

Speeding Up Sum-Check Proving

Quang Dao
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
qvd@andrew.cmu.edu

Zachary DeStefano
New York University
New York, New York, USA
zd2131@nyu.edu

Suyash Bagad
Shunya Labs
Amsterdam, Netherlands
suyash@ingonyama.com

Yuval Domb
Ingonyama
Petach Tikvah, Israel
yuval@ingonyama.com

Justin Thaler
Georgetown University and a16z
crypto research
Washington DC, USA
justin.r.thaler@gmail.com

Abstract

The sum-check protocol is a foundational primitive in modern cryptographic proof systems, but its prover-side cost has emerged as a concrete bottleneck. This paper introduces three complementary techniques that reduce sum-check proving time and memory, especially in zero-knowledge virtual machines (zkVMs).

First, for products of many multilinear polynomials, we give a new algorithm that significantly reduces the field multiplications required for proving. Second, our “small-value sum-check prover” speeds up the common setting where the polynomials being summed evaluate to 32- or 64-bit integers or elements of a small subfield of a larger extension field. Even outside the small-value setting, the same algorithm yields a faster “streaming prover,” a small-space prover applicable whenever the summands can be enumerated in small space, as in zkVMs. Third, in the ubiquitous case where one factor is an equality polynomial, we nearly eliminate its prover overhead by exploiting its decomposable tensor structure.

We implement these techniques in Jolt, a state-of-the-art zkVM, and evaluate their performance. In Jolt, they yield over an order-of-magnitude runtime speedup and memory reduction for the Spartan subprotocol, and 1.7×–2.2× speedups for a key high-degree sum-check subprotocol in the Shout batch-evaluation argument.

CCS Concepts

• Security and privacy → Cryptography; • Theory of computation → Interactive proof systems.

Keywords

sum-check, SNARKs, zkVMs, multilinear polynomials, finite-field arithmetic

ACM Reference Format:

Quang Dao, Zachary DeStefano, Suyash Bagad, Yuval Domb, and Justin Thaler. 2026. Speeding Up Sum-Check Proving. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3846581>



This work is licensed under a Creative Commons Attribution 4.0 International License. CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11

<https://doi.org/10.1145/3830454.3846581>

1 Introduction

The sum-check protocol [35] 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

$$\sum_{x \in \{0,1\}^\ell} g(x) = t,$$

where g is a multivariate polynomial of individual degree d over a finite field $\mathbb{F}$. From the verifier’s perspective, the protocol reduces the expensive task of summing 2^ℓ evaluations of g to the much cheaper task of evaluating g at a single random point in $\mathbb{F}^\ell$. In many applications g factors as a product of d multilinear polynomials; this is the case we consider throughout.

When used in succinct non-interactive arguments of knowledge (SNARKs), the sum-check protocol generally *minimizes* the amount of data the SNARK prover must commit to cryptographically, and *also* minimizes the time required to prove that this committed data is well-formed. Because of these features, the sum-check protocol now underpins SNARKs for arithmetic circuit satisfiability [26, 56, 58], rank-1 constraint systems (R1CS) [42], Plonkish and customizable constraint systems (CCS) [14, 44], the fastest lookup arguments [45, 46], polynomial commitment schemes [5, 23, 24, 39], folding schemes [11, 12, 31, 32, 38], and zero-knowledge virtual machines (zkVMs) like Jolt [6], SP1 Hypercube [48], and Ceno [34].

The sum-check protocol is so effective at reducing cryptographic costs that its own proving time and memory have become a major bottleneck in many proof systems.

Overhead of large-field arithmetic. The security of the sum-check protocol depends on the size of the finite field $\mathbb{F}$ over which it operates. For non-interactive security, the sum-check protocol needs to operate over fields of at least $\lambda = 128$ bits, with many systems (such as Jolt) requiring 256-bit fields to support elliptic curve-based polynomial commitments. This leads to a large overhead: a single 256-bit field multiplication requires 45–100 CPU cycles [59]. See the full version for more details on the necessity of large fields.

Time-space tradeoffs for sum-check algorithms. Let $M = 2^\ell$ be the size of the sum being proved. The fastest proving algorithms for products of multilinear polynomials [51, 54] perform a number of field operations that scales linearly with M ; however, these algorithms also require linear space (i.e., storing $\Theta(M)$ field elements). In practice, when M can be on the order of billions, the memory requirements of these algorithms can be prohibitive.

There are low-memory alternatives, called streaming algorithms, which use significantly less space at the cost of superlinear time. The best streaming algorithm, recently introduced by Baweja et al. [10], uses $O(M^{1/k})$ space and $O(M \log \log M)$ time for any constant k whenever d is a small constant (i.e., 3 or smaller). However, as d increases, this algorithm rapidly approaches the performance of a prior streaming algorithm [19] that costs $O(d^2 M \log M)$ time and $O(d + \log M)$ space. For realistic values of M , this is more than an order of magnitude slower than the linear-time algorithm.

To summarize, we are considering applications of the sum-check protocol to a low-degree function (often a product) of d multilinear polynomials over a λ -bit field $\mathbb{F}$. In a crude field-operation cost model, the linear-time algorithm of [51, 54] is essentially optimal for small d such as $d = 2$ or $d = 3$. In practice, however, λ -bit field multiplications dominate proving time and should not be regarded as unit cost; on top of this, moving to a sublinear-space prover incurs further overhead. Handling high d also matters: the Shout [46] batch-evaluation argument used in Jolt [2, 6], as currently configured, invokes sum-checks with d up to 32.

A major focus of our work is to speed up the Jolt zkVM prover [2, 6], a state-of-the-art zkVM that relies entirely on various sum-checks for its information-theoretic core. On large traces (e.g., 1 billion cycles), the Jolt prover spends about 65-70% of its time on sum-checks, making them a major bottleneck. While this is the main motivation, our techniques apply broadly to sum-check-based SNARKs, including Spartan [1, 42] and its variants such as Binus64 [50].

An earlier version of this manuscript appeared in prior work [7]. This paper reorganizes and expands that presentation, and directly combines and improves upon results from prior work [8, 21]. The public full version contains complete algorithm descriptions, proofs, implementation details, and additional evaluation.

1.1 Our results

Our main algorithmic contribution is an evaluation-form *round batching* prover. It keeps the first, most expensive sum-check rounds in the small-value regime by delaying the verifier's bindings and computing one multivariate prefix polynomial for several rounds at once. The remaining contributions make this primitive efficient for high-degree products, low-memory proving, and equality polynomials.

Except for univariate skip (Section 7), all techniques leave the protocol messages, verifier, proof size, and soundness analysis unchanged. Univariate skip, introduced by Gruen [28], preserves soundness but changes the first prover message. Table 1 gives the exact impact for every technique.

Round-batched prover algorithms. In Section 5, we combine the techniques below with an algorithmic change that improves prover performance across the time-space tradeoff. Prior streaming algorithms [10, 19] already group sum-check rounds into epochs, called *windows*, and amortize the cost of computing prover messages within a window. In a window of v rounds, our prover instead computes the single prefix polynomial

$$q(X_1, \dots, X_v) = \sum_{x' \in \{0,1\}^{\ell-v}} \prod_{k=1}^d p_k(X_1, \dots, X_v, x')$$

Table 1: Impact of our techniques on verifier work and proof size. In Jolt, univariate skip replaces four cubic sum-check rounds (12 field elements) with one degree-27 round (28 elements), then adds one claim linking the new round to the remaining rounds, for a net increase of 17 field elements.

Technique	Protocol impact
All other techniques	Verifier and proof unchanged
Univariate skip (Jolt)	One degree-27 check replaces four cubic checks; +17 $\mathbb{F}$ (544 B)

Table 2: Generic-setting prover asymptotic costs ($M=2^\ell$), counted in terms of bb (i.e., big-big) field multiplications. In all cases, the constants hidden by the asymptotics are concretely small for practical parameters.

Algorithm	Time	Space
LinearTime [51, 54]	$d^2 M$	dM
OptLinearTime	$(d \log d) M$	dM
LogSpace [19]	$d^2 M \log M$	$d + \log M$
CrossProduct $_k$ [10]	$d^2 M(k + \log \log M)$	$M^{1/k}$
EvalProductStream $_k$	$(d \log d) M(k + \log \log M)$	$M^{1/k}$

on the evaluation grid U_d^v . The next v sum-check messages are then obtained from this grid after receiving the verifier's challenges.

This computation is a multivariate instance of the high-degree product problem in Section 4. The same Toom-Cook recursion [17, 53] lets us store $(d+1)^v$ evaluations per window, rather than the $O((2^d)^v)$ coefficient-domain terms used by prior work [10]. Table 2 compares the resulting provers with prior work.

The same round batching primitive has two uses. In the *small-value setting*, applying it once at the start of the protocol keeps the first few rounds in small-small and small-big arithmetic. In the *streaming setting*, applying it repeatedly with a memory-bounded window schedule improves the per-pass degree dependence from $\Theta(d^2)$ in prior work [10] to $\Theta(d \log d)$.

Faster field arithmetic via small-value specialization and delayed reduction. In Section 3, we observe that while computing the product of two arbitrary finite field elements is expensive, when one or both elements are small interpreted as integers (e.g., storable in a single 64-bit or 32-bit machine word), it is possible to compute their product significantly faster. We then generalize these techniques to the efficient computation of linear combinations (and, more generally, low multiplicative depth circuits) over the appropriate finite field. Our techniques improve efficiency by allowing intermediate values to exceed the field size, deferring modular reduction until the end of the operation sequence.¹

We provide explicit procedures for small-small, small-big, and big-big multiplications, where the latter refers to standard finite field multiplication.

¹While the delayed reduction technique is not new and has been used in other contexts, our work is the first time it has been applied to speed up sum-check proving, yielding a substantial speedup for all existing sum-check algorithms (Section 3).

High-degree sum-check speedups. In Section 4, we apply this observation to the problem of efficiently computing the product of d multilinear polynomials, each in v variables. This computation is a core subroutine in all known sum-check proving algorithms and dominates their runtimes.

The most concretely efficient prior baseline is the naive algorithm that performs $O(d^{v+1})$ (big-big) multiplications to compute this product by directly evaluating the product at $(d+1)^v$ points. We lower this cost to $O(d^v + d \log d)$ (big-big) multiplications with a concretely small constant. In the univariate case of $v = 1$, this is an improvement from $O(d^2)$ to $O(d \log d)$ multiplications, while for $v > 1$, this is an improvement by a factor of d over the naive algorithm.² For large d used in practice, such as $d = 16$ or $d = 32$, this yields substantial concrete speedups: up to about $2.5\times$ in our high-degree kernel microbenchmarks, and about $1.7\times$ – $2.2\times$ in Jolt’s read-access (RA) virtualization sum-check benchmark (see §9).

This approach is inspired by a suggestion of Ben Diamond [22]; however, making the idea concretely efficient requires a routine to efficiently extrapolate from a small set of evaluation points to a larger set. Our contributions here consist of generalizing the approach to the multivariate setting and making it concretely efficient, by optimizing the extrapolation step with a mix of specialized field operations and optimal multi-addition chains.

Equality-polynomial specialization. In practice, some of the multilinear polynomials in the sum-check protocol have a specific *decomposable* structure. That is, they can be expressed as a product of univariate polynomials. One common example is the equality polynomial

$$\tilde{eq}(W, X) = \prod_{i=1}^{\ell} (W_i \cdot X_i + (1 - W_i) \cdot (1 - X_i)),$$

with randomly chosen $W \in \mathbb{F}^\ell$, which is nearly ubiquitous in sum-check applications. Section 6 explains how to reduce the time and space added by this factor.

