The Sum-Check Protocol over Fields of Small Characteristic

Justin Thaler[*]

Abstract

The sum-check protocol of Lund, Fortnow, Karloff, and Nisan underlies SNARKs with the fastest known prover. In many of its applications, the prover can be implemented with a number of field operations that is linear in the number, n, of terms being summed.

We describe an optimized prover implementation when the protocol is applied over an extension field of a much smaller base field. The rough idea is to keep most of the prover's multiplications over the base field (at the cost of performing more *total* field multiplications).

When the sum-check protocol is applied to a product of polynomials that all output values in the base field, our algorithm reduces the number of extension field operations by multiple orders of magnitude. In other settings, our improvements are more modest but nonetheless meaningful.

In SNARK design, the sum-check protocol is often combined with a polynomial commitment scheme, which are growing faster, especially when the values being committed are small. These improved commitment schemes may render the sum-check prover the overall bottleneck, which our results help to mitigate.

1 Introduction

The sum-check protocol [LFKN90] underpins SNARKs with the fastest known prover. It is especially effective at forcing the prover to perform useful work, while minimizing the amount of data to which the prover must *cryptographically commit*. This alleviates a key bottleneck for SNARK provers, which is the cost (both in time and space) of computing cryptographic commitments to large vectors of field elements.

For an ℓ-variate polynomial g over field $\mathbb{F}$, the sum-check protocol forces the prover to sum up g's evaluations over $\{0,1\}^\ell$.[1] That is, the sum-check protocol is an interactive proof for computing

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

Throughout this manuscript, let $n = 2^\ell$ denote the number of terms in this sum. Suppose that g can be expressed as a product of d multilinear polynomials $p_1, \ldots, p_d$,

$$g(x) = \prod_{i=1}^{d} p_i(x). \tag{2}$$

Also, suppose that for each p_i, the prover is provided $p_i(x)$ for all inputs $x \in \{0,1\}^\ell$. For constant values of d, it is well-known that the prover in the sum-check protocol can be implemented with $O(n)$ field operations [CTY11, Tha13], which is within a constant factor of the time required simply to compute Equation (1) term-by-term.

[*]a16 crypto research and Georgetown University

[1]The sum-check protocol can more generally sum up g's evaluations over any product set H^ℓ for some $H \subseteq \mathbb{F}$.

SNARKs over large and small fields. A popular viewpoint in SNARK design today is that one should endeavor to work over a "small" field $\mathbb{F}$ for performance reasons.[2] This is because a given number m of field operations are much faster if those operations occur over, say, a 32-bit field rather than a 256-bit field. Similarly, hashing a vector of m field elements can be faster if all field elements are 32 bits rather than 256.

Most SNARKs are obtained by combining a protocol called a *polynomial IOP* with a cryptographic protocol called a *polynomial commitment scheme* to obtain an interactive succinct argument, and then applying the Fiat-Shamir transformation to render it non-interactive. Hashing-based polynomial commitment schemes such as FRI [BBHR18] have become popular, in part because they enable working over smaller fields than elliptic-curve-based polynomial commitment schemes.

In fact, whether or not it makes sense to work over a small field depends on several factors. For example, SNARK statements that are natively defined over a large prime-order field (such as proving knowledge of elliptic-curve-based digital signatures authorizing blockchain transactions) are most efficiently proven by working over that field. In addition, hashing-based commitment schemes are actually slower than curve-based ones if the hash function used is a slow "SNARK-friendly" hash function such as Poseidon [GKR$^+$21], and the prover only needs to commit to "small" values (say, in $\{0, 1, \ldots, 2^{20}\}$).[3] This is indeed the case in state-of-the-art sum-check-based SNARKs such as Lasso and Jolt [STW23, AST23].

Fortunately, new work by Diamond and Posen [DP23b] gives a substantially faster hashing-based commitment scheme for small values, and integrates the scheme with sum-check-based polynomial IOPs to give very fast SNARKs for standard, fast hash functions like Keccak. Diamond and Posen's SNARKs work over the field $\mathsf{GF}[2^{128}]$, and their prover's commitment costs are low enough that the sum-check protocol is likely to be the prover bottleneck. Motivated by these developments, our goal in this manuscript is to optimize the sum-check protocol when it is applied over fields of small characteristic.

Sum-check-based SNARKs over small fields. In current linear-time implementations of the sum-check prover, about half of the field multiplications performed by the prover occur over the extension field (i.e., both operands in the multiplication are extension-field elements). This is because, to ensure adequate soundness error in the sum-check protocol, random field elements $r_1, \ldots, r_\ell$ should be chosen in each round of the protocol from a field of size (at least) 2^{128}. So if the polynomial g being summed is defined over a small base field, $r_1, \ldots, r_\ell$ should be chosen from an extension field.

For example, if using a degree-4 extension of a 32-bit base field, then extension field multiplications are roughly 9-16 times more expensive than base field multiplications. If half of the multiplications performed are over the extension field, then the prover will be at least 5-8 times slower than if all multiplications were over the base field. In other words, even if the sum-check protocol contributes just 13% of the prover's work for a SNARK defined over a large field, they may become the dominant prover cost when the same SNARK is applied over a degree-4 extension field of a 32-bit base field. The situation is amplified further for larger-degree extensions. In particular, Diamond and Posen [DP23b] make particularly heavy use of degree-8 extensions (where the base field is $\mathsf{GF}[2^{16}]$ and the extension field is $\mathsf{GF}[2^{128}]$) and degree-128 extensions (where the base field is $\mathsf{GF}[2]$). Motivated by projects that seek more than 128 bits of security, in this manuscript we consider extension degrees up to 256.

In some contexts base field multiplications can even be considered so cheap relative to extension field multiplications that they are essentially *free*. One example is $\mathsf{GF}[2]$, as multiplying any field element by 0 or 1 is essentially trivial (the result is either 0 or x).

Repetition, and why it should be avoided. Another option would be to choose $r_1, \ldots, r_\ell$ from the base field, and apply parallel or sequential repetition. But sequential repetition does not increase security when combined with the Fiat-Shamir transformation to render the protocol non-interactive. The same goes for parallel repetition, at least if naively implemented, unless the number of repetitions is very large. Specifically,

[2]In order to achieve λ bits of security, deployed SNARKs work over a large field (size at least 2^λ) for at least some parts of the protocol. Hence, we personally prefer to view all SNARKs as working over a large field, with the question being whether the *characteristic* of that field is large or small.

[3]See for example `https://hungrycatsstudio.github.io/posts/benching-pcs/`.

when applying parallel repetition followed by Fiat-Shamir to an ℓ-round interactive protocol, $\ell \cdot k$ repetitions are necessary to amplify λ/k bits of security to λ bits of security. In the context of sum-check, this results in $O(nk \log n)$ base field operations for the prover, which is typically worse than the number of base field operations achieved by existing work on linear-time sum-check provers [CTY11, Tha13] (depending on details of the field extension and the algorithm used to perform multiplications in the extension field, this number is typically $O(nk^{1.58496\cdots})$ or perhaps $O(nk^2)$, where k is the degree of the field extension, see Section 2.1 for details.). Indeed, $\log n$ is typically 20 or larger.

We strongly recommend *not* to combine parallel repetition with the Fiat-Shamir transformation, due to both its introduction of subtle security issues and performance degradation. Regardless, our work improves over both the $O(nk \log n)$ base field operations from parallel repetition, as well as over the existing linear-time sum-check prover algorithms.

Our results. We first describe two algorithms from prior works for implementing the sum-check prover [CTY11, Tha13, CMT12], carefully optimizing them for the extension-field context we consider. Surprisingly, we point out that the second algorithm, which is asymptotically and concretely slower than the first algorithm in the standard "large field" setting, is actually cheaper than the first when base field multiplications (and base-field-times-extension-field multiplications) are *much* cheaper than extension-field multiplications.

We then present our main technical contribution: a third algorithm that performs almost no extension field multiplications in early rounds of the sum-check protocol (at the cost of performing quite a large number of base-field multiplications).

In later rounds, the costs of this new, third algorithm start to exceed those of the first two. Hence, after enough rounds have passed, it makes sense to "switch" from the third algorithm to one of the first two. We calculate the optimal sequence of switches, and compare the costs to prior work alone.

Generally speaking, the cheaper base field multiplications are relative to extension-field multiplications, the stronger our results. When it is reasonable to consider base field multiplications (and base-field-times-extension-field multiplications) as "free" relative to extension-field multiplications, our results speed up the sum-check prover by multiple orders of magnitude (Section 5).

When the relative costs obey those of Karatsuba's algorithm, our improvements are considerably more modest but can still be a factor of close to five (see Section 6).

Even modest improvements to sum-check prover time are meaningful. This is because recent SNARKs only require the prover to commit to base-field elements [STW23, AST23], and polynomial commitment schemes are growing extremely fast when committing only to such elements [DP23b]. This may render the sum-check protocol the prover bottleneck in these SNARKs, and our results help mitigate this bottleneck.

In Section 7 we describe extensions of our results to more general settings that arise in some of these recent SNARKs.

2 Preliminaries

2.1 Background on extension fields

Let $\mathbb{B}$ be a base field and $\mathbb{F}$ an extension of $\mathbb{B}$ of degree k. $\mathbb{F}$ is a k-dimensional vector space over $\mathbb{B}$, and elements of $\mathbb{F}$ are often represented relative to some basis $\beta_1, \ldots, \beta_k$ of this vector space. That is, an element $x \in \mathbb{F}$ can represented by $(\alpha_1, \ldots, \alpha_k)$ where

$$x = \sum_{i=1}^{k} \alpha_i \cdot \beta_i.$$

A popular basis to use when representing extension fields is the *monomial basis.* Indeed, the extension field $\mathbb{F}$ can be viewed as the set of all degree-k polynomials over the base field $\mathbb{B}$, modulo a degree-k irreducible polynomial over $\mathbb{B}$. In this view, an extension field element's representation under the standard monomial basis is simply its coefficients when viewed as such a polynomial.

