2$, satisfying the round check $H(0) + H(1) = \mathsf{claim}$. The input claims for shorter instances are pre-scaled by $2^{n_{\max}-n}$ so that the claim halves exactly once per dummy round and arrives at the true claim when the instance becomes active. The same halving convention is used by the verifier and by the zero-knowledge (BlindFold) path.

What changes under projective sum-check. If the round identity becomes $H(0) + H(\infty) = \mathsf{claim}$ (see Section 3), then $H(X) = \mathsf{claim}/2$ no longer satisfies it: $H(0) + H(\infty) = \mathsf{claim}/2 + 0 \neq \mathsf{claim}$. The simplest projective dummy round instead sends $H(X) = \mathsf{claim}$ (a constant polynomial whose evaluation at ∞ is zero), so $H(0) + H(\infty) = \mathsf{claim} + 0 = \mathsf{claim}$. This eliminates the need for the $2^{n_{\max}-n}$ input-claim scaling entirely: the claim is preserved unchanged through every dummy round.

Proof compression. Each sum-check round proof is a compressed univariate polynomial that omits its linear coefficient, which the verifier recovers from the round identity $H(0) + H(1) = \mathsf{claim}$. In the projective variant, the verifier would instead use $H(0) + H(\infty) = \mathsf{claim}$. Since the leading coefficient of a polynomial of degree d equals $H(\infty)$, the compressed representation can omit the leading coefficient instead of the linear one, and the verifier recovers it as $\mathsf{claim} - H(0)$.

B.2 Instance-level dummy rounds

Beyond the global batching layer, several individual sum-check instances in Jolt maintain their own internal schedules of dummy rounds. These arise from the Dory commitment layout, which packs different logical dimensions (address, cycle, column) into a single flat vector, creating gaps where certain variables do not affect the polynomial.

Advice claim reduction (Stage 6). This instance reduces advice-polynomial claims and operates over both "cycle" and "address" variables. Within the cycle phase, some variables correspond to a gap between the column and row dimensions of the Dory matrix. During these gap rounds the instance sends the Boolean dummy message $\mathsf{claim}/2$ and tracks an internal scaling factor 2^{-d} (where d counts the gap rounds traversed so far). Under the projective variant, these internal dummy rounds would instead send $H(X) = \mathsf{claim}$ with no scaling, just as in the global batching case.

RAF evaluation and output sum-checks (Stages 5–7). These instances handle the read-address-flag (RAF) evaluation and the output-consistency check. Both share the same pattern: an internal cycle gap arises when the address and cycle dimensions do not fill the full Dory matrix. The current code pre-scales the input claim by 2^{gap} and sends $\mathsf{claim}/2$ messages during gap rounds, then renormalizes the final opening point by splicing the gap-round challenges out of the global challenge vector. In a projective port the arithmetic pre-scaling and claim renormalization disappear: dummy rounds preserve the claim, so the input claim needs no correction. The structural extraction of gap-round challenges from the global challenge vector remains necessary, since the committed polynomial has fewer variables and the verifier must form a correctly-dimensioned opening point for the PCS.

B.3 Uni-skip and first-round proofs

Stages 1 and 2 of Jolt use a "uni-skip" optimization [4] that fuses several Boolean variables into a single higher-degree variable over a small symmetric domain $D = \{-K, \ldots, K\} \subset \mathbb{F}$. Instead of running $\log |D|$ sum-check rounds for these variables, the prover sends a single polynomial s_1 of degree $O(|D|)$, and the verifier checks that $\sum_{t \in D} s_1(t) = \mathsf{claim}$. Because D is symmetric, odd power sums vanish ($\sum_{t \in D} t^j = 0$ for odd j), so odd-indexed coefficients of s_1 cannot be recovered from the domain-sum identity. The current implementation omits the linear coefficient in regular batched sum-check rounds (recovering it from $H(0) + H(1) = \mathsf{claim}$), but this convention fails for uni-skip, so the uni-skip proof currently sends all polynomial coefficients. Under the projective variant, the regular rounds would instead omit the leading coefficient and recover it from $H(0) + H(\infty) = \mathsf{claim}$. The same idea applies to uni-skip: omitting the constant coefficient c_0 (whose power sum $S_0 = |D|$ is always nonzero) lets the verifier recover it from the domain-sum identity. This would compress each uni-skip proof by one field element.