Gruen [28] first exploited the three-part decomposition of $\tilde{eq}$; Setty, Thaler, and Zhu [46] gave a variant that preserves the standard verifier. Our contribution splits the remaining future-round table into sublinear-size pieces and combines this representation with evaluation-form round batching. Rewriting the computation as iterated sums lets the prover reuse these pieces across rounds.

Implementation and evaluation. We evaluate our techniques through an implementation integrated into the Jolt zkVM (§8). Our experimental evaluation (§9) reports paired microbenchmarks and subprotocol ablations together with an end-to-end sweep of the optimized Jolt prover. In Jolt’s Spartan Outer benchmarks, our most optimized variant (using univariate skip, which trades faster proving for a higher-degree verifier interpolation step) reaches $10.9\times$ runtime speedup over an unoptimized baseline (with none of our optimizations in this paper), and up to $17.5\times$ lower isolated peak memory; at memory-bottleneck scales, runtime speedup grows up to $57.9\times$. In Shout’s read-access virtualization sum-check, we observe $1.7\times$ – $2.2\times$ speedups across various configurable degree settings.

²The lower-order work in sb multiplications and additions is $O(d^{v+1})$, which is dominated by the bb savings for all practical values of d .

1.2 Related work

Sum-check’s central role in modern SNARKs has spurred extensive prover-side optimizations, both general and setting-specific, which we review below.

Sum-check origins. The sum-check protocol was introduced by Lund, Fortnow, Karloff, and Nisan [35] as a key component in interactive proofs for $\#P$ and later $PSPACE$ [47], where the prover ran in superpolynomial time even on problems solvable in logarithmic space. Goldwasser, Kalai, and Rothblum (GKR) [26] later used it to construct a polynomial-time interactive proof for any arithmetic circuit of polylogarithmic depth and polynomial size.

Linear time proving. Vu et al. [54] and Thaler [51] gave linear-space implementations of the sum-check prover for products of multilinear polynomials, performing $O(M)$ field operations for fixed d , where $M = 2^\ell$ is the size of the summation domain; this is colloquially called the “linear-time” prover. Cormode, Thaler, and Yi [19] later gave a space-efficient variant running in quasilinear time and logarithmic space. We follow the conventional “linear-time” name, but we stress that the dependence on d , and the distinction between big-by-big, big-by-small, and small-by-small multiplications, both glossed over by it, are central to our work.

These algorithms have been deployed in many systems, including a long line of linear-time GKR variants [18, 51, 54–56, 58, 60, 61] and Spartan [42], the latter forming a major component of the Jolt zkVM [6]. Sum-check has also been used to build efficient memory-checking arguments, which force an untrusted prover to correctly process reads and writes to a large memory (or just reads, for read-only memories) [43, 45, 62]; the most recent work in this line is Twist and Shout [46].

Low-memory proving. Baweja et al. [10] (generalizing Chiesa et al. [15]) give a sum-check prover for products of constantly many multilinear polynomials, running in $O(M \log \log M)$ field operations and $O(M^{1/k})$ space for any constant integer $k > 1$. Under natural conjectures, they also prove a matching $\Omega(M \log \log M)$ lower bound on field multiplications for products of two multilinear in sublinear space. This lower bound in M applies to our streaming improvement as well; we improve on the asymptotic dependence on the degree instead.

Delayed reduction. Postponing modular reduction to a canonical field representation across long operation sequences (often called “lazy reduction”) is a well-known optimization in pairing-based cryptography [3, 40, 41], and is widely used in high-performance elliptic-curve and SNARK implementations such as blst [49] and gnark-crypto [16] to optimize extension-field arithmetic and multi-scalar multiplications. We systematically apply this insight to the core linear combinations and low-degree polynomial evaluations of the sum-check protocol, where it composes with small-value arithmetic to yield substantial speedups; see §3 for implementation details and costs.

Small-characteristic sum-check optimizations. Gruen [28] also optimizes the sum-check prover, focusing on fields of small characteristic. Two of his optimizations are relevant to us. The first applies when one factor in the product is a decomposable multilinear, and is complementary to ours; we discuss it in detail in Section 6. The

second is the univariate-skip technique, which targets the same small-value regime as ours but modifies the protocol and increases the degree of the first-round prover polynomial; we revisit this tradeoff in Section 7 in the context of large prime fields.

Two further works also target this setting. Liu and Zhang [33] build on a technique of Dao and Thaler [20] to give a sum-check prover for products of d Boolean-valued multilinear polynomials over binary tower fields; however, for the common cases $d \in \{2, 3\}$, their algorithm is asymptotically slower than ours, and no implementation is provided. Wei et al. [57] obtain a fast sum-check prover implementation by combining parallel repetition with Neutron-Nova's [32] "sumfold" scheme, which reduces many independent sum-check instances to a single one. In contrast, our small-value optimizations (except univariate skip) apply "as is" to existing sum-check-based SNARKs and yield functionally equivalent implementations: the SNARK verifier is unchanged.

Structured sum-check protocols. The *sparse-dense sum-check* prover algorithm [45] runs in $O(cM)$ time and $O(M^{1/c})$ space for any integer $c \geq 1$, but applies only to products of an arbitrary multilinear and a structured one. Its refinement, the *prefix-suffix inner product* algorithm [37], broadens applicability and improves efficiency. Jolt uses these where possible, but they miss many of its sum-check instances (including in Spartan). The same work [37] also gives a small-space Jolt prover (without SNARK composition or recursion) whose efficiency benefits substantially from our optimizations in this setting.

2 Preliminaries

Notation. We index vectors from 1, and denote ranges using bracket notation as: $[a, b] := \{a, \dots, b\}$, $[n] := \{1, \dots, n\}$, $[< i] := \{1, \dots, i-1\}$, and $[> i] := \{i+1, \dots, n\}$ (where n is assumed to be clear from context). $U_d := \{\infty, 0, 1, \dots, d-1\}$ and $\widehat{U}_d := U_d \setminus \{\infty\}$. We write $\text{nat}_b(x) := \sum_{i=1}^{\ell} x_i \cdot b^{i-1}$ for $x \in [0, b-1]^\ell$, abbreviating $\text{nat}(x)$ when $b = 2$.

2.1 Finite fields

SNARKs that use elliptic curve-based polynomial commitments 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, such elements use $N = 4$ limbs of 64-bit unsigned integers. To accelerate multiplication, field elements are typically stored in *Montgomery form* [36]: an element $a \in \mathbb{F}_p$ is represented as $a' = aR \pmod{p}$, where $R = 2^{64N}$ makes multiplication and division by R cheap bit shifts.

Multiplying two Montgomery-form elements a', b' takes two steps: a large-integer product $c = a' \cdot b'$ ($2N$ limbs, N^2 native multiplications), then a Montgomery reduction $cR^{-1} \pmod{p}$ returning to Montgomery form ($N^2 + N$ multiplications).³ The total is thus $\approx 2N^2 + N$ multiplications and a similar number of additions, or ≈ 45 -100 CPU cycles at $N = 4$ [59] (see the full version).

Barrett reduction [9] avoids the division by R but, at the same multiplication count, requires more conditional subtractions and is

slower in practice. Hence most field arithmetic libraries use Montgomery representation for large prime fields.

Hash-based SNARKs often use ~ 128 -bit *extension fields* $\mathbb{F}_{p^k}$, k -dimensional vector spaces over a base $\mathbb{F}_p$. Elements are represented as base-field vectors in a chosen basis. A standard *monomial basis* represents elements as polynomials of degree less than k , with multiplication performed modulo a degree- k irreducible polynomial. When $k = 2^\tau$, a *tower basis* from iterated quadratic extensions is available. For both bases, multiplication costs $O(k^{\log_2 3}) \approx O(k^{1.585})$ base-field multiplications via Karatsuba's algorithm [30]. Prominent examples include the degree-4 tower over the 31-bit BabyBear field and the degree-128 tower $\text{GF}(2^{128})$ over $\text{GF}(2)$, used in systems like Binius [23].

2.2 Multilinear polynomials

We collect facts on multilinear polynomials used in our paper; see, e.g., [52] for derivations.

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

In particular, the *equality polynomial* $\tilde{\text{eq}}$ defined as

$$\tilde{\text{eq}}(X, Y) = \prod_{i=1}^{\ell} (X_i \cdot Y_i + (1 - X_i) \cdot (1 - Y_i)) \quad (1)$$

is the multilinear extension of the equality function $(\cdot \stackrel{?}{=} \cdot) : \{0, 1\}^\ell \times \{0, 1\}^\ell \rightarrow \{0, 1\}$, i.e., $\tilde{\text{eq}}(x, y) = 1[x = y]$ on $\{0, 1\}^\ell$.

LEMMA 2.1. *Let $p : \mathbb{F}^\ell \rightarrow \mathbb{F}$ be a multilinear polynomial. Then for any $0 \leq i \leq \ell$, any $(r_1, \dots, r_i) \in \mathbb{F}^i$, and any $x' \in \{0, 1\}^{\ell-i}$,*

$$p(r_1, \dots, r_i, x') = \sum_{y \in \{0, 1\}^i} \tilde{\text{eq}}((r_1, \dots, r_i), y) \cdot p(y, x'). \quad (2)$$

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

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

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

$$p(r_1, \dots, r_\ell) = \sum_{y \in \{0, 1\}^\ell} \tilde{\text{eq}}(r_1, \dots, r_\ell, y) \cdot p(y). \quad (4)$$

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

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

We use a standard variant accommodating the "evaluation at infinity" $s(\infty) := \text{leading coefficient of } s(X)$. Given the evaluations of a degree-at-most- k polynomial on U_k , we use *extrapolation* to mean computing evaluations of the same polynomial on a larger set U_h , where $h > k$.

LEMMA 2.2. *Let $s(X) = a \cdot X^d + \dots \in \mathbb{F}[X]$ be a polynomial of degree d . Then for any distinct evaluation points $x_1, \dots, x_d \in \mathbb{F}$,*

³One can reduce the count to $N^2 + 1$ [25], though added conditional branches may negate the gain.

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

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

For each round $i = 1, \dots, \ell$:

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

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

$\mathcal{P}$ 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 $C_i := s_i(r_i)$.

After ℓ rounds, we reduce to the claim that

$$g(r_1, \dots, r_\ell) = C_\ell.$$

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

Figure 1: The sum-check protocol

setting $U = \{\infty\} \cup \{x_1, \dots, x_d\}$, we have the identity:

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

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 $k \in [d]$.

Streaming model. Streaming algorithms assume that the evaluations $p_1(x), \dots, p_d(x)$ for $x \in \{0,1\}^\ell$ can be enumerated in linear time and small space, in lexicographic order (little- or big-endian). This holds whenever $p_1, \dots, p_d$ are multilinear extensions (MLEs) of functions $f_1, \dots, f_d: \{0,1\}^\ell \rightarrow \mathbb{F}$ whose hypercube evaluations are similarly streamable (e.g., from a VM execution trace).

2.3 The sum-check protocol

In Figure 1, we describe the sum-check protocol. For g of individual degree at most d over $\mathbb{F}$, it reduces a claim of the form

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

to a single evaluation $g(r) = v$ at random $r \in \mathbb{F}^\ell$. We take $g = p_1 \cdots p_d$ for multilinear $p_1, \dots, p_d$; existing algorithms extend by linearity to any degree- d function of multilinear.