2.1.1 Tower fields vs. the standard monomial basis

Suppose $k = 2^z$ for some integer $z > 0$. Then a degree-k extension field $\mathbb{F}$ of $\mathbb{B}$ is said to be constructed as a *tower* extension if it is constructed from $\mathbb{B}$ by first constructing a degree-2 extension $\mathbb{B}'$, and then constructing a degree-2 extension $\mathbb{B}''$ of $\mathbb{B}'$ (which is a degree-4 extension of $\mathbb{B}$), and so forth for z iterations. This leads to a basis for $\mathbb{F}$ in which, for any integer $j > 0$, the first j basis elements are in the degree-j extension field obtained after j iterations of the tower construction. Particularly fast and elegant tower field constructions are known for fields of characteristic two [Wie88, FP97].

There are (at least) two benefits to using a tower basis that are extremely important to applications of the sum-check protocol in SNARK design [DP23b].

Subfield elements are compressed. Let $\mathbb{B}'$ be a subfield arising in the tower construction, i.e., $\mathbb{B}'$ is a degree-j extension of $\mathbb{B}$ for some $j < k$ with j a power of two. Then one can identify any element $x \in \mathbb{B}'$ via just j elements of $\mathbb{B}$ (specifically, the first j coefficients of x in the tower basis, as all other coefficients are zero).

Information-theoretically, representing elements of $\mathbb{B}'$ with j base-field elements is also possible over the standard monomial basis, but this comes at a major cost: embedding $\mathbb{B}'$ into $\mathbb{F}$ becomes expensive. That is, unlike in the tower construction, it is *not* the case that the "compressed" representation of $\mathbb{B}'$ elements is the same as its representation in the standard monomial basis for $\mathbb{F}$.

Lower memory consumption for subfield elements can have a very large effect on performance: it affects bandwidth usage for hardware acceleration, and cache efficiency in CPUs.

Fast base-field-by-subfield (or sub-field-by-subfield) multiplication. The above also ensures that multiplying an element of $\mathbb{B}$ by an element of $\mathbb{B}'$ costs only j base-field multiplications, rather than $k = 2^z$ of them. Such fast multiplication of elements of the base field and subfields is not supported by the standard monomial basis for $\mathbb{F}$.

In this manuscript, for simplicity we present our algorithms assuming that the sum-check protocol is applied to an ℓ-variate polynomial g that is a product of multilinear polynomials $p_1, \ldots, p_d$, each of which maps $\{0,1\}^\ell$ to the base field $\mathbb{B}$. For tower fields, due to fast sub-field-by-sub-field multiplication, our algorithms also "automatically" improve on prior work under the weaker assumption that different p_i's map $\{0,1\}^\ell$ to different subfields of $\mathbb{F}$. See Section 7 for other extensions and applications of our results.

2.1.2 Multiplication algorithms for extension fields

Karatsuba's algorithm for tower field multiplication. Let $\mathbb{B}$ be a base field and let $\mathbb{F}$ be a degree-2 extension of $\mathbb{B}$. Using Karatsuba's algorithm, multiplying two elements of $\mathbb{F}$ can be done with roughly three base-field multiplications (and several addition operations, followed by reducing a degree-two polynomial modulo another degree-two polynomial). In general, doubling the extension degree roughly triples the cost of a multiplication in the extension field. Asymptotically, this means that multiplications in degree-k extension field $\mathbb{F}$ are roughly $O(k^{\log_2(3)}) = O(k^{1.58496\cdots})$ times more expensive than multiplications in the base field $\mathbb{F}$.[4]

Karatsuba's algorithm for non-tower bases. Karatsuba's algorithm applies in a different way over non-tower bases. Specifically, given two elements of $\mathbb{F}$ represented in the standard monomial basis, one can use Karatsuba's algorithm to multiply the two polynomials in $O(k^{1.58496\cdots})$ time (and then perform a single reduction modulo an irreducible polynomial of degree k).

[4]Optimized field multiplication algorithms have been studied for extension degrees k that are not a power of two. For example, multiplications in extension fields of degree $k = 3$ can be performed with 5 base field multiplications via the Toom-Cook algorithm. For extension degree $k = 5$, extension field multiplications can be done with nine base field multiplications (with over a hundred base field additions) [EMGI11], or with fourteen base field multiplications and a smaller number of additions.

Multiplication of extension field elements tends to be faster when using the standard monomial basis rather than a tower basis (though use of the monomial basis lacks the benefits discussed in Section 2.1.1). In particular, some hardware supports certain finite field arithmetic as a primitive operation when using the standard monomial basis (e.g., Intel's Galois Field instruction set (GFNI) has native support for $\mathsf{GF}[2^8]$ multiplication) and on this hardware multiplication via the monomial basis can be over an order of magnitude faster. However, with FPGAs, the difference is much smaller, with recent estimates indicating that multiplications in the monomial basis uses only 20% fewer resources than tower bases [DP23a].[5]

2.1.3 Notation for costs of field multiplications

Let bb denote the cost of multiplying two base field elements, $\mathsf{be} \approx k \cdot \mathsf{bb}$ denote the cost of multiplying a base field element by an extension field element, and ee denote the cost of applying two extension field elements. As discussed above, via Karatsuba's algorithm, if k is a power of two then $\mathsf{ee} \approx k^{1.5849} \cdot \mathsf{bb}$. Abusing notation, we also use bb as shorthand for base-base multiplications, be for base-extension multiplications, and ee for extension-extension.

When to model bb and be multiplications as "free", relative to ee multiplications. When the base field is $\mathbb{B} = \mathsf{GF}[2]$, multiplying a base field element b by an extension field e element is essentially free, as $b \cdot e$ is 0 if $b = 0$ and is e if $b = 1$. Hence, it is *not* the case that $\mathsf{ee} \approx k^{1.5849} \cdot \mathsf{bb}$ (as $\mathsf{bb} = 0$). The goal as an algorithm designer in this case is to minimize the number of extension field multiplications.

There are other situations where it may be reasonable to consider bb and be multiplications as much less expensive than ee multiplications (i.e., by more than a factor of $k^{1.5849}$ and $k^{0.5849}$ respectively). One example is when base-field multiplication is a primitive operation on relevant hardware (e.g., Intel's GFNI primitive instruction for GF$[2^8]$ multiplication). In this case, bb multiplications may be so cheap that the extra work performed by Karatsuba's algorithm for ee multiplication (outside of the $O(k^{1.5849})$ bb multiplications) could be a dominant cost.

The cheaper that bb and be multiplications are relative to ee multiplications, the more significant our improvements over prior work. This is because our algorithms perform a lot of bb and be multiplications, in order to reduce the number of ee multiplications.

Other considerations. As indicated above, we will be interested not only in base fields $\mathbb{B}$ that are of prime size, but also in base fields that are themselves prime power size. For example, one may want to view $\mathsf{GF}[2^{128}]$ as, say, a degree-16 extension of $\mathsf{GF}[2^8]$ rather than a degree-128 extension field of $\mathbb{B}$ because our algorithms assume that the sum-check protocol is applied to a product of ℓ-variate polynomials that map $\{0,1\}^\ell$ to $\mathbb{B}$ (though see Section 7 for weakenings of this assumption). In some settings, this will indeed hold for $\mathbb{B} = \mathsf{GF}[2]$, but in others it will not.

2.2 Background on the sum-check protocol

As per Equations (1) and (2), let us consider applying the sum-check polynomial to compute

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

where g has degree at most d in each variable. A complete description of the sum-check protocol is in the codebox below.

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 $c \in \{0, 1, \ldots, d\}$ suffices to uniquely specify s_j. Here, we are assuming that the characteristic of the field over which g is defined

[5]GFNI instructions also benefit tower field constructions. Very roughly speaking, one can build a tower field over the base field $\mathsf{GF}[2^8]$, using the monomial basis for the base field, and building a tower basis on top of that. GFNI also has primitive instructions to compute affine transformations in $\mathsf{GF}[2^8]$, allowing fast conversation between the monomial basis for $\mathsf{GF}[2^8]$ and a tower basis for $\mathsf{GF}[2^8]$ over $\mathsf{GF}[2]$.

is at least d. If this is not the case, then one should replace the set $\{0, 1, \ldots, d\}$ with a set $\{0, 1, x_1, \ldots, x_{d-1}\}$ for any convenient points $x_1, \ldots, x_{d-1}$ in the field. For example, if $\mathbb{F} = \mathsf{GF}[2^{128}]$ is constructed as a tower field (see Section 2.1), then it makes sense to choose $x_1, \ldots, x_{d-1}$ to all reside in the subfield $\mathsf{GF}[2^k]$ where k is the smallest power of two greater than $\log(d-1)$.

Accordingly, in round j, the prover must compute

$$s_j(c) = \sum_{x \in \{0,1\}^{\ell-j}} g(r_1, \ldots, r_{j-1}, c, x), \tag{3}$$

for all $c \in \{0, 1, \ldots, d\}$. We will ignore the cost of all additions in our accounting below, as well as multiplications by 2.

Description of Sum-Check Protocol applied to the polynomial g of degree at most d in each variable (description taken from [Tha22, Chapter 4]). In this paper, we assume g is defined over a base field $\mathbb{B}$ and that $\mathbb{F}$ is an extension field of $\mathbb{B}$.