The uni-skip domain D is independent of the underlying interpolating set ($\{0,1\}^n$ or $\{0,\infty\}^n$): it is chosen for computational efficiency (small symmetric integers), not tied to the hypercube. A projective port therefore requires no changes to the uni-skip protocol itself, aside from the optional compression above. The multilinear evaluations that feed into the first-round polynomial will naturally use the monomial-coefficient representation, but the domain and verification check stay the same.

B.4 Stage 8 Dory opening (PCS batching)

After the sum-check phases are complete, Stage 8 collects all polynomial opening claims and batches them into a single Dory opening proof. Not all claims are used directly: some are first multiplied by correction factors that account for the zero-padding and sub-matrix embedding in the Dory commitment layout.

Dense polynomial padding. The dense committed polynomials (RamInc, RdInc) are shorter than the full Dory matrix and are zero-padded to fill it. Under Boolean interpolation, the multilinear extension of a zero-padded polynomial acquires a factor of $\widetilde{\mathsf{eq}}(r, \mathbf{0}) = \prod_i (1 - r_i)$ for each extra variable, because the Lagrange basis polynomial for the vertex 0 on $\{0, 1\}$ is $(1 - Z_i)$. Under projective interpolation, the Lagrange basis for vertex 0 on $\{0, \infty\}$ is the constant 1 (by Proposition 3.1, the coefficient indexed by $\emptyset$ is $a_\emptyset$). The zero-selector factor therefore becomes trivially 1, and no scaling is needed.

Advice embedding. The advice polynomials (trusted and untrusted) are committed in a smaller sub-matrix that occupies the top-left block of the main Dory matrix. The Boolean correction factor multiplies $\prod_i (1 - r_i)$ over the address variables that do not appear in the advice polynomial, for the same reason as above. Under projective interpolation, these factors also become 1: the coefficient form of a polynomial that does not depend on a variable simply contains no monomials involving that variable, so no correction is needed to account for the embedding.

Separation of batching and scaling. The current implementation maintains a clean separation: the joint commitment uses only the γ-power batching coefficients (one per polynomial), while the scaling factors are applied only to the claims. Under projective interpolation, the dense-padding and advice-embedding scaling factors become trivial, simplifying this step.

B.5 Summary

Integrating projective sum-check into Jolt requires coordinated changes to the prover, verifier, proof compression, and (in the zero-knowledge setting) the BlindFold witness construction. However, the required changes are systematic: each integration point replaces a Boolean round identity ($H(0) + H(1) = \mathsf{claim}$, claim halving, $\prod(1 - r_i)$ selectors) with its projective counterpart ($H(0) + H(\infty) = \mathsf{claim}$, claim preservation, trivial selectors). In every case we examined, the projective variant is at least as simple as the Boolean one, and often simpler. Dummy rounds no longer require claim scaling or rescaling (Sections B.1 and B.2). The zero-padding and sub-matrix embedding factors that currently thread through Stage 8 become trivially 1 (Section B.4). Uni-skip proofs remain unchanged and even gain a one-element compression opportunity. The lookup-table rewrite of Section A is a necessary first step, but the full integration touches fewer invariants than the Boolean version requires to maintain.

C Sum-check implementation notes

The speedups reported in Section 6 reflect both algebraic simplifications and implementation-level choices. For several hot kernels, semantically equivalent Rust source produced measurably different machine code on our target (Apple M4 Max, LLVM 18, `aarch64-apple-darwin`). This appendix records the main engineering lessons so that the reader can separate algebraic wins from code-generation wins.