In each round j , the honest prover sends a univariate polynomial s_j of degree d . Since a degree- d univariate is determined by $d+1$ evaluations, the values $\{s_j(c) : c \in \{\infty\} \cup \{0, \dots, d-1\}\}$ specify s_j .⁴ The prover can in fact omit $s_j(0)$, letting the verifier derive it via $s_j(0) = C_{j-1} - s_j(1)$ (honest execution).

THEOREM 2.3. *The sum-check protocol applied to a polynomial g of individual degree at most d is perfectly complete and has soundness error at most $d\ell/|\mathbb{F}|$.*

See, e.g., [52] for a proof of this standard result, attributed to Lund, Fortnow, Karloff, and Nisan [35].⁵

⁴We assume an injection from $\{0, \dots, d-1\}$ into $\mathbb{F}$. Large prime fields use the natural identification; for binary extension fields, a natural choice is the elements whose basis representation matches each integer's binary form.

⁵The protocol in fact satisfies *round-by-round soundness* $d/|\mathbb{F}|$, ensuring security (in the ROM) under the Fiat-Shamir transform; see [13].

Linear-time algorithm for sum-check proving. We recall the linear-time algorithm from prior work [19, 51, 54], running in time linear in $M = 2^\ell$. The idea is to “bind” the polynomial to each verifier challenge: after round i , cache the partial evaluations $p_k(r_1, \dots, r_i, x')$ for all $k \in [d]$, $x' \in \{0,1\}^{\ell-i}$. This cache halves each round, enabling round $i+1$ in $O(d^2 \cdot 2^{\ell-i})$ time, for a total of $O(d^2 \cdot 2^\ell)$. See the full version for the full description and cost analysis of LinearTime_{SC}, along with two prior streaming algorithms [10, 19].

3 Non-black-box field arithmetic

Finite-field arithmetic is typically treated as a black box. Internally, each operation has two steps: the operands are treated as integers (or polynomials) and the corresponding ring operation is performed, yielding a non-reduced result; the result is then *reduced* to a canonical field representative.

Opening up this black box, we identify two techniques that modify, defer, or batch reductions: (i) a more efficient multiplication when one operand is “small” (fits in a machine word), and (ii) *delayed* (also called *deferred* or *lazy*) reduction, which minimizes Barrett or Montgomery reductions across long sequences of operations. As noted in Section 1.2, the latter is not new but, to our knowledge, has not been systematically applied to sum-check.

3.1 Multiplying with small integers

Recall from §2.1 that multiplying two N -limb Montgomery-form elements costs $2N^2 + N$ native multiplications. When one operand fits in a single 64-bit limb, we can skip Montgomery conversion and instead:

- (1) Multiply the N -limb field element by the small integer directly. This bignum multiply costs N native multiplications.
- (2) Apply a single optimized Barrett reduction to the $(N+1)$ -limb result, at $N+1$ native multiplications.

This drops the cost from $2N^2 + N$ to $2N + 1$ native multiplications, a 2-3× speedup for ~256-bit fields. See the full version for details.

We henceforth distinguish three multiplication types: *big-big* (**bb**, two field elements), *small-big* (**sb**, a machine integer times a field element), and *small-small* (**ss**, two machine integers). We treat **bb** as the dominant cost in complexity statements, accounting for **sb** and **ss** as secondary when relevant.

3.2 Small value arithmetic over tower fields

Tower fields admit an analogue of small-value arithmetic. Recall from §2.1 that a tower field $\mathbb{F}$ is built from a chain of quadratic extensions $\mathbb{B} \subset \mathbb{B}^{(1)} \subset \dots \subset \mathbb{B}^{(\ell)} = \mathbb{F}$. The recursive structure yields two benefits, already exploited in prior work [23, 24]:

- (1) **Compressed subfield representation:** An element $x \in \mathbb{B}^{(j)}$ is a $\mathbb{B}$ -linear combination of the first 2^j basis vectors of $\mathbb{F}$, so it is stored as 2^j base-field coefficients.
- (2) **Fast subfield multiplication:** To multiply $x \in \mathbb{B}^{(j)}$ by $y \in \mathbb{B}^{(j')}$ ($j < j'$), write $y = y_0 + y_1\alpha$ with $y_0, y_1 \in \mathbb{B}^{(j'-1)}$ and recurse: $x \cdot y = (x \cdot y_0) + (x \cdot y_1)\alpha$. The resulting algorithm costs only $2^{j'-j}$ multiplications in $\mathbb{B}^{(j)}$.

These let subfield operations behave as small-value arithmetic, yielding significant gains. See the full version for further details.

3.3 Delayed reduction

We now turn from a single multiplication to a linear combination, specifically a dot product of small integers with large field elements. This primitive is particularly useful for sum-check (Sections 4 and 5).

Formally, consider $S = \sum_{i=1}^k c_i \cdot a_i \pmod{p}$, with each c_i a 64-bit unsigned integer and each a_i a Montgomery-form field element.

A black-box approach uses k **bb** multiplications; our **sb** primitive improves this to k **sb**. Opening the box further, the per-step modular reductions can be deferred into a single reduction at the end.

The full procedure: compute integer products $T_i = c_i \cdot a_i$ ($\leq N + 1$ limbs each), sum $T = \sum_{i=1}^k T_i$, and apply a single Barrett reduction to obtain $S = T \bmod p$. When $\sum_i c_i < 2^{64}$, T stays within $N + 1$ limbs and the specialized single-pass Barrett reduction of the full version applies directly; otherwise T may grow to $N + 2$ limbs, requiring either a widened final reduction or batched reduction.

This amortizes reduction: k small-big integer multiplications (N each) plus one final reduction ($N + 1$), totaling $kN + (N + 1)$ native multiplications. That is nearly $2\times$ better than raw **sb** and far better than the black-box approach.

The primitive adapts to (i) larger c_i (e.g., 128-bit) by widening the intermediate representation, and (ii) negative c_i by splitting into positive and negative subsums, taking their difference, then reducing once.

Delayed reduction extends to field extensions; see the full version for full details.

Application to sum-check. Recall that in every round, the prover must evaluate expressions of the form

$$s_j(u) = \sum_{x' \in \{0,1\}^{\ell-j}} \prod_{i=1}^d p_i(r_{\lfloor \cdot \rfloor j}, u, x'),$$

which are long-running sums of products. For each summand, compute $h(x') := \prod_{i=1}^{d-1} p_i(r_{\lfloor \cdot \rfloor j}, u, x')$, form $h(x') \cdot p_d(r_{\lfloor \cdot \rfloor j}, u, x')$ in unreduced form, and defer reduction until many summands accumulate (or until the accumulator is updated). This replaces many per-term reductions by one per accumulator, lowering prover cost without protocol changes. The effect is most pronounced at low degrees ($d \in \{2, 3\}$), where the final multiplication dominates per-summand work; Section 9 measures it empirically.

Rough cost model. Later sections often trade one operation for a handful of another; for 256-bit fields we use a rough cost model. Setting $\mathfrak{ss} = 1$, we take **bb** additions, **sb** multiplications, **bb** multiplications, and **sb** linear combinations of n variables to cost roughly 5, 8, 32, and $5n + 3$, respectively. The model deliberately uses simple round numbers and is hardware- and field-dependent, but is borne out by the empirical results of Section 9.

4 Handling high-degree polynomials

Almost all sum-check provers must compute the product of d linear (or multilinear) polynomials, with d up to 32 in practice. Adding two v -variate degree- d polynomials costs $(d + 1)^v$ additions and no multiplications, but multiplying d such polynomials is much more expensive in practice.

We give a general algorithm for this product that is concretely faster than all prior approaches at all practical choices of (d, v) . For exposition, we treat the univariate case first, then generalize.

4.1 Product of univariate polynomials

Direct-evaluation baseline. Let $p_1, \dots, p_d$ be d linear polynomials. The standard approach evaluates the product polynomial $\prod_i p_i(x)$ directly at $d + 1$ points: extrapolate each p_i to U_d using $d - 1$ additions, then take pointwise products on U_d at $d - 1$ **bb** multiplications each. The total is $(d - 1)(d + 1)$ **bb** multiplications and $d(d - 1)$ additions. This algorithm is simple, optimal for $d = 2$, and widely used in practice as a baseline for small d .

Coefficient-domain divide and conquer. Hyperplonk [14] introduces a divide-and-conquer variant that pairs $p_1, \dots, p_d$ into a balanced binary tree and, at each internal node, multiplies the two subtree products using a fast polynomial-multiplication primitive (FFT for the asymptotic claim, Karatsuba as the practical fallback for low degree). With FFT this gives $O(d \log^2 d)$ **bb** multiplications, but FFTs require fields with smooth-order subgroups (absent in many SNARK-relevant fields), and the hidden constants make the algorithm slower than the standard approach at practical d . We thus take the standard approach as the baseline.

Evaluation-form recursion (this work). We introduce an alternative algorithm whose **bb** work is $O(d \log d)$, with $O(d^2)$ lower-order (**sb**/addition) cost. Since **sb** and additions are significantly cheaper than **bb**, this is concretely more efficient at practical d . We further optimize the constant factors over a naive realization.

The algorithm takes the evaluations of d linear polynomials at $\{0, 1\}$ and returns the evaluations of their product on U_d . It splits the d polynomials in half, recursively computes each half's product (yielding degrees $\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$ on $U_{\lfloor d/2 \rfloor}$ and $U_{\lceil d/2 \rceil}$), extrapolates both to U_d , and multiplies pointwise.

Let $a(d)$ be the number of **bb** multiplications incurred by this algorithm. A simple accounting gives the base case $a(1) = 0$ and recurrence $a(d) = a(\lfloor d/2 \rfloor) + a(\lceil d/2 \rceil) + (d + 1)$, leading to the closed form formula

$$a(d) = d \lceil \log_2 d \rceil + 2d - 2^{\lceil \log_2 d \rceil} - 1.$$

In particular, $a(d) \leq d \lceil \log_2 d \rceil + d - 1$, with equality at powers of two. A naive extrapolation step uses $\Theta(d^2)$ **sb** multiplications and a similar number of additions; these can be optimized via Section 3.3, and nearly all **sb** operations can be eliminated using Vandermonde structure and optimal multi-addition chains (see the full version).

As a subroutine in `LinearTimeSC`, this reduces **bb** multiplications from $\Theta(d^2 M)$ to $\Theta(d \log d \cdot M)$. A slight refinement omits the evaluation at 0, saving one **bb** multiplication; this yields a concrete improvement for all $d \geq 4$, taking the count from $12M$ to $10M$ at $d = 4$ and from $992M$ to $190M$ at $d = 32$ ($a > 5\times$ improvement).

4.2 Product of multilinear polynomials

The univariate algorithm is a special case of Procedure 1, which computes the product of d multilinear polynomials with minimal **bb** multiplications (used later in Section 5). It receives $p_1, \dots, p_d$ by their evaluations on $\{0, 1\}^v$ and returns $g = \prod_{k=1}^d p_k$ also in evaluation form, on the extended grid $U_d^v := \{0, 1, \dots, d - 1, \infty\}^v$.

Procedure 1 $\text{MultiProductEval}_{v,d}$: Efficient product of multilinear polynomials in evaluation form using extrapolation.