- At the start of the protocol, the prover sends a value C_1 claimed to equal the value defined in Expression (4).
- In the first round, $\mathcal{P}$ sends the univariate polynomial $s_1(X_1)$ claimed to equal

  $$\sum_{(x_2,\ldots,x_\ell) \in \{0,1\}^{\ell-1}} g(X_1, x_2, \ldots, x_\ell).$$

  $\mathcal{V}$ checks that

  $$C_1 = s_1(0) + s_1(1),$$

  and that s_1 is a univariate polynomial of degree at most d, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_1 \in \mathbb{F}$, and sends r_1 to $\mathcal{P}$.
- In the jth round, for $1 < j < \ell$, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $s_j(X_j)$ claimed to equal

  $$\sum_{(x_{j+1},\ldots,x_\ell) \in \{0,1\}^{\ell-j}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_\ell).$$

  $\mathcal{V}$ checks that s_j is a univariate polynomial of degree at most d, and that $s_{j-1}(r_{j-1}) = s_j(0) + s_j(1)$, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_j \in \mathbb{F}$, and sends r_j to $\mathcal{P}$.
- In Round ℓ, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $s_\ell(X_\ell)$ claimed to equal

  $$g(r_1, \ldots, r_{\ell-1}, X_\ell).$$

  $\mathcal{V}$ checks that s_ℓ is a univariate polynomial of degree at most d, rejecting if not, and also checks that $s_{\ell-1}(r_{\ell-1}) = s_\ell(0) + s_\ell(1)$.
- $\mathcal{V}$ chooses a random element $r_\ell \in \mathbb{F}$ and evaluates $g(r_1, \ldots, r_\ell)$ with a single oracle query to g. $\mathcal{V}$ checks that $s_\ell(r_\ell) = g(r_1, \ldots, r_\ell)$, rejecting if not.
- If $\mathcal{V}$ has not yet rejected, $\mathcal{V}$ halts and accepts.

Theorem 1. *The sum-check protocol is a perfectly complete protocol for computing*

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

with soundness error at most $\ell \cdot d/|\mathbb{F}|$. That is, an honest prover will always pass the verifier's checks, and a dishonest prover will pass the verifier's checks with probability at most $\ell \cdot d/|\mathbb{F}|$.

Unless stated otherwise, when applying the sum-check protocol to an ℓ-variate polynomial g, we assume throughout that g is defined over the base field $\mathbb{B}$. In particular, we assume that $g(x) \in \mathbb{B}$ for all $x \in \{0,1\}^\ell$.

For expository purposes, for each of the sum-check prover algorithms we describe, we begin by considering Equation (2) in the case that $d = 2$. In this case, for readability, let us replace p_1 with p and p_2 with q, so that the goal of the sum-check protocol is to compute

$$\sum_{x\in\{0,1\}^\ell} p(x)\cdot q(x). \tag{4}$$

2.3 A key lemma for multilinear polynomials

The following lemma will be used throughout this note.

Lemma 1. *Suppose $p\colon \mathbb{F}^\ell \to \mathbb{F}$ is an ℓ-variate multilinear polynomial over $\mathbb{F}$. Then for any input $(r_1, x') \in \mathbb{F}\times\mathbb{F}^{\ell-1}$,*

$$p(r_1, x') = r_1 \cdot p(1, x') + (1 - r_1)\cdot p(0, x'). \tag{5}$$

Proof. The right hand side of Equation (5) is clearly a multilinear polynomial in $x = (r_1, x')$, and agrees with $p(x)$ for all $x = (r_1, x') \in \{0,1\}^\ell$. Hence it must equal $p(x)$, as $\{0,1\}^\ell$ is an interpolating set for multilinear polynomials. That is, if p and q are two multilinear polynomials satisfying $p(x) = q(x)$ for all $x \in \{0,1\}^\ell$, then p and q are the same polynomial.

□

Lagrange basis polynomials and a generalization of Lemma 1. For any $S \in \{0,1\}^\ell$, let

$$\chi_S(x) = \prod_{i=1}^{\ell}(x_i S_i + (1-x_i)(1-S_i))$$

denote the S'th multilinear Lagrange basis polynomial. For example, if $\ell = 4$ and $S = (0,1,1,0)$, then $\chi_S(x) = (1-x_i)x_2x_3(1-x_4)$. We have the following generalization of Lemma 1

Lemma 2. *Suppose $p\colon \mathbb{F}^\ell \to \mathbb{F}$ is an ℓ-variate multilinear polynomial over $\mathbb{F}$. Then then for any input $((r_1,\ldots,r_i), x') \in \mathbb{F}^i \times \mathbb{F}^{\ell-i}$,*

$$p(r_1,\ldots,r_i,x') = \prod_{S\subseteq i}\chi_S(r_1,\ldots,r_i)\cdot p(S,x'). \tag{6}$$

Proof. The right hand side of Equation (6) is a multilinear polynomial in $x = (r_1,\ldots,r_i,x')$, and agrees with $p(x)$ for all $x = (r_1,\ldots,r_i,x') \in \{0,1\}^\ell$. Hence it must equal $p(x)$, as $\{0,1\}^\ell$ is an interpolating set for multilinear polynomials. □

The equality function and its multilinear extension. Let $\widetilde{\mathsf{eq}}_\ell\colon \mathbb{F}^\ell\times\mathbb{F}^\ell \to \mathbb{F}$ be the following multilinear polynomial:

$$\widetilde{\mathsf{eq}}_\ell(x,y) = \prod_{j=1}^{\ell}\left(x_j y_j + (1-x_j)(1-y_j)\right).$$

$\widetilde{\mathsf{eq}}_\ell$ is the unique multilinear polynomial satisfying, for all $x, y \in \{0,1\}^\ell$,

$$\widetilde{\mathsf{eq}}_\ell(x,y) = \begin{cases} 1 \text{ if } x = y \\ 0 \text{ otherwise.}\end{cases}$$

That is, $\widetilde{\mathsf{eq}}_\ell$ is the so-called multilinear extension of the equality function over $\{0,1\}^\ell\times\{0,1\}^\ell$. Note that for any $S \in \{0,1\}^\ell$, $\widetilde{\mathsf{eq}}_\ell(S,y) = \chi_S(y)$. We omit the subscript ℓ from $\widetilde{\mathsf{eq}}_\ell$ when ℓ is clear from context.

3 Existing algorithms: Algorithms 1 and 2

3.1 Algorithm 1

3.1.1 Description when $d = 2$

Consider applying the sum-check polynomial to $g(x) = p(x) \cdot q(x)$ per Equation (4). The known linear-time sum-check prover [CTY11, Tha13] operates as follows. The prover maintains two arrays, say A and B, which initially store all evaluations of p and q over $\{0,1\}^\ell$. We will index entries of A and B by $x \in \{0,1\}^\ell$, so that at initialization, $A[x]$ stores $p(x)$ and $B[x]$ stores $q(x)$. In each round, the size of the arrays will halve.

Round 1. Given the contents of A and B upon initialization, the prover can compute $s_1(0)$ and $s_1(1)$ with $n = 2^\ell$ bb multiplications in total. Indeed,

$$s_1(0) = \sum_{x \in \{0,1\}^{\ell-1}} p(0,x) \cdot q(0,x) = \sum_{x \in \{0,1\}^{\ell-1}} A(0,x) \cdot B(0,x), \tag{7}$$

and similarly for

$$s_1(1) = \sum_{x \in \{0,1\}^{\ell-1}} p(1,x) \cdot q(0,x) = \sum_{x \in \{0,1\}^{\ell-1}} A(1,x) \cdot B(1,x). \tag{8}$$

How does the prover compute $s_1(2)$? By Lemma 1,

$$\begin{aligned} s_1(2) &= \sum_{x \in \{0,1\}^{\ell-1}} p(2,x) \cdot q(2,x) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} \left((1-2) \cdot p(0,x) + 2 \cdot p(1,x)\right) \cdot \left((1-2) \cdot q(0,x) + 2 \cdot q(1,x)\right). \end{aligned} \tag{9}$$

Since we are ignoring the cost of additions and multiplications by two, $s_1(2)$ can be computed with $n/2$ bb multiplications. Indeed, $((1-2) \cdot p(0,x) + 2 \cdot p(1,x))$ is a base field element that can be computed via additions and multiplications by two, as is $((1-2) \cdot q(0,x) + 2 \cdot q(1,x))$, and the results can be multiplied together with one base-field multiplication.

After the verifier selects $r_1 \in \mathbb{F}$, the prover updates the arrays A and B as follows. For each $x \in \{0,1\}^{\ell-1}$, the prover sets

$$A[x] \leftarrow (1-r_1)A[0,x] + r_1 A[1,x] = A[0,x] + r_1(A[1,x] - A[0,x])$$

and

$$B[x] \leftarrow (1-r_1)B[0,x] + r_1 B[1,x] = B[0,x] + r_1(B[1,x] - B[0,x]).$$

Updating both arrays costs n be multiplications in total ($n/2$ per array). By Lemma 1, at the end of the update, for each $x \in \{0,1\}^{\ell-1}$, $A[x] = p(r_1, x)$ and $B[x] = q(r_1, x)$.

Round 2. Given the contents of the updated arrays, the prover can compute $s_2(0)$ and $s_2(1)$ with $n/4$ ee multiplications in total, since

$$s_2(0) = \sum_{x \in \{0,1\}^{\ell-2}} p(r_1, 0, x) \cdot q(r_1, 0, x) = \sum_{x \in \{0,1\}^{\ell-2}} A[0,x] \cdot B[0,x]$$

and

$$s_2(1) = s_1(r_1) - s_2(0).$$

By Lemma 1,

$$s_2(2) = \sum_{x\in\{0,1\}^{\ell-2}} p(r_1, 2, x) \cdot q(r_1, 2, x)$$

$$= \sum_{x\in\{0,1\}^{\ell-2}} ((1-2)\cdot p(r_1,0,x) + 2\cdot p(r_1,1,x)) \cdot ((1-2)\cdot q(r_1,0,x) + 2\cdot q(r_1,1,x)) \tag{10}$$

$$= \sum_{x\in\{0,1\}^{\ell-2}} ((1-2)\cdot A[0,x] + 2\cdot A[1,x]) \cdot ((1-2)\cdot B[0,x] + 2\cdot B[1,x]) \,.$$

Hence, $s_2(2)$ can be computed in $n/4$ ee multiplications.

After the verifier chooses $r_2 \in \mathbb{F}$, the prover updates A and B as follows. For each $x \in \{0,1\}^{\ell-2}$, the prover applies the update:

$$A[x] \leftarrow (1-r_2)A[0,x] + r_2A[1,x] = A[0,x] + r_2(A[1,x] - A[0,x])$$

and

$$B[x] \leftarrow (1-r_2)B[0,x] + r_2B[1,x] = B[0,x] + r_2(B[1,x] - B[0,x]),$$

thereby ensuring via Lemma 1 that $A[x] = p(r_1, r_2, x)$ and $B[x] = q(r_1, r_2, x)$.