C.1 Benchmark methodology

All end-to-end results in Table 6 were collected with Criterion [22] under Cargo's `bench` profile, which enables thin link-time optimization. Using `cargo bench` rather than an ordinary release build was significant: thin LTO changed the degree of cross-crate inlining in the field arithmetic, shifting some rows by several percent.

Long benchmark sweeps on the M4 Max were sensitive to thermal drift and run ordering. To control for this, we used a five-second warm-up, a ten-second measurement window per variant, and randomized the order of variants within each field block. Rows whose confidence intervals overlapped were validated with focused reruns that compared only the relevant variants within a single thermal window. These focused reruns were particularly important for Fp128 and GF(2^{128}), where a noisy full sweep could blur a modest gain or create a spurious regression.

C.2 Source-level code shaping

The projective identities remove subtractions algebraically, but realizing the full speedup required shaping the Rust source so that LLVM kept the intended values in registers. Three patterns recurred across the hot kernels.

Explicit temporaries. Naming intermediate sums $s_f = f_0 + f_\infty$, $s_g = g_0 + g_\infty$ in separate `let` bindings, rather than writing $(f_0 + f_\infty)(g_0 + g_\infty)$ inline, consistently shortened live ranges and reduced stack spills on the M4/NEON backend.

Asymmetric inlining. For the packed BabyBear Ext5 accumulation, two variants of the same arithmetic fragment were maintained: one marked `#[inline(always)]` (used for the $q(\infty)$ path) and one marked `#[inline(never)]` (used for the $q(1)$ path). This kept the hottest loop body compact while the less critical path paid a function-call overhead that was cheaper than the spills it avoided.

Intentional reloads. In the delayed projective BB Ext5 kernel, reloading the packed eq weights for the $q(1)$ path rather than extending their live range across the $q(\infty)$ accumulation reduced total stack traffic, despite the redundant memory access.

None of these changes altered the arithmetic operation counts; their impact came entirely from compiled function size, register pressure, and stack traffic.

C.3 Case study: BB Ext5 eager projective kernel

The clearest example arose in the eager degree-2 $\times$ eq projective kernel over BB Ext5. An early packed NEON implementation was algebraically correct but slightly slower than the Boolean baseline. Assembly inspection revealed the cause: the projective kernel compiled to a larger function, used a deeper stack frame, and issued substantially more stack references around the packed quintic multiply-and-accumulate.

Reordering the loop body to form the temporary sums (s_f, s_g) before the packed $q(\infty)$ accumulation reduced this spill pressure. The compiled function shrank from 5 996 to 5 436 bytes, the stack frame dropped from `0x530` to `0x410` bytes, and the total stack-reference count fell from 385 to 248. After this single source-level change, the projective row again outperformed the Boolean row.

A related issue appeared in the delayed BB Ext5 projective kernel, where the asymmetric-inlining and intentional-reload patterns from Section C.2 were jointly necessary to recover the expected gain. Together, these two kernels illustrate how an algebraically neutral source-level change can yield a measurable end-to-end speedup when the compiled code is register-pressure limited.

C.4 Cross-field observations

The same implementation discipline mattered to different degrees across fields. On BN254 and Fp128, the projective basis retains a genuine algebraic advantage because subtraction is nontrivial and lower-order terms constitute a meaningful share of the total prover cost. Focused reruns confirmed that Fp128 benefits from the projective basis in both the eager and delayed degree-2 $\times$ eq kernels, even when a noisy full sweep temporarily suggested otherwise. KoalaBear Ext5 required a similar adjustment: the delayed kernels only matched the eager path after precomputing the packed round-challenge layout once per round and reusing it during binding. On $\mathrm{GF}(2^{128})$, addition and subtraction are both XOR, so the basis change is algebraically neutral; the remaining differences reflect ordinary measurement and code-generation noise. The $\mathrm{GF}(2^{128})$ results therefore serve as a control: careful code shaping can remove accidental regressions, but it cannot produce an algebraic win when the underlying field operations offer none.