Input: d multilinear polynomials $p_1, \dots, p_d$ in v variables, specified by their evaluations over $\{0, 1\}^v$

Output: The product polynomial $g(x) = \prod_{i=1}^d p_i(x)$, specified by its evaluations over U_d^v

```

1: if  $d = 1$  then
2:    $g(x) \leftarrow p_1(x)$  {Identify the input two-point domain with  $U_1$  via a fixed bijection}
3:   return  $g$ 
4: else
5:    $m \leftarrow \lfloor d/2 \rfloor$ 
6:    $q_L \leftarrow \text{MultiProductEval}_{v,m}(p_1, \dots, p_m)$ 
7:    $q_R \leftarrow \text{MultiProductEval}_{v,d-m}(p_{m+1}, \dots, p_d)$ 
8:    $q'_L \leftarrow \text{MultiExtrapolate}_{v,m,d}(q_L)$ 
9:    $q'_R \leftarrow \text{MultiExtrapolate}_{v,d-m,d}(q_R)$ 
10:   $g \leftarrow q'_L \circ q'_R$  {Pointwise product of evaluations}
11:  return  $g$ 
12: end if

```

Procedure 2 $\text{MultiExtrapolate}_{v,k,d}$: Multivariate polynomial extrapolation from evaluations at U_k^v to U_d^v using a univariate extrapolation subroutine as a black-box.

Input: A v -variate polynomial p with per-variable degree at most k , specified as evaluations over U_k^v

Output: q , the evaluations of p over U_d^v
Initialize q with known evaluations over U_k^v

```

1:  $q \leftarrow p$ 
2: for  $j = 1$  to  $v$  do
   For each slice orthogonal to dimension  $j$ 
3:   for each  $(x_l, x_r) \in U_d^{j-1} \times U_k^{v-j}$  do
     Apply univariate extrapolation on the  $j$ th axis
4:      $q(x_l, U_d, x_r) \leftarrow \text{Extrapolate}_{k,d}(q(x_l, U_k, x_r))$ 
5:   end for
6: end for
7: return  $q$ 

```

The multivariate algorithm works with evaluations over U_d^v in place of U_d , so univariate extrapolations $U_k \rightarrow U_d$ become multivariate extrapolations $U_k^v \rightarrow U_d^v$.

Multivariate extrapolation (Procedure 2) is repeated univariate extrapolation along each dimension. In step $j \in [v]$, the evaluations on $U_d^{j-1} \times U_k^{v-j+1}$ are extended to $U_d^j \times U_k^{v-j}$ via $(d+1)^{j-1}(k+1)^{v-j}$ univariate extrapolations (one per setting of the other coordinates). The total over all v steps is

$$\sum_{j=1}^v (d+1)^{j-1}(k+1)^{v-j}$$

univariate extrapolations from U_k^v to U_d^v .

The **bb** count satisfies $a(d) = a(\lfloor d/2 \rfloor) + a(\lceil d/2 \rceil) + (d+1)^v$ with $a(1) = 0$. Combining with the extrapolation analysis of the full version, we obtain the following summary.

Table 3: Ratio of the number of **bb multiplications between a baseline multilinear polynomial product algorithm and our optimized algorithm (Procedure 1), for various numbers of variables v and polynomials d . These operations dominate the cost of the algorithm, so they can be treated as a proxy for overall speedup. * indicates entries where either the numerator or denominator exceeds 10^6 (the display cutoff used to keep the table readable). Our algorithm is significantly faster for all parameters listed and this speedup increases sharply as one moves down or right in the table.**

$d \backslash v$	1	2	3	4	5
3	$8/7$	$32/25$	$128/91$	$512/337$	$2048/1267$
4	$15/11$	$75/43$	$375/179$	$1875/787$	$9375/3611$
5	$24/16$	$144/70$	$864/334$	$5184/1714$	$31104/9286$
6	$35/21$	$245/99$	$1715/525$	$12005/3075$	$84035/19341$
7	$48/26$	$384/132$	$3072/782$	$24576/5220$	$196608/37646$
8	$63/31$	$567/167$	$5103/1087$	$45927/8135$	$413343/66271$
9	$80/37$	$800/213$	$8000/1513$	$80000/12501$	$800000/112897$
10	$99/43$	$1089/261$	$11979/1999$	$131769/18069$	*
16	$255/79$	$4335/623$	$73695/7087$	*	*
32	$1023/191$	$33759/2335$	*	*	*

LEMMA 4.1. For $v = 1$, the algorithm $\text{MultiProductEval}_{1,d}$ requires $O(d \log d)$ **bb** multiplications and $O(d^2)$ **sb** multiplications plus additions. For every fixed $v \geq 2$, $\text{MultiProductEval}_{v,d}$ requires $O(d^v)$ **bb** multiplications and $O(d^{v+1})$ **sb** multiplications plus additions.

The **sb**/addition cost is $\Theta(d^{v+1})$ because each univariate extrapolation from degree k to $2k$ costs $\Theta(k^2)$, and the per-level sums form a geometric series dominated by the root.

Using the extrapolation optimizations of the full version, Table 3 summarizes the **bb** ratio between the naive algorithm and ours for various (d, v) . In sum-check, omitting the evaluation at 0 saves part of one univariate extrapolation and one **bb** multiplication, though for large (d, v) this is marginal.

5 Small-value and streaming proving

Combining small-value arithmetic with our fast multilinear-product algorithm, we design a prover based on *round batching* that precomputes an evaluation grid and then answers a contiguous window of rounds by interpolating this grid at the verifier's challenges. This yields two simultaneous improvements: (i) in the *small-value* setting, a prover whose speedup over LinearTimeSC scales as $\Theta((d^2\kappa)^{1/\delta})$ for fixed d , where κ is the cost ratio between **bb** and **ss** multiplications and $\delta = \log_2(d+1)$; and (ii) a streaming prover, improving the dependence on d from $\Theta(d^2)$ to $\Theta(d \log d)$ over the prior algorithm of Baweja et al. [10]. The full version gives pseudocode and per-round cost analyses for the linear-time, logarithmic-space, and cross-product baselines used throughout the comparisons in this paper.

5.1 Small-value setting

We first motivate round batching in the context of small-value proving, then show how it generalizes to the streaming setting.

Taking advantage of small evaluations. Consider the linear-time algorithm $\text{LinearTime}_{\text{SC}}$ in the small-value setting applied to the polynomial

$$g(X_1, \dots, X_\ell) = \prod_{k=1}^d p_k(X_1, \dots, X_\ell).$$

In the first round, it is possible to perform $\$s$ and $\$b$ multiplications, as, by definition, the evaluations of g required for computing the first prover message are all small. However, from the second round onwards, the prover exclusively performs $\$b$ multiplications, since the evaluations after binding $p_k(r_1, \dots, r_{i-1}, x')$ are no longer small, due to the presence of the large random challenges $r_1, \dots, r_{i-1} \leftarrow \mathbb{F}$.

In each round, the number of evaluations required to compute the sum is halved, so this approach makes the most expensive round far cheaper. Extending this benefit to the first few rounds, by computing the first few messages with only $\$s$ and $\$b$ multiplications, would significantly reduce prover work.

The presence of the random values $r_1, \dots, r_{i-1}$ eliminates the possibility of exclusively using $\$s$ and $\$b$ multiplications; however, the effect of their contributions can be delayed.

We consider a prover which delays binding the variables $X_1, \dots, X_{i-1}$ to the values $r_1, \dots, r_{i-1}$, opting to treat them *symbolically* instead. This approach has a clear tradeoff: it eliminates a large number of $\$b$ multiplications, but it also increases the number of $\$s$ multiplications and additions required. If the prover takes this approach and wants to bind the polynomial only after the first v rounds, then it first needs to compute the v -variate polynomial that suffices to answer the first v rounds:

$$q(X_1, \dots, X_v) := \sum_{x' \in \{0,1\}^{\ell-v}} \prod_{k=1}^d p_k(X_1, \dots, X_v, x'),$$

purely from the evaluations of $p_1, \dots, p_d$ on $\{0,1\}^v$. For all x' , computing the product in the summand is *exactly* the multilinear product evaluation problem, which we have an efficient algorithm for in Procedure 1.⁶

Moreover, when the evaluations of $p_1, \dots, p_d$ over the Boolean hypercube $\{0,1\}^\ell$ are small (and d is not significantly larger than what is seen in practice), most $\$b$ and $\$s$ multiplications in MultiProductEval can be replaced with $\$s$ multiplications.⁷ This is because extrapolations of small values remain small, and the growth rate of the products of these extended evaluations is similarly controlled. For typical values of d and v , this primitive requires $\Theta(d^v)$ $\$s$ multiplications and not a single $\$b$ multiplication. More generally, the small-value specialization is valid whenever the extrapolated values stay within the $\$s/\$b$ kernels' machine-word budget, and every delayed-reduction accumulator stays within the limb bounds of Section 3.3; exact bounds and derivations appear in the full version.

Given this modified cost structure, the prover can tune v , the number of initial rounds to batch together before binding, to balance

⁶To be precise, it is possible to skip one out of $(d+1)^v$ evaluations of q , for instance the evaluation at the all-zero point. This is a simple generalization of the trick in the $v=1$ case, described in Section 2.3. The impact of this quickly becomes negligible as v grows.

⁷For exceptionally large d , it may also be better to use the evaluation points at $0, 1, \dots, d$ rather than $0, 1, \dots, d-1, \infty$ as this results in a smaller largest entry of the extrapolation matrix.

the savings from avoiding $\$b$ multiplications against the increased number of $\$s$ multiplications and additions.

5.2 Streaming setting

In the linear-time small-value setting we focused strictly on reducing runtime; however, the streaming setting imposes a space budget of $O(M^{1/k})$ for a tunable $k \geq 2$. As it turns out, an extension of the round batching idea is also well-suited to this setting when applied repeatedly, rather than just once.

Recall, as the sum-check progresses, the number of summands decreases by a factor of 2 each round. The $\text{LinearTime}_{\text{SC}}$ algorithm uses standard caching techniques to exploit this halving; however, these techniques require far more space than is available to the prover in the streaming setting.

Instead of caching, our prover batches rounds $[j, j + \omega - 1]$ together, which incurs a *normalized* overhead of $\approx (d+1)^\omega / 2^{j+\omega}$ relative to the linear-time baseline. The exponential decrease in j makes this overhead more tolerable as the protocol progresses, so we start with small windows and grow them as fewer variables remain to be processed.

Mechanically, the high-level design of our prover algorithm is similar to that of Baweja et al. [10] as they both consist broadly of three phases:

- (1) A time-constrained precomputation phase with geometrically growing window sizes.
- (2) A space-constrained precomputation phase with a fixed integer window chosen to meet the $O(M^{1/k})$ memory target.
- (3) A final phase identical to $\text{LinearTime}_{\text{SC}}$ once the remaining variable count is small enough.

However, at a low level, the implementation of each phase, as well as the points at which the prover transitions between phases differ significantly. The full version contains additional details, particularly on the schedule for phases.

Algorithm and cost summary. Figure 2 describes the complete algorithm with the option to specialize to the small-value or streaming setting (or both). The window schedule Ω determines the behavior: for streaming, Ω uses the progressive integer schedule shown there (with floor/clip operations and a final-phase cutoff at $\ell - S_{T+1} \leq \ell/k$), rather than the single early window used in the small-value setting.