Round $i > 2$. Following the above blueprint from round 2, in each round $i > 2$, the prover ensures that at the start of round i, A and B respectively store $p(r_1, \ldots, r_{i-1}, x)$ and $q(r_1, \ldots, r_{i-1}, x)$ for all $x \in \{0,1\}^{\ell-i+1}$. Given these values, the prover can compute $s_i(0)$, $s_i(1)$, and $s_i(2)$ with $n/2^{i-1}$ ee multiplications in total. Here, $n/2^i$ ee multiplications are devoted to computing $s_i(0)$ (from which the value $s_i(1)$ can be derived, given $s_{i-1}(r_{i-1})$), and another $n/2^i$ are devoted to computing $s_i(2)$.

The prover can then update the two arrays with $n/2^{i-1}$ ee multiplications in total, ensuring that A and B respectively store $p(r_1, \ldots, r_i, x)$ and $q(r_1, \ldots, r_i, x)$ for all $x \in \{0,1\}^{\ell-i}$.

Total Algorithm 1 prover costs when $d = 2$. Across all ℓ rounds, the prover's work in Algorithm 1 is as follows:

$$((3n/2)\cdot \mathsf{bb} + n\cdot \mathsf{be}) + \sum_{i=2}^{\ell} 4n/2^i \cdot \mathsf{ee}$$
$$\leq (3n/2)\cdot \mathsf{bb} + n\cdot \mathsf{be} + 2n\cdot \mathsf{ee}.$$

Here, the first term is for computing the round 1 message $s_1(0)$, $s_1(1)$, and $s_1(2)$, and the following array update. The sum is for computing the round i message and array updates for all rounds $i \geq 2$.

3.1.2 Algorithm 1 for general degrees d

Algorithm 1 has a straightforward generalization to the case where $g(x) = p_1(x) \cdot p_2(x) \cdots p_d(x)$. The algorithm stores d arrays, with the j'th array at the end of round i storing the values $p_i(r_1, \ldots, r_i, x)$ for all $x \in \{0,1\}^{\ell-i}$.

Assuming that multiplication by field elements in $\{0, 1, \ldots, d\}$ are free, the cost in each round $i \geq 2$ of computing $s_i(0)$, $s_i(1), \ldots, s_i(d)$ is $d(d-1)$ ee multiplications. This is because $s_i(1)$ can be derived as $s_{i-1}(r_{i-1}) - s_i(0)$, while the other d evaluations of s_i can each be expressed as the product of d ee elements, one for each of the d arrays. The cost at the end of round i of updating all d arrays is $d\cdot n/2^i$ ee multiplications.

Hence, the cost of Algorithm 1 across all ℓ rounds is:

$$((d\cdot(d-1)\cdot n/2)\cdot \mathsf{bb} + (dn/2)\cdot \mathsf{be}) + \sum_{i=2}^{\ell} d^2n/2^i \cdot \mathsf{ee} \tag{11}$$

$$\leq ((d^2-1)n/2)\cdot \mathsf{bb} + (dn/2)\cdot \mathsf{be} + (d^2/2)\cdot n\cdot \mathsf{ee}. \tag{12}$$

In Expression (11), the expressions before the sum account for computing the round 1 message $s_1(0)$, $s_1(1)$, $\dots$ $s_1(d)$,[6] and the following array update. The sum in Expression (11) is for computing the round i message and array updates for all rounds $i \geq 2$.

3.2 Algorithm 2

3.2.1 The case of $d = 2$

A second known sum-check prover implementation, dating to work of Cormode, Mitzenmacher, and Thaler [CMT12], has the prover perform $O(2^\ell)$ field operations *per round* rather than in total.[7]

However, as we will show, most of these field operations are be operations rather than ee operations. Even for "dense" polynomials p (where $m = n$ and $\ell = \log n$), $m \log n$ be multiplications can be faster than $O(n)$ ee multiplications.

Round 1. Round 1 proceeds identically to Algorithm 1, with the prover computing $s_1(0)$, $s_1(1)$ and $s_1(2)$ with n bb multiplications in total.[8]

The difference from Algorithm 1 is that, after the verifier selects $r_1 \in \mathbb{F}$, the prover does *not* update the arrays A and B.

Round $i \geq 2$. In each round $i \geq 2$, the prover can compute $s_i(0)$, $s_i(1)$, and $s_i(2)$ as follows. For each $x \in \{0,1\}^{\ell-i}$ and $y \in \{0,1\}^{i-1}$, let

$$C[y,0,x] = \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \dots, r_{i-1}) \cdot p(y,0,x)$$

$$C[y,1,x] = \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \dots, r_{i-1}) \cdot p(y,1,x)$$

$$D[y,0,x] = \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \dots, r_{i-1}) \cdot q(y,0,x)$$

$$D[y,1,x] = \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \dots, r_{i-1}) \cdot q(y,1,x)$$

$$E[y,x] = \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \dots, r_{i-1}) \left(-p(y,0,x) + 2p(y,1,x)\right),$$

and

$$F[y,x] = -\widetilde{\mathsf{eq}}_{i-1}(y, r_1, \dots, r_{i-1}) \left(-q(y,0,x) + 2q(y,1,x)\right),$$

Lemma 2 implies that

$$\sum_{y \in \{0,1\}^{i-1}} C[y,0,x] = p(r_1, \dots, r_{i-1}, 0, x),$$

$$\sum_{y \in \{0,1\}^{i-1}} C[y,1,x] = p(r_1, \dots, r_{i-1}, 1, x),$$

$$\sum_{y \in \{0,1\}^{i-1}} D[y,0,x] = q(r_1, \dots, r_{i-1}, 0, x),$$

$$\sum_{y \in \{0,1\}^{i-1}} D[y,1,x] = q(r_1, \dots, r_{i-1}, 1, x),$$

$$\sum_{y \in \{0,1\}^{i-1}} E[y,x] = p(r_1, \dots, r_{i-1}, 2, x),$$

[6]Specifically, evaluating $s_1(i)$ requires $(d-1) \cdot n/2$ base field multiplications for any $i \in \{0, 1, \dots, d\}$. Here, $n/2$ is the number of terms in the sum defining s_1, see Equation (3).

[7]More precisely, the number of field operations performed by the prover in each round is linear in the *sparsity* m of p and q, meaning the number of inputs $x \in \{0,1\}^\ell$ for which $p(x) \cdot q(x) \neq 0$. However, we won't focus on sparse polynomials in this manuscript.

[8]For Algorithm 1, we stated a bound of $n + n/2$, but it is easy to see by inspection that computing $s_1(0)$ and $s_1(1)$ only require one bb multiplication per $x \in \{0,1\}^\ell$ such that $p(x) \cdot q(x) \neq 0$. Similarly, the $n/2$ term can be replaced with m.

and

$$\sum_{y\in\{0,1\}^{i-1}} F[y,x] = q(r_1,\ldots,r_{i-1},2,x).$$

Standard techniques enable the prover to use 2^{i-1} ee multiplications to compute $\widetilde{\mathsf{eq}}_{i-1}(y,r_1,\ldots,r_{i-1})$ for all $y\in\{0,1\}^{i-1}$, given $\widetilde{\mathsf{eq}}_{i-2}(y,r_1,\ldots,r_{i-1})$ for all $y\in\{0,1\}^{i-2}$ (see [Tha22, Lemma 3.8]). With these values in hand, the prover can compute all necessary values (that is, $C[y,0,x]$, $C[y,1,x]$, $D[y,0,x]$, $D[y,1,x]$, $E[y,x]$ and $F[y,x]$) with $3n$ be multiplications. At that point, the prover can compute $s_2(0)$, $s_2(1)$, and $s_2(2)$ with $n/2^i$ ee multiplications each, owing to the fact that

$$s_i(0) = \sum_{x\in\{0,1\}^{\ell-i}} p(r_1,\ldots,r_{i-1},0,x)\cdot q(r_1,\ldots,r_{i-1},0,x) \tag{13}$$

$$s_i(1) = \sum_{x\in\{0,1\}^{\ell-i}} p(r_1,\ldots,r_{i-1},1,x)\cdot q(r_1,\ldots,r_{i-1},1,x), \tag{14}$$

and

$$s_i(2) = \sum_{x\in\{0,1\}^{\ell-i}} p(r_1,\ldots,r_{i-1},2,x)\cdot q(r_1,\ldots,r_{i-1},2,x). \tag{15}$$

As an optimization, $s_i(1)$ can instead be derived as

$$s_i(1) = s_{i-1}(r_{i-1}) - s_i(0).$$

Algorithm 2 costs when $d=2$. With the aforementioned optimization, in each round i, the prover performs $2n$ be multiplications and $\left(2\cdot n/2^i + 2^{i-1}\right)$ ee multiplications.

Remark 1 (Cost comparison of Algorithm 1 vs. Algorithm 2)**.** *Algorithm 2 has fewer* ee *multiplications in each round i, until the final $\ell/2$ rounds (when the 2^{i-1} term for Algorithm 2 becomes dominant). After round $\ell/2$, one should "switch" from Algorithm 2 to Algorithm 1. That is, the $2n/2^i$* ee *multiplications in round i of Algorithm 2 is superior to the $4n/2^i$* ee *multiplications of Algorithm 1.*

The main downside of Algorithm 2 is that it also performs $2n$ be *multiplications per round. However, when* be *multiplications are "free" (e.g., when the base field is* $\mathsf{GF}[2]$*), then this downside is not relevant, and Algorithm 2 is preferable to Algorithm 1 until the last few rounds.*

Conceptually, for general degree bounds d, Algorithm 2 cuts out all of the $d/2^i$ many ee *multiplications that Algorithm 1 "spends" to update its d arrays in each round. This benefit is particularly significant for small degrees d, e.g., for $d=2$ this cuts the number of* ee *multiplications by a factor of 2. The price that Algorithm 2 pays for this is increasing the number of* be *multiplications from about $\Theta(dn)$ across all rounds, to $\Theta(dn)$* per round.

3.2.2 Algorithm 2 for general degrees d

Algorithm 2 has a straightforward generalization to the case where $g(x) = p_1(x)\cdot p_2(x)\cdots p_d(x)$. Assuming that multiplication by field elements in $\{0,1,\ldots,d\}$ are free, the cost of this algorithm in each round $i>1$ is:

$$(d+1)\cdot n\cdot \mathsf{be} + \left(d(d-1)n/2^i + 2^{i-1}\right)\cdot \mathsf{ee}.$$

Here, the 2^{i-1} term is the number of ee multiplications required to evaluate all $(i-1)$-variate Lagrange basis polynomials at $(r_1,\ldots,r_{i-1})$ via a standard memoization procedure (see [Tha22, Figure 3.3 and Lemma 3.8]). The $d(d-1)n/2^i$ term is the number of ee multiplications required to evaluate the degree-d analog of Equations (13)-(15). Specifically, there are $d+1$ equations, one for each of $s_i(0)$, $s_i(1)$, $\ldots$, $s_i(d)$. Each equation involves a sum over $n/2^i$ terms, with each term involving a product of d extension-field elements (such a product can be computed with $d-1$ ee multiplications). However, $s_i(1)$ can be derived as $s_i(1) = s_{i-1}(r_{i-1}) - s_i(0)$, reducing the effective number of equations that must be computed from $d+1$ to d.

4 An optimized prover for extension fields: Algorithm 3

Overview of the improvement. In the existing linear-time prover algorithm (Algorithm 1), starting in Round 2 the prover begins multiplying extension-field elements, because in round 1 the first variable of p and q was bound to a random extension field element r_1.

The main idea for optimization is that in Expression (10), although $p(r_1, x)$ and $q(r_1, x)$ for $x \in \{0,1\}^{\ell-1}$, is an extension field element, it is a simple expression of just four base-field elements, namely $p(0,x)$, $p(1,x)$, $q(0,x)$ and $q(1,x)$. In fact, it is a linear combination of the four products $p(0,x) \cdot q(0,x)$, $p(1,x) \cdot q(1,x)$, $p(0,x) \cdot q(1,x)$, $p(1,x) \cdot q(0,x)$. Moreover, the first two of these four products already had to be computed just to determine the correct answer. So it makes sense (for the first several rounds at least) not to treat $p(r_1, x)$ and $q(r_1, x)$ as arbitrary extension-field elements, but rather to compute them as the appropriate linear combination of (products of) base field elements, thereby keeping (almost) all arithmetic within the base field for the first several rounds. We call this Algorithm 3.

Below, we first work out the optimal combination of Algorithms 1 and 3, handling early rounds with Algorithm 3 before "switching over" to Algorithm 1. After that, we work out the optimal combination of all three algorithms in settings where using all three makes sense. In these settings, the prover's messages in the earliest rounds are computed with Algorithm 3, before "switching over" to Algorithm 2, and then finally switching over to Algorithm 1.

4.1 Details of Algorithm 3 when $d = 2$

The prover maintains an array C initially of length $n = 2^\ell$, indexed by $x \in \{0,1\}^\ell$. Initially, $C[x]$ contains $p(x) \cdot q(x)$. This initialization costs $n \cdot \mathsf{bb}$ multiplications.

Round 1. Given the contents of the array, the prover can compute $s_1(0)$ and $s_1(1)$ with no multiplications at all, since

$$s_1(0) = \sum_{x \in \{0,1\}^{\ell-1}} p(0,x) \cdot q(0,x) = \sum_{x \in \{0,1\}^\ell :\, x_1 = 0} C[x],$$

and

$$s_1(1) = \sum_{x \in \{0,1\}^{\ell-1}} p(1,x) \cdot q(1,x) = \sum_{x \in \{0,1\}^\ell :\, x_1 = 1} C[x].$$

How does the prover compute $s_1(2)$? Per Equation (9),

$$\begin{aligned} s_1(2) &= \sum_{x \in \{0,1\}^{\ell-1}} p(2,x) \cdot q(2,x) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} ((1-2) \cdot p(0,x) + 2 \cdot p(1,x)) \cdot ((1-2) \cdot q(0,x) + 2 \cdot q(1,x)) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} (p(0,x) \cdot q(0,x) + 4p(1,x) \cdot q(1,x) - 2q(0,x) \cdot p(1,x) - 2p(1,x) \cdot q(0,x)). \end{aligned}$$

Since we are ignoring the cost of additions and multiplications by two, this quantity can be computed in n bb multiplications, as the products $p(0,x) \cdot q(0,x)$ and $p(1,x) \cdot q(1,x)$ have already all been computed, so the only additional products required are the "cross-terms" $q(0,x) \cdot p(1,x)$ and $p(1,x) \cdot q(0,x)$ for all $x \in \{0,1\}^{\ell-1}$.

These extra products (namely, $p(y) \cdot q(\bar{y})$ for all $y \in \{0,1\}^\ell$ with $\bar{y}$ denoting y with the first bit flipped) are stored by the prover for use in future rounds. Specifically, the data structure C is updated to store not only $p(y)q(y)$ for all $y \in \{0,1\}^\ell$, but also $p(y) \cdot q(\bar{y})$.

Remark 2. *In Algorithm 1 (Section 3.1.1), the prover computed*

$$((1-2) \cdot p(0,x) + 2 \cdot p(1,x)) \cdot ((1-2) \cdot q(0,x) + 2 \cdot q(1,x))$$

with a single base-field multiplication, while here we are computing it with two base-field multiplications (one for each cross term, $p(0,x) \cdot q(1,x)$ and $p(1,x) \cdot q(0,x)$), in addition to the two base-field multiplications, that were required simply to compute the correct answer, namely $p(0,x) \cdot q(0,x)$ and $p(1,x) \cdot q(1,x)$. The reason to pay the extra price in our new prover implementation, of two base field multiplications instead of one, is that these cross terms will be useful in subsequent rounds.

Round 2. Given the products stored in the data structure C, the prover can compute $s_2(0)$ and $s_2(1)$ with just three additional be multiplications and two ee multiplications in total. This is because $s_2(0)$ and $s_2(1)$ are simple expressions of the already-computed products, which are all of the form $p(x) \cdot q(x)$ and $p(x) \cdot q(\bar{x})$ as x ranges over $\{0,1\}^\ell$. For example:

$$s_2(0) = \sum_{x \in \{0,1\}^{\ell-2}} p(r_1, 0, x) \cdot q(r_1, 0, x)$$

$$= \sum_{x \in \{0,1\}^{\ell-2}} ((1-r_1) \cdot p(0,0,x) + r_1 \cdot p(1,0,x)) \cdot ((1-r_1)q(0,0,x) + r_1 \cdot q(1,0,x)).$$

Expanding the x'th term of this sum yields:

$$(1-r_1)^2 \cdot p(0,0,x) \cdot q(0,0,x) + (1-r_1) \cdot r_1 \cdot p(0,0,x) \cdot q(1,0,x) + r_1 \cdot (1-r_1) \cdot p(1,0,x) \cdot q(0,1,z) + r_1^2 \cdot p(1,0,x) \cdot q(1,0,x).$$

Hence, every term equals a previously-computed product, times either r_1^2, $r_1(1-r_1)$, or $(1-r_1)^2$.[9]

Computing $s_2(2)$, however, involves additional products, namely all those of the form $p(x) \cdot q(x')$, where x and x' *disagree* in their second bit (and may or may not disagree on their first bit). This is an additional $2n \cdot \mathsf{bb}$ multiplications (since for every one of the n possible inputs x, there are two new inputs x' such that the prover must compute $p(x) \cdot q(x')$, namely the x' that agrees with x in the first bit but not the second, and the x' that agrees with x in the second bit but not the first).

Specifically,

$$s_2(2) = \sum_{x \in \{0,1\}^{\ell-2}} p(r_1, 2, x) \cdot q(r_1, 2, x) = \sum_{x \in \{0,1\}^{\ell-2}} ((1-r_1)p(0,2,x) + r_1 p(1,2,x)) \cdot ((1-r_1)q(0,2,x) + r_1 q(1,2,x)).$$

This expression in turn equals

$$\sum_{x \in \{0,1\}^{\ell-2}} G(x) \cdot H(x), \tag{16}$$

where

$$G(x) = ((1-r_1)((1-2)p(0,0,x) + 2p(0,1,x))) + r_1((1-2)p(1,0,x) + 2p(1,1,x)))$$

and

$$H(x) = ((1-r_1)((1-2)q(0,0,x) + 2q(0,1,x)) + r_1(1-2)q(1,0,x) + 2q(1,1,x))).$$

Applying the distributive law expresses $G(x) \cdot H(x)$ as the desired sum of sixteen different products of evaluations of p and q, namely:

$$\begin{aligned}
&(1-r_1)^2 (p(0,0,x) \cdot q(0,0,x) - 2p(0,0,x) \cdot q(0,1,x) - 2p(0,1,x) \cdot q(0,0,x) + 4p(0,1,x) \cdot q(0,1,x)) \\
&+(1-r_1)r_1 (p(0,0,x) \cdot q(1,0,x) - 2p(0,0,x) \cdot q(1,1,x) - 2p(0,1,x) \cdot q(1,0,x) + 4p(0,1,x) \cdot q(1,1,x)) \\
&+(1-r_1)r_1 (p(1,0,x) \cdot q(0,0,x) - 2p(1,0,x) \cdot q(0,1,x) - 2p(1,1,x) \cdot q(0,0,x) + 4p(1,1,x) \cdot q(0,1,x)) \\
&+r_1^2 (p(1,0,x) \cdot q(1,0,x) - 2p(1,0,x) \cdot q(1,1,x) - 2p(1,1,x) \cdot q(1,0,x) + 4p(0,1,x) \cdot q(1,1,x)).
\end{aligned}$$

[9] Of course, it is simpler and cheaper to compute $s_2(1)$ as $s_1(r_1) - s_2(0)$.