In the small-value setting, with the cost analysis in the full version, we get the following result. Let κ denote the ratio of costs between $\$b$ and $\$s$ multiplications.⁸

LEMMA 5.1. *For fixed degree parameter d , at the optimal early-window size $v^* = \log_{d+1}(d^2\kappa)$ (clipped/rounded to valid rounds), $\text{EvalProductSV}_{\text{SC}}$ has big-multiplication-equivalent runtime*

$$\Theta\left(M \cdot d^{2-2/\delta} \cdot \kappa^{-1/\delta}\right),$$

and space usage

$$\Theta\left((d+1)^{v^*}\right) = \Theta(d^2\kappa),$$

⁸For large prime fields with Montgomery multiplication, $\kappa \approx 2N^2 + N$, where N is the number of limbs, so $\kappa = \Theta(N^2)$; for tower extension fields with Karatsuba, $\kappa \approx N^{\log_2(3)}$, where N is the extension degree.

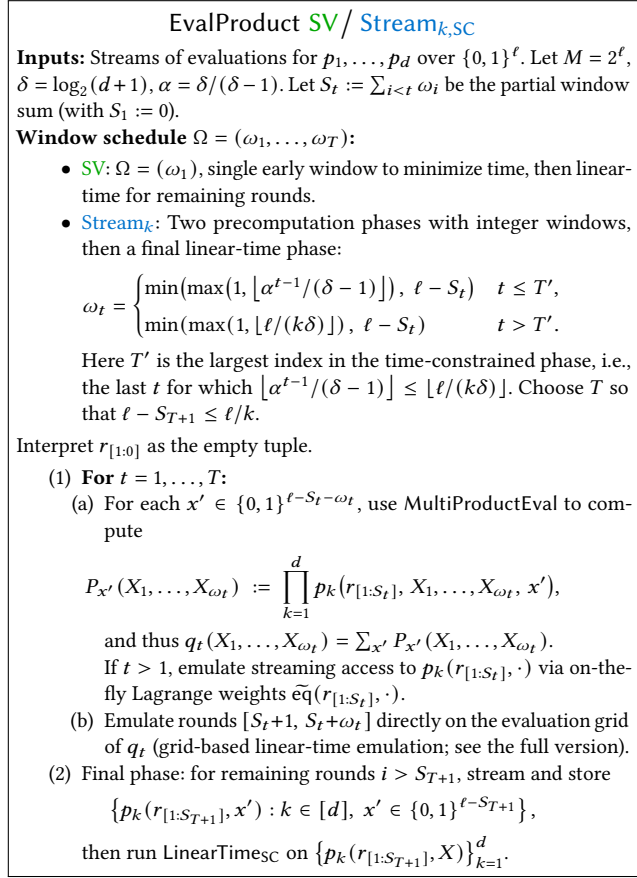


Figure 2: Combined small-value and streaming prover, parameterized by an integer window schedule $\Omega = (\omega_1, \dots, \omega_T)$ and a final linear-time phase.

where $\delta = \log_2(d+1)$. Compared with LinearTimeSC (runtime $\Theta(d^2M)$), this gives speedup $\Theta((d^2\kappa)^{1/\delta})$.

For the streaming setting, we have the following result.

LEMMA 5.2. *For products of d multilinear polynomials, the evaluation-basis round-batched prover*

$$\text{EvalProductStream}_{k,SC}$$

uses $O(M^{1/k})$ space and runs in time

$$\Theta((d \log d)M(k + \log \log M)).$$

This improves over the prior work of Baweja et al. [10] which has complexity

$$\Theta(d^2M(k + \log \log M)).$$

Hidden in these asymptotics are small constants which we also improve on for parameters seen in practice. Roughly speaking, their algorithm computes the *coefficients* rather than the evaluations of the product polynomials, which is more expensive. A full analysis is provided in the full version.

The improvement stems from a better growth base $\alpha = \delta/(\delta-1)$ compared to $\alpha_{\text{old}} = d/(d-1)$ that reduces the number of time-constrained passes, and more efficient evaluation-form storage that

increases the allowable space-constrained window size from $\ell/(dk)$ to $\ell/(k\delta)$. However, each pass still requires $\Theta(d \cdot M)$ work for prefix adaptation to handle intermediate challenges.

6 Decomposable multilinear polynomials

Many applications of the sum-check protocol involve a polynomial which has as one of its factors a multilinear polynomial that decomposes into a product of univariate linear polynomials. The canonical example, and most common one, is the *equality polynomial* $\tilde{eq}(w, X) = \prod_{i=1}^\ell \tilde{eq}(w_i, X_i)$ with a randomly chosen challenge vector w . It arises whenever a protocol batches multiple constraints into a single sum-check; for instance, the Spartan protocol [42] verifies RICS satisfiability by checking that

$$\sum_{X \in \{0,1\}^\ell} \tilde{eq}(w, X) \cdot (\tilde{Az}(X) \cdot \tilde{Bz}(X) - \tilde{Cz}(X)) = 0, \quad (7)$$

where $\tilde{Az}, \tilde{Bz}, \tilde{Cz}$ are multilinear extensions derived from the constraint matrices A, B, C and a witness vector z . For clarity we focus on $\tilde{eq}$ throughout this section, but our techniques apply to any decomposable multilinear polynomial.

The three-part decomposition (Gruen). For a window of rounds $[i, i+v-1]$, the linear-factor structure of $\tilde{eq}(w, X)$ splits into three pieces:

- **Past:** a constant $\alpha = \tilde{eq}(w_{[<i]}, r_{[<i]})$ on the already-bound variables.
- **Window:** a product of linear factors $\prod_{j=1}^v \tilde{eq}(w_{i+j-1}, X_j)$ on the live variables.
- **Future:** weights $\tilde{eq}(w_{[>i+v-1]}, x')$ for $x' \in \{0, 1\}^{\ell-i-v+1}$ on the unbound variables.

This decomposition is due to Gruen [28] (in the per-round $v = 1$ case), who observed that the past and window pieces can be reconstructed by the verifier and need not appear in the prover's per-round message.⁹ The per-window task reduces to evaluating

$$T(X_1, \dots, X_v) := \sum_{x'} \tilde{eq}(w_{[>i+v-1]}, x') \prod_{k=1}^d p_k(r_{[<i]}, X_1, \dots, X_v, x')$$

on the grid U_d^v . Combined with our multiproduct evaluation (Section 4), this reduces the number of **bb** multiplications by more than $2.3\times$ at $d = 3, v = 4$.

Splitting the future-rounds table (this work). Gruen's algorithm still materializes the future-rounds table $\{\tilde{eq}(w_{[>i+v-1]}, x')\}_{x'}$, costing $\Theta(2^{\ell-i-v+1})$ **bb** multiplications per window ($\Theta(2^\ell)$ in the first one). We further decompose this factor: partitioning the suffix into $k \geq 2$ groups gives

$$\tilde{eq}(w_{[>i+v-1]}, x') = \prod_{j=1}^k \tilde{eq}(w_{\text{range}_j}, x'_j),$$

where the ranges partition the suffix coordinates and x'_j is the corresponding projection of x' . The single big table is replaced by k tables of size $\approx M^{1/k}$ (where $M = 2^\ell$), combined on the fly while streaming over x' . Space drops from $\Theta(M)$ to $O(M^{1/k})$ and table construction from $\Theta(M)$ to $O(kM^{1/k})$ **bb** multiplications. The two-way ($k = 2$)

⁹See also [46] for a small variation that keeps the verifier unchanged.

variant gives the best tradeoff in practice, saving roughly $M/2$ field multiplications beyond Gruen’s per-round baseline.

Algorithm and cost. For $g(X) = \tilde{\text{eq}}(w, X) \cdot \prod_{k=1}^d p_k(X)$ over rounds $[i, i + v - 1]$, the prover (i) builds the k small future-eq tables, (ii) streams $x' \in \{0, 1\}^{\ell-i-v+1}$ and accumulates T on U_d^v via the multiproduct evaluation of Section 4, and (iii) multiplies T pointwise by $\alpha \cdot \prod_{j=1}^v \tilde{\text{eq}}(w_{i+j-1}, X_j)$ to extract the round messages. Compared to sum-check on $\prod_k p_k(X)$ alone, this adds per window:

- **Split-Eq tables.** $O(k M^{1/k})$ **bb** multiplications to build.
- **Eq scaling.** $(d+1)^v \cdot 2^{\ell-i-v+1}$ multiplications, one per grid point per suffix; in the small-value setting these are **sb**.
- **Window-linear extraction.** $(d+1)^v$ **sb** multiplications, applied once when multiplying T by the window-linear factors.
- **Space.** $O(M^{1/k})$ for the suffix-eq tables.

In the small-value setting, the multiproduct itself stays on U_d^v at $\Theta(d^v) \cdot 2^{\ell-i-v+1}$ **ss** multiplications, against $\Theta((d+1)^v) \cdot 2^{\ell-i-v+1}$ **sb** on U_{d+1}^v for the naive eq-as-extra-factor baseline. See the full version for the full algorithm and the iterated-sum cost.

7 Univariate Skip over Large Prime Fields

Univariate skip, introduced by Gruen [28], is a generic way to reduce prover work by “fusing” several Boolean variables into a single higher-degree variable, in a setting where we start out with base field values but sum-check is over the extension field. In this section, we show that univariate skip can also apply in the setting of small values inside a large prime field, which allows us to apply the technique to applications such as Spartan-in-Jolt.

General idea. Let $f : \mathbb{F}^n \rightarrow \mathbb{F}$ be the polynomial being summed. Univariate skip chooses a set of m components to pack into one variable Y (here m is a design parameter, not necessarily a power of two), and an interpolation domain $D = \{u_0, \dots, u_{m-1}\} \subset \mathbb{F}$ of the same size. Let $\{L_j(Y)\}_{j=0}^{m-1}$ be the Lagrange basis for D . Define the packed polynomial

$$f_{\text{pack}}(Y, \cdot) := \sum_{j=0}^{m-1} L_j(Y) \cdot f_j(\cdot),$$

where $f_0, \dots, f_{m-1}$ are the objects being packed (for example, slices indexed by Boolean prefixes, or batched constraint components in zero-check). By construction, $f_{\text{pack}}(u_j, \cdot) = f_j(\cdot)$ for all j , and operationally this replaces several rounds/components by one higher-degree univariate step. The familiar Boolean-block version is the special case $m = 2^k$ with $f_j(\cdot) = f(\text{bin}(j), \cdot)$ for a k -bit prefix; in batched zero-check settings, m can instead match the number of packed constraints or terms.

Adapting to large-prime fields. Gruen’s instantiation [28] chooses D as an m -sized multiplicative subgroup of a small base field, leveraging a base/extension-field split that does not exist in our setting. For small values inside a large prime field, we instead take a small symmetric integer interval around 0:

$$D_m := \{-\lfloor m/2 \rfloor, -\lfloor m/2 \rfloor + 1, \dots, -\lfloor m/2 \rfloor + m - 1\} \subset \mathbb{F}.$$

This choice keeps interpolation points numerically small and lets signed partial sums cancel in our accumulators; combined with our small-by-big multiplications (§3.1) and delayed reductions (§3.3),

even a quadratic-in- m interpolation step has low concrete cost. We defer the detailed justification, the explicit Lagrange-coefficient bound, and the resulting practical limit on m to the full version.

Applicability. Univariate skip is most useful when the prover can choose a block of variables to pack and then reuse this structure across many evaluations, as arises in batched zero-check and other sum-check instances. As in Gruen’s analysis [28], the parameter m gives a prover-verifier tradeoff: larger m gives more prover amortization but increases the degree and size of the first message and the verifier’s evaluation cost.