Hence, in round two, the prover appends an additional $2n$ products to the data structure C, so that C stores all products of the form $p(x)q(\bar{x})$, where x ranges over $\{0,1\}^\ell$ and $\bar{x}$ ranges over the four vectors in $\{0,1\}^\ell$ that agree with x in all but the first two coordinates.

Round i. In round $i > 2$, the prover can always compute $s_i(0)$ and $s_i(1)$ given products computed and stored in the data structure C during the previous round (namely, $p(x) \cdot q(\bar{x})$, where x ranges over $\{0,1\}^\ell$ and $\bar{x}$ ranges over the 2^i vectors in $\{0,1\}^\ell$ that agree with x on all but the first i coordinates).

Computing $s_i(2)$ requires an additional $2^{i-1} \cdot n$ bb multiplications, the results of which are stored in the data structure C. Specifically, at the start of round i, C contains all products of the form $p(x)q(\bar{x})$ where x ranges over $\{0,1\}^\ell$ and $\bar{x}$ ranges over vectors in $\{0,1\}^\ell$ that agree with x on all but the first $i-1$ coordinates. During round i, the prover appends an additional $2^{i-1} \cdot n$ base field elements to C, ensuring that C contains $p(x) \cdot q(\bar{x})$, where now $\bar{x}$ ranges over vectors that agree with x on all but the first i coordinates.

When expressing $s_i(2)$ as a linear combination of the values stored in C at the end of round i, $p(y) \cdot q(y')$ gets multiplied by

$$\widetilde{\mathsf{eq}}((y_1, \dots, y_i), (r_1, \dots, r_{i-1}, 2)) \cdot \widetilde{\mathsf{eq}}((y'_1, \dots, y'_i), (r_1, \dots, r_{i-1}, 2)). \tag{17}$$

For each $j = 1, \dots, i-1$, letting $z_j = y_j + y'_j$. this is a product of the factors

$$r_j^{z_j} \cdot (1 - r_j)^{2 - z_j},$$

(along with the additional factor $\widetilde{\mathsf{eq}}(y_i, 2) \cdot \widetilde{\mathsf{eq}}(y'_i, 2)$).

Hence, the algorithm at each round i computes an array D of 3^{i-1} values, one for each vector $z = (z_1, \dots, z_{i-1}) \in \{0,1,2\}^{i-1}$, with $D[z]$ equal to:

$$\prod_{j=1}^{i-1} r_j^{z_j} \cdot (1 - r_j)^{2 - z_j}.$$

D in round i can be updated with $1 + 3^{i-1}$ ee multiplications in total, via the recurrence[10]

$$D[z] \leftarrow r_{i-1}^{z_{i-1}} \cdot D[z_1, \dots, z_{i-2}] + (1 - r_{i-1})^{2 - z_{i-1}} \cdot D[z_1, \dots, z_{i-2}]. \tag{18}$$

This means that across the entirety of the first j rounds, the number of ee multiplications to maintain the array D is at most $j + 3^j$, and the number of be multiplications to multiply each entry of D by the appropriate sum of entries of C is 3^j.

Combining Algorithm 3 with prior algorithms. In Algorithm 3, eventually i gets large enough that $2^i \cdot n$ bb multiplications and $1 + 3^{i-1}$ ee multiplications is worse than the cost of Algorithm 1 at round $i+1$ onwards. Moreover, Algorithm 3's need to store $2^i \cdot n$ base field elements in round i can also be prohibitive in practice when i gets large.

Accordingly, eventually one should "switch over" to Algorithm 1 or Algorithm 2. Per Remark 1, Algorithm 1 should be used if be multiplications are expensive, while Algorithm 2 should be used if be multiplications are cheap. Later (Sections 5 and 6), we work out the optimal rounds at which to switch over from Algorithm 3 to Algorithm 1 and/or Algorithm 2.

4.2 Algorithm 3 When $d = 3$

Let $g(x) = p(x) \cdot q(x) \cdot h(x)$ where p, q, and h are each multilinear. The prover maintains two arrays C and C' initially of length $n = 2^\ell$, indexed by $x \in \{0,1\}^\ell$. Initially, $C[x]$ contains $p(x) \cdot q(x)$ and $C'[x]$ contains $C[x] \cdot h(x)$. This initialization of the two arrays costs $2n \cdot \mathsf{bb}$ multiplications in total.

[10]The first ee multiplication simply computes r_{i-1}^2, from which $(1 - r_i)^2$ can be derived with no additional ee multiplications.

Round 1. Given the contents of the array, the prover can compute $s_1(0)$ and $s_1(1)$ with no multiplications at all, since

$$s_1(0) = \sum_{x \in \{0,1\}^{\ell-1}} p(0,x) \cdot q(0,x) \cdot h(0,x) = \sum_{x \in \{0,1\}^{\ell} :\, x_1 = 0} C'[x],$$

and

$$s_1(1) = \sum_{x \in \{0,1\}^{\ell-1}} p(1,x) \cdot q(1,x) \cdot h(1,x) = \sum_{x \in \{0,1\}^{\ell} :\, x_1 = 1} C'[x].$$

The prover computes $s_1(2)$ as follows. Per Equation (9),

$$\begin{aligned} s_1(2) &= \sum_{x \in \{0,1\}^{\ell-1}} p(2,x) \cdot q(2,x) \cdot h(2,x) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} ((1-2) \cdot p(0,x) + 2 \cdot p(1,x)) \cdot ((1-2) \cdot q(0,x) + 2 \cdot q(1,x)) \cdot ((1-2) \cdot h(0,x) + 2 \cdot h(1,x)) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} z(x), \end{aligned} \tag{19}$$

where

$$\begin{aligned} z(x) = &-p(0,x) \cdot q(0,x) \cdot h(0,x) + 2q(0,x) \cdot p(0,x) \cdot h(1,x) + 2q(0,x) \cdot p(1,x) \cdot h(0,x) - 4q(0,x) \cdot p(1,x) \cdot h(1,x) \\ &+ 2q(1,x) \cdot p(0,x) \cdot h(0,x) - 4q(1,x) \cdot p(0,x) \cdot h(1,x) - 4q(1,x) \cdot p(1,x) \cdot h(0,x) + 8p(1,x) \cdot q(1,x) \cdot h(1,x). \end{aligned}$$

Here, $z(x)$ involves eight terms, one for each product of the form $p(y) \cdot q(y') \cdot q(y'')$ where $y, y', y'' \in \{0,1\}^{\ell}$ agree on their last $\ell - 1$ bits (and may or may not differ in their first bit). How expensive is it to compute all such products given the contents of C and C'?

Since we are ignoring the cost of additions and multiplications by powers of two, this quantity can be computed in $4n$ bb multiplications. Indeed, the products $p(0,x) \cdot q(0,x) \cdot h(0,x) = C'[0,x]$ and $p(1,x) \cdot q(1,x) \cdot h(1,x) = C'[1,x]$ have already all been computed. Meanwhile, $p(0,x) \cdot q(0,x) \cdot h(1,x)$ and $p(1,x) \cdot q(1,x) \cdot h(0,x)$ can each be computed with one additional bb multiplication each (as they equal $C[0,x] \cdot h(1,x)$ and $C[1,x] \cdot h(0,x)$ respectively). The remaining four terms equal one of the two "cross-terms" computed by Algorithm 3 in round 1 of the degree-2 case (namely $p(y) \cdot q(\bar{y})$ for some $y \in \{0,1\}^{\ell}$), times either $h(y)$ or $h(\bar{y})$. So across all $x \in \{0,1\}^{\ell-1}$, these four terms can be computed with $3n$ bb multiplications in total: n for the cross-terms $p(y) \cdot q(\bar{y})$ and $2n$ more to multiply each such cross-term by $h(y)$ and $h(\bar{y})$.

All of these extra products are stored by the prover in C and C' for use in future rounds. Specifically, as in the degree-two case, the data structure C is updated to store not only $p(y)q(y)$ for all $y \in \{0,1\}^{\ell}$, but also $p(y) \cdot q(\bar{y})$. Similarly, the data structure C' is updated to store $p(y) \cdot q(y') \cdot q(y'')$ where $y, y', y'' \in \{0,1\}^{\ell}$ may or may not differ in their first bit.

Because $s_1(X)$ has degree $d = 3$, the prover has to evaluate not only $s_1(0)$, $s_1(1)$, and $s_1(2)$, but also $s_1(3)$. Fortunately, analogous to Equation (10),

$$s_1(3) = \sum_{x \in \{0,1\}^{\ell-1}} ((1-3) \cdot p(0,x) + 3 \cdot p(1,x)) \cdot ((1-3) \cdot q(0,x) + 3 \cdot q(1,x)) \cdot ((1-3) \cdot h(0,x) + 3 \cdot h(1,x)).$$

This sum can also be written as

$$\sum_{x \in \{0,1\}^{\ell-1}} z'(x)$$

such that $z'(x)$ is a weighted sum of the same products arising in the computation of $s_1(2)$, namely $p(y) \cdot q(y') \cdot q(y'')$ where $y, y', y'' \in \{0,1\}^{\ell}$ agree on their last $\ell - 1$ bits. So $s_1(3)$ can be computed without any additional multiplications.