Verifier work and proof size. The current Jolt Spartan Outer instance has 19 constraints, whose ordinary encoding uses five Boolean constraint-index variables. Jolt splits them into groups of 10 and 9: one Boolean variable selects the group, while a size-10 univariate coordinate replaces the other four Boolean variables; see the full version for details. The resulting first-round polynomial has degree at most 27 and sends 28 coefficients. The four compressed cubic rounds it replaces send 12 field elements, so the round-message payload grows by 16 BN254 elements. Jolt also sends the polynomial’s value at the sampled point to link this round to the remaining sum-check. The total added field payload is therefore 17 elements, or 544 bytes.

The verifier checks the size-10 domain sum with a 28-term dot product and checks that the linking claim equals the degree-27 polynomial evaluated at the sampled point. In the current implementation, this replaces 20 general field multiplications and 28 additions for the four cubic checks with 54 general field multiplications, 28 small-integer multiplications, and 55 additions or subtractions. The optimization does not change the polynomial-commitment openings (Table 1).

8 Implementation

For field arithmetic, we extended the arkworks [4] library with optimized small-by-big multiplications (Section 3.1) and delayed-reduction routines (Section 3.3), exposing bignum multiplication separately from modular reduction so that both can compose inside long-running sum-check accumulators.

Our Jolt integration applies our techniques in three places, with the algorithmic detail given in the full version:

- **Delayed reduction** (Section 3) is applied across all sum-check-heavy components.
- For the **Spartan Outer sum-check**, we implement an ablation chain over a no-optimization baseline, adding split-eq (Section 6), delayed reduction, round batching (Section 5.1), and univariate skip (Section 7); we also include streaming variants of our approach (Section 5) and Baweja et al. [10].
- For the **Instruction RA Virtualization** sum-check (Shout), we apply our high-degree multiproduct kernel together with split-eq; the effective degree is configurable in $\{4, 8, 16, 32\}$.

9 Evaluation

Our evaluation aims to answer the following questions:

- (1) How do our new white-box finite field techniques compare to black-box alternatives?

- (2) How do our techniques improve the prover time and space of various sum-check primitives over prior work?
- (3) What is the end-to-end impact of our techniques on real-world systems?

Experimental setup. All experiments are run over the BN254 scalar field (the field currently used in Jolt). We run benchmarks on a MacBook Pro with Apple M4 Max (12 performance cores, 64 GB RAM). We use Criterion [29] to run the microbenchmarks and report its timing estimates, and Chrome-trace [27] span analysis for Instruction RA Virtualization measurements in Jolt. All microbenchmarks are run single-threaded, and the multithreaded benchmarks are specified as such. Every reported speedup compares variants on the same machine, field backend, and thread configuration. For memory, we use three complementary metrics: isolated process peak RSS (Resident Set Size), which measures the maximum amount of physical RAM a process consumes over its lifetime, for Spartan Outer runs, setup-normalized Δ peak RSS for streaming comparisons (Δ peak RSS = peak RSS – RSS immediately after trace/setup), and allocation-level heap snapshots for retained-state accounting. This allows us to measure the memory overhead of components when compared to the rest of Jolt.

9.1 The cost of finite field operations

We first benchmark primitive finite-field operations, including $\$s$, $\$b$, and $\$b$ multiplication costs and their internal breakdowns.

Next, we benchmark linear combinations with small coefficients using: (i) naive $\$b$ multiplication, (ii) $\$b$ multiplication, and (iii) delayed reduction across the whole combination.

Finally, we benchmark the product of d multilinear polynomials in v variables (the core high-degree kernel from Section 4). Table 5 reports speedups of our optimized method over the naive baseline. Speedups grow with d and v , reaching about 2.5 $\times$ at $(d, v) = (32, 18)$ and $(32, 20)$.

9.2 Delayed reduction in degree-2 sum-check

Tables 6 and 7 isolate delayed reduction (Section 3) inside two single-threaded sum-check prover microbenchmarks: a clean degree-2 instance $\sum_x p(x) \cdot q(x)$ and a degree-2 product with an equality-polynomial factor, $\sum_x \tilde{eq}(\tau, x) \cdot p(x) \cdot q(x)$, using split-eq (Section 6). The clean instance gains a stable $\approx 1.37\times$ across variable sizes 14–26; with split-eq applied, the gain shrinks to a consistent $\approx 1.09\times$ since fewer multiplications remain on the critical path where reductions can be deferred. The larger ratio between an isolated generic multiplication and a specialized small-value operation is not an expected full-prover speedup: table traversal, additions, challenge binding, memory traffic, and ineligible field operations remain. These measurements isolate the resulting degree-2 prover improvement.

9.3 Jolt integration benchmarks

We evaluate two Jolt integrations: Spartan Outer (small-value and equality-polynomial optimizations) and Instruction RA Virtualization inside Shout (the high-degree optimization); see the full version for full details. We report Spartan ablations as SHA2-chain- I , where I is the number of SHA-256 iterations. One iteration is $\approx 3.4 \times 10^3$ RISC-V cycles, so the workload is $\approx 3.4 \times 10^3 \cdot I$ cycles (e.g., $I = 8192$ is about 2.8×10^7 cycles). For end-to-end results, we use a *scale*

parameter: scale s means the trace is padded to $\approx 2^s$ cycles (e.g., at $s = 22$ the unpadded trace is about 3.8M cycles, padded to about 4.2M cycles in total).

Spartan ablation (linear-space). Figure 3 reports a cumulative ablation¹⁰ over variants of Spartan Outer:

- **Baseline:** textbook linear-time Spartan Outer prover, no optimizations.
- **Split-eq:** adds equality-polynomial factorization (Section 6).
- **Delayed reduction:** adds delayed reduction (Section 3).
- **Round batching:** adds the multi-round small-value precomputation strategy of Section 5.1.
- **Univariate skip:** replaces round batching with univariate skip (Section 7); our fastest linear-space variant and the current Jolt default.

Performance improves monotonically across the chain. The dominant transition is delayed reduction $\rightarrow$ round batching: 4.10–6.01 $\times$ in the steady-state compute regime ($I = 128$ –2048, about 4.4×10^5 – 7.0×10^6 RISC-V 64-bit (RV64) cycles), widening to 8.81–14.95 $\times$ in the memory-bottleneck regime ($I \geq 4096$) where dense variants become memory-bound; other per-stage transitions contribute 1.02–1.47 $\times$ each in steady state and 1.02–3.01 $\times$ in the memory-bottleneck regime. Cumulatively, at $I = 8192$ univariate skip is 57.9 $\times$ faster than baseline (about 2.1s vs. 121s) and round batching is 45.9 $\times$ faster, with isolated peak RSS dropping from 46.6–50.1 GB (dense variants) to 11.5–15.0 GB. Univariate skip changes the prover/verifier tradeoff and is discussed separately in Section 7.

Streaming regime (low-space). Table 8 compares linear-space univariate skip against two $\sqrt{M}$ -space streaming variants: our *evaluation-basis streaming* prover and a *cross-product streaming* baseline (coefficient form, following Baweja et al.). Evaluation-basis streaming is consistently faster than cross-product streaming: it has a 1.6–2.7 $\times$ slowdown vs. univariate skip across all sizes, compared with 1.9–3.0 $\times$ for cross-product. Setup-normalized Δ peak RSS confirms the intended $\sqrt{M}$ working set: at $I = 8192$, both streaming variants use ≤ 36 MB versus 8.59 GB for univariate skip, two orders of magnitude lower (the $I \leq 64$ values sit near the allocator/setup noise floor and we interpret them only qualitatively).

Instruction RA Virtualization in Shout. Table 9 reports trace measurements at scale $s = 22$ ($\approx 2^{22}$ cycles) for $n_{RA} \in \{8, 4, 2, 1\}$ virtual polynomials (degrees 5, 9, 17, 33): total Instruction RA Virtualization time improves by 1.70–2.18 $\times$ over baseline kernels, substantially reducing its share of total proving time.

End-to-end sweep. Table 10 reports a consolidated SHA2-chain sweep for scale parameters $s = 22$ –25 (3 runs per scale, default optimized configuration): proving time grows from 9.76 ± 0.12 s to 51.09 ± 1.11 s, with the Instruction RA Virtualization share rising from 8.0% to 11.8% and the Spartan Outer share remaining smaller (3.0%–4.2%). This sweep measures the optimized full prover rather than a paired full-prover ablation. The paired effects of the techniques are isolated in Figure 3 and Table 9; the end-to-end sweep shows their component shares and the resulting whole-system time and memory footprint.

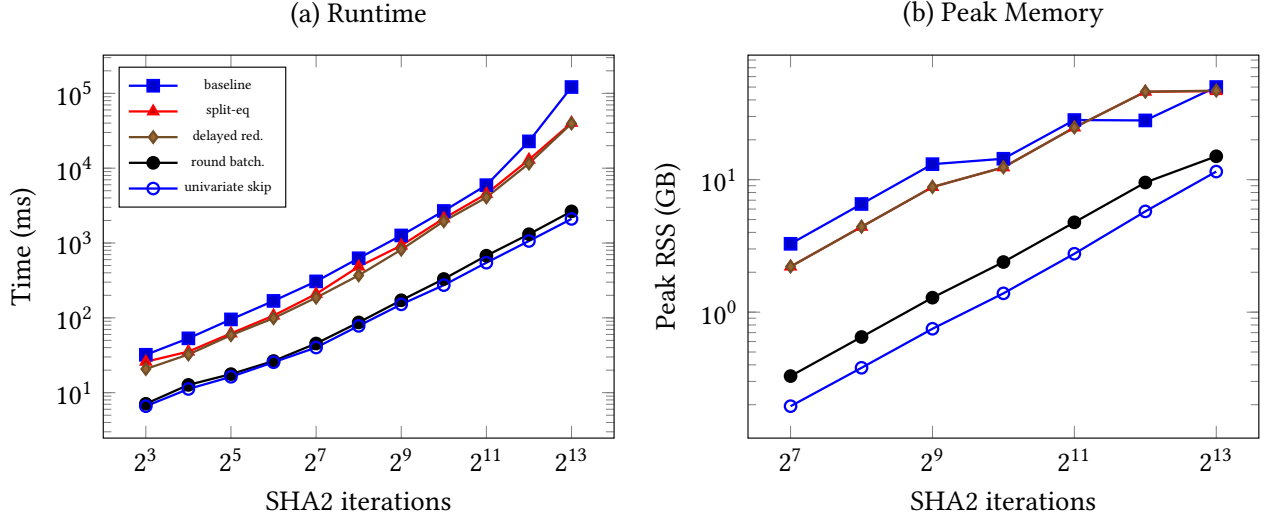
¹⁰By *ablation*, we mean starting from a baseline and adding one optimization at a time to isolate each optimization’s contribution.

Table 4: Single-threaded BN254 field-arithmetic microbenchmarks on the Apple M4 Max. Small-big multiplication avoids a full Montgomery multiplication, while delayed reduction amortizes reduction across a complete linear combination.**(a) Primitive operations.** Constituent timings are measured separately, so they need not sum to the combined timing.

Operation	Time (ns)
$\$s$ multiply	0.36
$\$b$ add	1.99
$\$b$ multiply	4.65
integer multiply	2.02
Barrett reduction	3.67
$\$b$ multiply	10.50
integer multiply	4.48
Montgomery reduction	6.67

(b) Linear combinations with small coefficients. The speedup compares delayed reduction with baseline $\$b$ arithmetic.

Linear-combination size	2	4	8	16	32
Baseline (ns)	55.10	107.27	212.85	438.95	873.31
$\$b$ (ns)	11.49	23.07	42.57	81.71	167.60
Delayed reduction (ns)	5.26	9.32	12.08	26.00	36.84
Speedup ($\times$)	10.48	11.51	17.62	16.88	23.71

**Figure 3: Spartan Outer sum-check ablation in Jolt. (a) Runtime in milliseconds; (b) peak process RSS in GB. Both axes use logarithmic scales. Round batching and univariate skip preserve near-linear scaling, while the dense baselines encounter memory pressure at the largest workloads.**

Memory measurements. From allocation-level runs at 16k and 32k SHA2 iterations, we measure a stable $\approx 4.00\times$ retained-state reduction for streaming versus linear-space univariate skip; at SHA2-chain 128–8192 (Figure 3), we find univariate skip’s isolated peak RSS $\approx 4.4\text{--}17.5\times$ below baseline, with round batching $\approx 1.3\text{--}1.7\times$ above univariate skip. End-to-end, we measure process peak memory of 4.98, 9.91, 14.12, and 27.67 GB at scales $s = 22, 23, 24, 25$ (Monitor/Perfetto, single run per scale).

10 Discussion

The techniques in this paper occupy different points in the prover time-space tradeoff. When linear space is available, round batching and univariate skip give the fastest Spartan Outer provers in our experiments. When memory is constrained, evaluation-basis streaming reduces the working set to 18 MB at $I = 8192$, compared with 8.59 GB for univariate skip, at a $1.6\text{--}2.7\times$ runtime cost across

the measured range (Table 8). All techniques except univariate skip leave the protocol, verifier cost, and soundness analysis unchanged. Univariate skip preserves soundness but adds 544 bytes of field-element payload and replaces four cubic verifier checks with one degree-27 check in the current Jolt instance (Section 7 and Table 1).

Scope and limitations. Our measurements use the BN254 scalar field and an Apple M4 Max. The relative cost of integer multiplication, modular reduction, memory traffic, and parallelism changes across fields and machines, so the exact speedups need not transfer unchanged. All ratios compare implementations in the same Arkworks backend and on the same machine; we claim portability for the operation-count reductions, not for the exact timing ratios. The algorithmic savings do transfer: small-big multiplication avoids a full Montgomery multiplication, delayed reduction removes repeated reductions, and the high-degree algorithm reduces

Table 5: High-degree product benchmark for computing the product of d multilinear polynomials in v variables. Values are Criterion mean estimates from a dedicated microbench.

Speedup (base/opt)	$v = 12$	$v = 16$	$v = 20$
$d = 2$	1.12×	2.07×	2.92×
$d = 4$	1.17×	2.04×	2.07×
$d = 8$	1.27×	1.83×	2.01×
$d = 16$	1.33×	1.91×	2.04×
$d = 32$	1.48×	2.31×	2.47×
<i>Optimized runtime (μs)</i>			
$d = 2$	88	246	2,235
$d = 4$	145	606	7,038
$d = 8$	351	1,640	21,262
$d = 16$	949	5,648	76,196
$d = 32$	3,054	17,120	237,080
<i>Baseline runtime (μs)</i>			
$d = 2$	98	509	6,521
$d = 4$	170	1,235	14,568
$d = 8$	446	2,997	42,677
$d = 16$	1,263	10,786	155,200
$d = 32$	4,511	39,618	585,530

Table 6: Degree-2 sum-check prover microbenchmark for $\sum_x p(x) \cdot q(x)$, comparing baseline multiplication against delayed reduction in these prover kernels.

Variables	Baseline (ms)	Unreduced (ms)	Speedup
14	1.15	0.87	1.31×
16	5.42	3.98	1.36×
18	21.0	15.2	1.38×
20	82.6	59.4	1.39×
22	331.5	235.2	1.41×
24	1,324	965	1.37×
26	5,316	3,819	1.39×

Table 7: Degree-2 sum-check prover microbenchmark with an equality-polynomial factor, $\sum_x \tilde{eq}(\tau, x) \cdot p(x) \cdot q(x)$. Both variants use split-eq; the optimized variant additionally uses delayed reduction.

Variables	Baseline (ms)	Unreduced (ms)	Speedup
14	2.63	2.49	1.06×
16	9.51	8.57	1.11×
18	35.0	32.2	1.09×
20	136.4	124.3	1.10×
22	538.9	494.9	1.09×

the number of big-big multiplications (Sections 3 and 4). Small-value arithmetic also requires explicit bounds on coefficients and unreduced accumulators. If those bounds exceed the available limbs, the implementation must use a widened accumulator or reduce in batches (Section 3.3).

Table 8: Spartan Outer runtime and memory tradeoff. The slowdown range is measured against univariate skip over $I = 8-8192$; runtime and setup-normalized Δ peak RSS are shown at $I = 8192$.

Variant	Runtime	Slowdown	Δ peak RSS
Univariate skip	2.099 s	1.00×	8.593 GB
Evaluation-basis stream	5.647 s	1.62–2.70×	0.018 GB
Cross-product stream	6.122 s	1.88–2.97×	0.036 GB

Table 9: Jolt Shout Instruction RA virtualization benchmark at $s = 22$ (multithreaded). Deg denotes degree; Base/Opt are baseline/optimized times; Speedup is Base/Opt; and %prov (b)/(o) are their shares of total proving time.

Deg	Base (ms)	Opt (ms)	Speedup	%prov (b)	%prov (o)
5	1,602	921	1.74×	13.0%	8.4%
9	2,135	1,247	1.71×	17.5%	11.2%
17	3,837	2,254	1.70×	26.0%	17.1%
33	6,501	2,979	2.18×	39.7%	22.8%

Table 10: End-to-end SHA2-chain proving sweep in Jolt (default optimized configuration, BN254, 3 runs per scale, mean $\pm$ standard deviation). Component shares are extracted from Chrome-trace spans. These include optimized implementations of both Instruction RA Virtualization and the Spartan Outer sum-check.

Scale	Prove time (s)	Instruction RA share	Spartan Outer share
22	9.76 $\pm$ 0.12	8.0%	3.0%
23	15.72 $\pm$ 0.18	9.6%	3.4%
24	29.79 $\pm$ 0.38	10.2%	3.6%
25	51.09 $\pm$ 1.11	11.8%	4.2%

Our implementation evaluates the prime-field versions of these routines. The tower-field and extension-field analogues in Section 3.2 and the full version have not yet been benchmarked in a complete proof system. Benchmarking those variants and tuning their schedules to machine-specific costs remain future work.

Acknowledgments

Quang Dao thanks Riad Wahby for helpful discussions, especially on the delayed-reduction technique and its implementation.

Disclosures

Justin Thaler is a Research Partner at a16z crypto and is an investor in various blockchain-based platforms, as well as in the broader crypto ecosystem (see general a16z disclosures).

References

- [1] 2020. Spartan: High-speed zkSNARKs without trusted setup. <https://github.com/Microsoft/Spartan>.
- [2] 2025. Jolt Codebase. <https://github.com/a16z/jolt>.
- [3] Diego F Aranha, Koray Karabina, Patrick Longa, Catherine H Gebotys, and Julio López. 2011. Faster software for sparse isogenies and pairings. *Cryptology ePrint*