Round 2. Given the products stored in the data structures C and C', the prover can compute $s_2(0)$ and $s_2(1)$ with just four additional be multiplications and six ee multiplications in total. This is because $s_2(0)$ and $s_2(1)$ are simple expressions of the already-computed products, which are all of the form $p(y) \cdot q(y') \cdot h(y'')$ as y ranges over $\{0,1\}^\ell$ and y' and y'' may or may not differ from y in their first bit. For example:

$$\begin{aligned} s_2(0) &= \sum_{x \in \{0,1\}^{\ell-2}} p(r_1,0,x) \cdot q(r_1,0,x) \cdot h(r_1,0,x) \\ &= \sum_{x \in \{0,1\}^{\ell-2}} ((1-r_1) \cdot p(0,0,x) + r_1 \cdot p(1,0,x)) \cdot ((1-r_1)q(0,0,x) + r_1 \cdot q(1,0,x)) \cdot ((1-r_1)h(0,0,x) + r_1 \cdot h(1,0,x)) . \end{aligned}$$

Expanding the x'th term of this sum yields:

$$\begin{gathered} (1-r_1)^3 \cdot p(0,0,x) \cdot q(0,0,x) \cdot h(0,0,x) + (1-r_1)^2 \cdot r_1 \cdot p(0,0,x) \cdot q(0,0,x) \cdot h(1,0,x) + \\ (1-r_1)^2 \cdot r_1 \cdot p(0,0,x) \cdot q(1,0,x) \cdot h(0,0,x) + (1-r_1) \cdot r_1^2 \cdot p(0,0,x) \cdot q(1,0,x) \cdot h(1,0,x) \\ (1-r_1)^2 r_1 \cdot p(1,0,x) \cdot q(0,0,x) \cdot h(0,0,x) + (1-r_1) \cdot r_1^2 \cdot p(1,0,x) \cdot q(0,0,x) \cdot h(1,0,x) + \\ (1-r_1) \cdot r_1^2 \cdot p(1,0,x) \cdot q(1,0,x) \cdot h(0,0,x) + r_1^3 \cdot p(1,0,x) \cdot q(1,0,x) \cdot h(1,0,x). \end{gathered}$$

Hence, every term equals a previously-computed product, times either r_1^3, $r_1^2(1-r_1)$, $r_1(1-r_1)^2$, or $(1-r_1)^3$.

Computing $s_2(2)$ and $s_2(3)$. Computing $s_2(2)$, however, involves additional products, namely all those of the form $p(y) \cdot q(y') \cdot h(y'')$, where y, y', and y' may or may not disagree on their first *two* bits. This is $16n \cdot$ bb terms in total (since for every one of the $4n$ possible choices of y, y', there are four new inputs y'' such that the prover must compute $p(y) \cdot q(y') \cdot h(y'')$).

Specifically,

$$\begin{aligned} s_2(2) &= \sum_{x \in \{0,1\}^{\ell-2}} p(r_1,2,x) \cdot q(r_1,2,x) \cdot h(r_1,2,x) \\ &= \sum_{x \in \{0,1\}^{\ell-2}} ((1-r_1)p(0,2,x) + r_1 p(1,2,x)) \cdot ((1-r_1)q(0,2,x) + r_1 q(1,2,x)) \cdot ((1-r_1)h(0,2,x) + r_1 h(1,2,x)) . \end{aligned}$$

This expression in turn equals

$$\sum_{x \in \{0,1\}^{\ell-2}} F(x) \cdot G(x) \cdot H(x), \tag{20}$$

where

$$F(x) = ((1-r_1)((1-2)p(0,0,x) + 2p(0,1,x))) + r_1((1-2)p(1,0,x) + 2p(1,1,x)))$$

and

$$G(x) = ((1-r_1)((1-2)q(0,0,x) + 2q(0,1,x)) + r_1(1-2)q(1,0,x) + 2q(1,1,x))) ,$$

and

$$H(x) = ((1-r_1)((1-2)h(0,0,x) + 2h(0,1,x)) + r_1(1-2)h(1,0,x) + 2h(1,1,x))) ,$$

Applying the distributive law expresses $F(x) \cdot G(x) \cdot H(x)$ (for $x \in \{0,1\}^{\ell-2}$) as the desired sum of 64 different products of evaluations of p, q, and h (each multiplied by r_1^3, $r_1^2(1-r_1)$, $r_1(1-r_1)^2$, or $(1-r_1)^3$). Across all such x, this indeed results in $64 \cdot (n/4) = 16n$ products of the form $p(y) \cdot q(y') \cdot q(y'')$ in total.

Optimizing computation of $s_2(2)$ and $s_2(3)$. However, several of the terms (or partial products thereof) have already been computed in round one. Specifically, $p(y) \cdot q(y') \cdot h(y'')$ is already stored in C' so long as y, y', and y'' all agree in their second bit. This captures $4n$ out of the $16n$ terms. For the remaining $12n$ terms, if y and y' agree on their second bit[11] then C already stores $p(y) \cdot q(y')$ and hence just one more multiplication is required to compute $p(y) \cdot q(y') \cdot h(y'')$ (and the algorithm appends the result to C'). If y and y' do not agree on their second bit, then two multiplications are required, one to compute $p(y) \cdot q(y')$ (which the algorithm appends to C) and one to multiply the result by $h(y'')$ (the algorithm appends the result to C'). In total, this is $12n + 2n = 14n$ bb multiplications. Here, the $2n$ term captures the multiplications required in total to compute the relevant products appended to C, and the $12n$ term captures the additional multiplications required to compute the additional entries of C'.

The prover stores all products (including partial products $p(y) \cdot q(y')$) in round two. That is, the prover expands the size of the data structure C to $4n$, so that C stores all products of the form $p(y)q(y')$, where y ranges over $\{0,1\}^\ell$ and y' ranges over the four vectors in $\{0,1\}^\ell$ that agree with y in all but the first two coordinates. The prover similarly ensures that C' has size $16n$, containing all products of the form $p(y) \cdot q(y') \cdot q(y'')$ where y, y', and y'' agree in all but the first two coordinates.

As with $s_1(3)$, $s_2(3)$ can be computed without any additional multiplications.

Round i. In round $i > 2$, the prover can always compute $s_i(0)$ and $s_i(1)$ given products computed and stored in the data structure C' during the previous round. For degree $d = 3$, the total cost is at most $2 \cdot (d-1) + (d+1) + (d+1)^{i-1}$ ee multiplications to compute all $(d+1)^{i-1}$ relevant products of powers of $r_1, (1-r_1), r_2, (1-r_2), \ldots, r_{i-1}, (1-r_{i-1})$, and $(d+1)^{i-1}$ be multiplications to multiply the results by the appropriate sums of entries of C'. Here, $2 \cdot (d-1)$ counts the number of multiplications needed to compute the first d powers of r_i and $(1-r_i)$, $d+1$ counts the number of multiplications needed to derive $r_i^j \cdot (1-r_i)^{i-j}$ for $j = 0, \ldots, d$, and $(d+1)^{i-1}$ counts the number of multiplications needed to derive every possible product of these values across variables $1, \ldots, i-1$ (given that all possible products for the first $i-2$ variables were computed and stored via previous rounds).

Computing $s_i(2)$ and $s_i(3)$ requires an additional $\left(2^{i-1} + (4^i - 4^{i-1})\right) \cdot n$ bb multiplications to update the entries of C and C'.

4.3 Algorithm 3 for general d

Algorithm description. Suppose $g(x) = p_1(x) \cdot p_2(x) \cdot \cdots \cdot p_d(x)$. For general d, the algorithm is closely analogous to the degree-3 case. Rather than maintaining two arrays C and C' as in the case $d = 3$, the prover will maintain $d-1$ arrays $C_2, \ldots, C_d$. At the end of each round j, C_i will store all relevant products of the form $p_1(y^{(1)}) \cdot p_2(y^{(2)}) \cdot \cdots \cdot p_i(y^{(i)})$, where $y^{(1)} \ldots, y^{(i)} \in \{0,1\}^\ell$ agree in their last $\ell - j$ entries. The same reasoning as for the degree $d = 3$ case explains that $s_i(0), \ldots, s_i(d)$ are each a linear combination of these values, with the coefficients in the linear combinations given by products of appropriate powers of $r_1, (1-r_1), \ldots, r_j, (1-r_j)$. The number of ee and be multiplications needed to compute these coefficients across the entirety of the first j rounds is at most $(d-1)j + (d+1)^j$.

Completing the cost analysis. In each round, the arrays are updated one at a time, starting with C_2 and proceeding to C_d. The cost of updating the i'th array in round j is $2^{ji} - 2^{j(i-1)}$ bb multiplications. Indeed, in round j, array C_i grows from size $2^{(i-1)j}$ to 2^{ij}, and each new element can be computed by multiplying an already-computed element of C_{i-1} by $p_i(y^{(i)})$ for some $y^{(i)} \in \{0,1\}^\ell$.

Thus, the cost for applying this algorithm for the first j rounds is $(d-1)j + (d+1)^j$ ee multiplications, $(d+1)^j$ be multiplications, plus the following number of bb multiplications:

$$\left((d-1) + 2^j + 4^j + 8^j + \cdots + 2^{(d-1)j}\right) n. \tag{21}$$

[11] $4n$ out of the remaining $12n$ terms agree on their second bit.

5 Cost optimization when bb and be multiplications are "free"

When the goal is to minimize the number of ee multiplications, with be and bb multiplications considered free, it is optimal to use Algorithm 1 for the early rounds of sum-check, then switch to Algorithm 2, then to Algorithm 3.

How to implement the switch from Algorithm 2 to Algorithm 1. The prover can switch from Algorithm 2 to Algorithm 1 at the end of round i by computing the contents of all d arrays from Algorithm 1 at the end of round i of the protocol (i.e., immediately after r_i has been bound). This requires $(n - n/2^i) \cdot \mathsf{be}$ multiplications per array (of course, the precise number is not important if we are ignoring the cost of be multiplications).

For example, consider the case that $d = 2$ so there are two arrays A and B. Then after round 1, for each $x \in \{0,1\}^{\ell-1}$,

$$A[x] = r_1 p(1,x) + (1-r_1)p(0,x) = p(0,x) + r_1(p(1,x) - p(0,x)),$$

which can be computed with $n/2$ be multiplications. And after round 2, for each $x \in \{0,1\}^{\ell-2}$,

$$A[x] = r_1 r_2 p(1,1,x) + r_1(1-r_2)p(1,0,x) + (1-r_1)r_2 p(0,1,x) + (1-r_1)(1-r_2)p(0,0,x)$$
$$= r_1 r_2 \left(p(1,1,x) - p(1,0,x) - p(0,1,x) + p(0,0,x)\right) + r_1 \left(p(1,0,x) - p(0,0,x)\right) + r_2 \left(p(0,1,x) - p(0,0,x)\right) + p(0,0,x).$$

For general rounds $i > 2$, for each $x \in \{0,1\}^{\ell-i}$, at the end of round i of Algorithm 1, $A[x]$ can be expressed as a sum of 2^i terms, each involving a multiplication by an extension field element (a product of a subset of $\{r_1, \ldots, r_i\}$, where the empty product equals 1) and a base field element (obtained as a sum of at most 2^i evaluations of p). This means all entries of A can be computed at the end of round i with $(1 - 1/2^i) \cdot n$ be multiplications in total.