- Archive (2011).
- [4] arkworks contributors. 2022. arkworks zkSNARK ecosystem. <https://arkworks.rs>.
 - [5] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. 2025. WHIR: Reed–Solomon Proximity Testing with Super-Fast Verification. In *Advances in Cryptology – EUROCRYPT 2025 (Lecture Notes in Computer Science, Vol. 15604)*. Springer, 214–243. doi:10.1007/978-3-031-91134-7_8
 - [6] Arasu Arun, Srinath Setty, and Justin Thaler. 2024. Jolt: Snarks for virtual machines via lookups. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.
 - [7] Suyash Bagad, Quang Dao, Yuval Domb, and Justin Thaler. 2025. Speeding Up Sum-Check Proving. Cryptology ePrint Archive, Paper 2025/1117. <https://eprint.iacr.org/2025/1117>
 - [8] Suyash Bagad, Yuval Domb, and Justin Thaler. 2024. The Sum-Check Protocol over Fields of Small Characteristic. Cryptology ePrint Archive, Paper 2024/1046. <https://eprint.iacr.org/2024/1046>
 - [9] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.
 - [10] Anubhav Baweja, Alessandro Chiesa, Elisabetta Fedele, Giacomo Fenzi, Pratyush Mishra, Tushar Mopuri, and Andrew Zitek-Estrada. 2025. Time-Space Trade-Offs for Sumcheck. In *Theory of Cryptography Conference (TCC) 2025 (Lecture Notes in Computer Science, Vol. 16271)*. Springer, 37–70. doi:10.1007/978-3-032-12290-2_2
 - [11] Dan Boneh and Binyi Chen. 2025. LatticeFold: A Lattice-Based Folding Scheme and Its Applications to Succinct Proof Systems. In *Advances in Cryptology – ASIACRYPT 2025 (Lecture Notes in Computer Science, Vol. 16247)*. Springer, 330–362. doi:10.1007/978-981-95-5099-9_11
 - [12] Dan Boneh and Binyi Chen. 2025. LatticeFold+: Faster, Simpler, Shorter Lattice-Based Folding for Succinct Proof Systems. In *Advances in Cryptology – CRYPTO 2025 (Lecture Notes in Computer Science, Vol. 16006)*. Springer, 327–361. doi:10.1007/978-3-032-01907-3_11
 - [13] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. 2019. Fiat–Shamir: From Practice to Theory. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 1082–1090. doi:10.1145/3313276.3316380
 - [14] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. 2023. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. In *Advances in Cryptology – EUROCRYPT 2023 (Lecture Notes in Computer Science, Vol. 14005)*. Springer, 499–530. doi:10.1007/978-3-031-30617-4_17
 - [15] Alessandro Chiesa, Elisabetta Fedele, Giacomo Fenzi, and Andrew Zitek-Estrada. 2024. A Time-Space Tradeoff for the Sumcheck Prover. Cryptology ePrint Archive, Paper 2024/524. <https://eprint.iacr.org/2024/524>
 - [16] Consensys. 2024. gnark-crypto: High-performance cryptographic and elliptic curve library in Go. <https://github.com/Consensys/gnark-crypto>.
 - [17] Stephen A. Cook. 1969. On the minimum computation time of functions. In *Proceedings of the first annual ACM symposium on Theory of computing*. ACM, 187–192.
 - [18] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. 2012. Practical Verified Computation with Streaming Interactive Proofs. In *ITCS*.
 - [19] Graham Cormode, Justin Thaler, and Ke Yi. 2011. Verifying Computations with Streaming Interactive Proofs. *Proc. VLDB Endow.* 5, 1 (2011), 25–36. doi:10.14778/2047485.2047488
 - [20] Quang Dao and Justin Thaler. 2024. Constraint-Packing and the Sum-Check Protocol over Binary Tower Fields. Cryptology ePrint Archive, Paper 2024/1038. <https://eprint.iacr.org/2024/1038>
 - [21] Quang Dao and Justin Thaler. 2024. More Optimizations to Sum-Check Proving. Cryptology ePrint Archive, Paper 2024/1210. <https://eprint.iacr.org/2024/1210>
 - [22] Ben Diamond. 2025. Sumchecks of Medium Degree. https://hackmd.io/@benediamond/Sye_bB1wJl. Accessed 2026-03-24.
 - [23] Benjamin E. Diamond and Jim Posen. 2025. Succinct Arguments over Towers of Binary Fields. In *Advances in Cryptology – EUROCRYPT 2025 (Lecture Notes in Computer Science, Vol. 15604)*. Springer, 93–122. doi:10.1007/978-3-031-91134-7_4
 - [24] Benjamin E. Diamond and Jim Posen. 2026. Polylogarithmic Proofs for Multilinear over Binary Towers. In *Advances in Cryptology – EUROCRYPT 2026 (Lecture Notes in Computer Science, Vol. 16547)*. Springer, 3–32. doi:10.1007/978-3-032-25336-1_1
 - [25] Yuval Domb. 2024. The Barrett-Montgomery Duality and a New Multi-Precision Modular Reduction Scheme with Only $n^2 - 1$ Operations. <https://medium.com/@ingonyama/the-barrett-montgomery-duality-and-a-new-multi-precision-modular-reduction-scheme-with-only-n2-1-b40ede4792da>. Accessed: 2025-04-09.
 - [26] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. 2015. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)* 62, 4 (2015), 1–64.
 - [27] Google. 2024. Trace Event Format. <https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OQQtYMH4h6lOnSsKchNaySU/preview>.
 - [28] Angus Gruen. 2024. Some Improvements for the PIOP for ZeroCheck. *Cryptology ePrint Archive* (2024).
 - [29] Brook Heisler. 2024. Criterion.rs: Statistics-driven Micro-benchmarking in Rust. <https://github.com/criterion-rs/criterion.rs>.
 - [30] Anatolii Alexeevich Karatsuba. 1995. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211 (1995), 169–183.
 - [31] Abhiram Kothapalli and Srinath Setty. 2024. HyperNova: Recursive arguments for customizable constraint systems. In *Annual International Cryptology Conference*. Springer, 345–379.
 - [32] Abhiram Kothapalli and Srinath Setty. 2026. NeutronNova: Group-Based Folding Done Right. In *Advances in Cryptology – CRYPTO 2026 (Lecture Notes in Computer Science, Vol. 16808)*. Springer, 524–556. doi:10.1007/978-3-032-35424-2_18
 - [33] Tianyi Liu and Yupeng Zhang. 2025. Efficient SNARKs for Boolean Circuits via Sumcheck over Tower Fields. Cryptology ePrint Archive, Paper 2025/594. <https://eprint.iacr.org/2025/594>
 - [34] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. 2025. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. *Journal of Cryptology* 38, 2 (2025), 17.
 - [35] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1990. Algebraic methods for interactive proof systems. In *FOCS*.
 - [36] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44, 170 (1985), 519–521. doi:10.2307/2007970
 - [37] Vineet Nair, Justin Thaler, and Michael Zhu. 2025. Proving CPU Executions in Small Space. Cryptology ePrint Archive, Paper 2025/611. <https://eprint.iacr.org/2025/611>
 - [38] Wilson Nguyen and Srinath Setty. 2026. Neo and SuperNeo: Post-quantum Folding with Pay-per-Bit Costs over Small Fields. In *Advances in Cryptology – CRYPTO 2026 (Lecture Notes in Computer Science, Vol. 16808)*. Springer, 492–523. doi:10.1007/978-3-032-35424-2_17
 - [39] Andrija Novakovic and Guillermo Angeris. 2025. Ligerito: A Small and Concretely Fast Polynomial Commitment Scheme.
 - [40] Michael Scott. 2007. Implementing cryptographic pairings. In *Pairing-Based Cryptography–Pairing 2007: First International Conference, Tokyo, Japan, July 2–4, 2007. Proceedings 1*. Springer, 177–196.
 - [41] Michael Scott. 2017. Slothful reduction. Cryptology ePrint Archive, Paper 2017/437. <https://eprint.iacr.org/2017/437>
 - [42] Srinath Setty. 2020. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Annual International Cryptology Conference*. Springer, 704–737.
 - [43] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. 2018. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.
 - [44] Srinath Setty, Justin Thaler, and Riad Wahby. 2023. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552. <https://eprint.iacr.org/2023/552>
 - [45] Srinath Setty, Justin Thaler, and Riad Wahby. 2024. Unlocking the lookup singularity with lasso. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 180–209.
 - [46] Srinath Setty, Justin Thaler, and Michael Zhu. 2026. Twist and Shout: Faster Memory Checking Arguments via One-Hot Addressing and Increments. In *Advances in Cryptology – CRYPTO 2026 (Lecture Notes in Computer Science, Vol. 16808)*. Springer, 267–296. doi:10.1007/978-3-032-35424-2_10
 - [47] Adi Shamir. 1992. IP = PSPACE. *Journal of the ACM (JACM)* 39, 4 (1992), 869–877.
 - [48] Succinct. 2025. SP1 Hypercube: Proving Ethereum in Real-Time. <https://blog.succinct.xyz/sp1-hypercube/>.
 - [49] Supranational. 2024. blst: Multilingual RSA-and ECDSA-capable BLS12-381 signature library. <https://github.com/supranational/blst>.
 - [50] The Irreducible team. 2023. Binius: Rust implementation of SNARK over towers of binary fields. <https://gitlab.com/IrreducibleOSS/binius>.
 - [51] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *CRYPTO*.
 - [52] Justin Thaler. 2022. Proofs, arguments, and zero-knowledge. *Foundations and Trends in Privacy and Security* 4, 2–4 (2022), 117–660.
 - [53] Andrei Toom. 1963. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. In *Soviet Mathematics Doklady, No 3*. 714–716.
 - [54] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. 2013. A hybrid architecture for verifiable computation. In *2013 IEEE Symposium on Security and Privacy*.
 - [55] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. 2017. Full accounting for verifiable outsourcing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.
 - [56] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*.
 - [57] Yuanju Wei, Kaixuan Wang, Binwu Xiang, Xinxuan Zhang, Hailong Wang, Yi Deng, Xudong Zhu, and Li Lin. 2025. Packed Sumcheck over Fields of Small Characteristic with Application to Verifiable FHE. Cryptology ePrint Archive, Paper 2025/719. <https://eprint.iacr.org/2025/719>
 - [58] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover

- Computation. In *Advances in Cryptology – CRYPTO 2019*.
- [59] XopMC. [n.d.]. secp256k1-x64-avx: very fast (NOT SECURE) implementation of arithmetic on curve secp256k1 on x86_64. <https://github.com/XopMC/secp256k1-x64-avx>. README benchmark: Montgomery mul = 49 cycles/op on Intel Core i7-6700 (Skylake); 93 cycles/op on Intel Core i3-2328M (Sandy Bridge).
- [60] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 859–876.
- [61] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy*.
- [62] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster Verifiable RAM With Program-Independent Preprocessing. In *2018 IEEE Symposium on Security and Privacy*.

A Baseline prover algorithms

The linear-time prover [51, 54] stores every factor’s Boolean evaluation table, binds adjacent entries after each challenge, and halves the tables per round. The logarithmic-space prover [19] stores no bound tables and reconstructs partial evaluations with an equality-weighted input pass in each round. The cross-product prover [10] handles several rounds per pass using a coefficient-domain tensor indexed by one Boolean prefix per factor. Our prover instead stores one product polynomial on $\mathbb{U}_d^0$, improving the degree dependence as analyzed in Section 5.

B Selected field-arithmetic details

This appendix makes the small-big multiplication primitive from Section 3.1 self-contained. Let p satisfy $2^{np-1} < p < 2^{np}$, and let a field element be stored in Montgomery form as $a' = aR \bmod p$ using N 64-bit limbs. For a 64-bit unsigned integer b , first compute the integer product $c = a'b$ using N native multiplications. The result has at most $N + 1$ limbs and satisfies $0 \leq c < p \cdot 2^{64}$.

Precompute $\mu = \lfloor 2^{np+64}/(2p) \rfloor$. The following specialized Barrett reduction returns $c \bmod p$:

- (1) Set $\hat{c} = \lceil c/2^{np} \rceil$ and $m = \lfloor \hat{c}\mu/2^{64} \rfloor$.
- (2) Set $r = c - 2mp$. If $r < 0$, replace r by $r + 2p$.
- (3) Choose $k \in \{0, 1, 2, 3\}$ such that $kp \leq r < (k+1)p$, and return $r - kp$.

Thus $c \bmod p = abR \bmod p$ is in Montgomery form.

The full version proves that the quotient estimate differs from $\lfloor c/(2p) \rfloor$ by at most one, so the corrected remainder lies in $[0, 4p)$ and the final subtraction is correct.

The complete multiplication therefore costs N native multiplications for $a'b$ and $N + 1$ for the reduction, for a total of $2N + 1$, compared with $2N^2 + N$ for a generic Montgomery multiplication. The same bound explains delayed reduction for a linear combination $T = \sum_i c_i a_i$: if $\sum_i c_i < 2^{64}$, then $T < p \cdot 2^{64}$ and one final application of the procedure suffices. Larger coefficient sums require either a wider accumulator or reductions in batches, as discussed in Section 3.3.

C Univariate skip parameter bounds

Gruen’s construction extrapolates over a multiplicative subgroup of a small base field [28]. We instead use the symmetric integer

domain D_m from Section 7 inside one large prime field. Its small points and symmetry limit coefficient growth and let the prover delay reduction.

The prover constructs the packed message by direct $O(m^2)$ interpolation, which is inexpensive for small m because most products use small integer coefficients. The verifier uses linear-time dot-product and Horner evaluations; Section 7 gives the exact costs. The remaining constraint is the width of the prover’s packed accumulator. For an integer shift s from the left endpoint of an m -point window, the shifted Lagrange coefficients are

$$\alpha_i(s) = (-1)^{m-1-i} \binom{s}{i} \binom{s-i-1}{m-1-i}.$$

Define $\Lambda = \max_s \sum_{i=0}^{m-1} |\alpha_i(s)|$ over all target shifts. If each packed value satisfies $|v_i| \leq B$, then

$$\left| \sum_i \alpha_i(s) v_i \right| \leq B\Lambda.$$

Thus a signed 128-bit accumulator for 64-bit inputs is safe whenever $B\Lambda < 2^{127}$.

For the odd symmetric domain $D = \{-K, \dots, K\}$ and the degree-2 target window $\{-2K, \dots, 2K\}$, exact integer evaluation gives $\Lambda_{14} \approx 2^{62.6}$, $\Lambda_{15} \approx 2^{67.3}$, and $\Lambda_{16} \approx 2^{72.0}$. Under worst-case full-width 64-bit inputs, $K \leq 14$ is therefore a conservative choice for this setting. Higher packed degrees use a larger target window and require recomputing Λ before selecting the window size.

Open Science

We provide an artifact containing the prover implementations and Jolt integration; benchmark harnesses and configurations for field arithmetic, multilinear products, Spartan Outer, Shout, and end-to-end proving; analysis scripts and processed values for every table and figure; and ARTIFACT.md with the toolchain, build and experiment commands, and expected outputs. Our evaluation uses no external datasets, models, or private inputs. The full version gives the complete inventory and reproduction procedure.

Ethical Considerations

This work improves sum-check proving and evaluates the implementations on benchmark workloads. It involves no human participants, user data, or vulnerability discovery or exploitation. Faster proving can reduce the computation, memory, and energy required by verifiable and privacy-preserving applications, but also lower the cost of proofs for undesirable applications. Our techniques add no capabilities and leave protocol security unchanged; we judge that their benefits outweigh this limited dual-use risk.

Generative AI Usage

We used various Google Gemini, Claude, and OpenAI GPT models for editorial revisions, author-specified appendix exposition, implementation, and benchmarking. The authors reviewed and edited all output, checked its technical content, and reproduced the results.