The optimal round to switch from Algorithm 2 to Algorithm 1. If the switchover happens at the end of round j, then for rounds $R = j+1, \ldots, \ell$, the number of ee multiplications that the Algorithm 1 prover performs across rounds $j+1, \ldots, \ell$ is:

$$d(d-1)n \sum_{R=j+1}^{\ell} 1/2^R \leq \left(d^2/2^j\right) \cdot n.$$

Accordingly, the optimal round i for switching from Algorithm 2 to Algorithm 1 is roughly the i satisfying $d^2 n/2^i = d(d-1)2^i$, which means $i \approx \ell/2$.

The optimal round to switch from Algorithm 3 to Algorithm 2. If the switch from Algorithm 3 to Algorithm 2 occurs at the end of round j, then the total number of ee multiplications performed is:

$$(d-1)j + (d+1)^j + \left(\sum_{i=j+1}^{\ell/2} (d^2 - d)n/2^i + 2^{i-1} \right) + \left(\sum_{i=\ell/2+1}^{\ell} d^2 n/2^i \right).$$

Hence, the optimal switchover round j, for switching from Algorithm 3 to Algorithm 2, is roughly the j satisfying $(d+1)^j = (d^2-d)n/2^j$, which means

$$j \approx \log(n)/(1 + \log(d+1)).$$

In this case, for constant d the total number of ee multiplications is $O(n^{1-1/(1+\log(d+1))})$. For example, if $d = 3$, this is $O(n^{2/3})$ and if $d = 16$, then this is about $O(n^{0.803})$. In particular, for any constant degree d, we reduce the number of ee multiplications to be *sublinear* in the number n of terms being summed.

Savings over prior work. The best prior algorithm (the combination of Algorithms 1 and 2 discussed in Remark 1) required roughly the following number of ee multiplications:

$$d(d-1)\cdot n/2.$$

For example, if $d=3$ we have reduced the prover's cost by a factor of about $\Theta(n^{1/3})$.

Concretely, the savings can be several orders of magnitude. For example, for $d=3$ and for reasonable values of n (say, $2^{20}\leq n\leq 2^{30}$), we improve the prover time relative to prior algorithms by a factor of several hundred. Specifically, when $n=2^{30}$, our new algorithm does 7.47 million extension field multiplications, while Algorithm 1 alone would do $3\cdot 2^{30}$ of them. This is a savings of over $430\times$. For $n=2^{24}$, our algorithm does 434,000 ee multiplications, vs. $3\cdot 2^{24}$ for Algorithm 1 alone. The savings is therefore still larger than a factor of 115.

In practice, the high space complexity of Algorithm 3 may necessitate switching to Algorithm 2 earlier than round $\log(n)/(1+\log_2(d+1))$. Fortunately, most of the savings over prior work comes from the first few rounds, as the number of ee multiplications falls geometrically as the switchover round increases.

6 Cost optimization when bb and be multiplication aren't free

When be multiplications are not free, and Karatsuba's algorithm is used for ee multiplications, it typically does not make sense to use Algorithm 2, as the savings in ee multiplications relative to Algorithm 1 does not compensate for the increased number of be multiplications. So in this case, the optimal combination is to switch straight from Algorithm 3 to Algorithm 1. In the following sections, we determine at what point it is best to switch (in the case $d=2$ we also slightly optimize the cost of implementing the switch). We then calculate how much the resulting algorithm improves over Algorithm 1 alone.

6.1 Degree $d=2$

An extra optimization when switching to Algorithm 1. If the switchover from Algorithm 3 to Algorithm 1 happens at the end of round j, then per Remark 2, the cost of round j can be reduced by a factor of two, from $2^{j-1}n$ bb multiplications, to 2^{j-2} bb multiplications. This is because the reason for computing extra cross terms $p(x)q(\bar{x})$ in round j that were not already computed by round $j-1$ is to make use of those cross terms in future rounds, yet if the switch-over happens at the end of round j then there is no point to computing these cross terms. For example, in the case $j=2$, Equation (16) reveals that $s_2(2)$ can be expressed as

$$\sum_{x\in\{0,1\}^{\ell-2}} G(x)\cdot H(x)$$

where $G(x)$ is of the form $w(x)+r_1z(x)$ and $H(x)$ is of the form $w'(x)+r_1z'(x)$ for some base field elements $w(x), z(x), w'(x), z'(x)$. Hence,

$$s_2(2)=\left(\sum_{x\in\{0,1\}^{\ell-2}} w(x)\cdot w'(x)\right)+r_1\cdot\left(\sum_{x\in\{0,1\}^{\ell-2}} w(x)\cdot z'(x)+z(x)w'(x)\right)+r_1^2\left(\sum_{x\in\{0,1\}^{\ell-2}} z(x)z'(x)\right).$$

This therefore entails the prover computing 4 base field multiplications for each of the $n/4$ terms of the sum in Equation (16), a factor-2 improvement over the $2n$ base field multiplications required to compute the extra cross terms that would otherwise be added to the data structure C in round j.

Total costs of combining Algorithms 1 and 3. In summary, if the switchover to Algorithm 3 occurs at the end of round j, the total prover cost for the remaining rounds is $4n/2^j$ ee multiplications, plus $\left(n+\left(\sum_{i=1}^{j}2^{i-1}\cdot n\right)-2^{j-2}n\right)\cdot\mathsf{bb}=(3/4)\cdot 2^j\cdot n\cdot\mathsf{bb}$ from the first j rounds, plus $(2\cdot n-n/2^j)$ be

Extension degree	4	8	16	32	64	128	256
Optimal switchover round j (switch at end of round j)	3	4	4	5	6	7	8
Prover cost in terms of ee mults per term of sum	2.0	1.27	0.78	0.483	0.302	0.192	0.122
Prover cost in terms of bb mults per term of sum	18	34	63	117	220	420	800
Prover cost in ee mults for Algorithm 1	2.61	2.35	2.21	2.14	2.09	2.06	2.04
Factor improvement over Algorithm 1 alone	1.31	1.85	2.83	4.43	6.92	10.8	16.7

Table 1: Cost of our prover implementation combining Algorithms 1 and 3, for $d = 2$, in terms of number of ee multiplications. We assume that Karatsuba's algorithm is used for extension field multiplication, so that be and ee multiplications are respectively k and $k^{1.585}$ times costlier than bb multiplications.

multiplications to compute the A and B arrays used in Algorithm 1 at the end of round j.[12] Thus, the total prover cost will be:

$$(3/4) \cdot 2^j \cdot n \cdot \mathsf{bb} + (2 \cdot n - n/2^j + 3^j) \cdot \mathsf{be} + \left(4/2^j + (j + 3^j)\right) \cdot n \cdot \mathsf{ee}. \tag{22}$$

Optimal choice of switchover. The optimal choice of switchover round j occurs roughly when setting

$$(3/4) \cdot 2^{j+1} \cdot n \cdot \mathsf{bb} = \left(4n/2^j\right) \mathsf{ee} \Longleftrightarrow \mathsf{ee} = (3/8)2^{2j} \cdot \mathsf{bb} \Longleftrightarrow j = \frac{\log((8/3) \cdot \mathsf{ee}/\mathsf{bb})}{2}$$

If using a degree k extension and Karatsuba's algorithm for extension field multiplication, then $\mathsf{ee} \approx k^{1.5849} \cdot \mathsf{bb}$, and the above simplifies to

$$j = \frac{1.5849 \cdot \log(k) + \log(8/3)}{2}.$$

See Table 1 for concretely optimal switchover rounds.

As the extension degree k approaches infinity, the optimal switchover occurs roughly at round $.8 \cdot \log k$. This more or less replaces the $2 \cdot n$ ee multiplications of Algorithm 1 with roughly $2n/2^{.8 \log(k)}$ ee multiplications, a savings of roughly a factor of $k^{0.8}$. However, most the savings come from the first few rounds.

6.2 The case of general d

For general degrees d, per Equation (21), if one switches from Algorithm 3 to Algorithm 1 at the end of round j, the total prover cost will be:

$$\left(\left((d-1) + 2^j + 4^j + 8^j + \cdots + 2^{(d-1)j}\right) n\right) \mathsf{bb} + \left(d(1 - 1/2^j)n\right) \cdot \mathsf{be} + \left((d^2/2^j)n\right) \cdot \mathsf{ee},$$

plus (at most) an additional $(d-1)j + (d+1)^j$ be and ee multiplications. These last quantities are independent of n (so long as the switchover round j is independent of n) and will not be a significant contributor to costs for values of j relevant to this section.

This can be compared to the cost of Algorithm 1 alone, which recall (Equation (12)) is roughly:

$$((d^2 - 1)n/2) \cdot \mathsf{bb} + (dn/2) \cdot \mathsf{be} + (d^2/2) \cdot n \cdot \mathsf{ee}.$$

For degree $d = 3$, we numerically calculate the optimal switchover round and improvement factor in Table 2. The improvement is a relatively modest factor of 1.89 for extension degree $k = 16$, but grows to a substantial factor of 5.4 for $k = 128$.

[12] Algorithm 3 also incurs at most an additional $j + 3^j$ ee and be multiplications in total over the first j rounds, per Equation (18). We include these terms in Expression (22) for completeness, but they will not be a major contributor to costs for values of j relevant to this section of the manuscript. This is because per Table 1, the optimal switchover round j is at most 9 and we are generally interested in sums with at least $2^\ell \geq 2^{20}$ terms.

Extension degree	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$
Optimal switchover round j (switch at the end of round j)	2	3	3	4	4
Prover cost in terms of ee mults per term of sum	3.73	2.56	1.77	1.19	0.85
Algorithm 1 prover costs (ee mults per term)	5.1	4.85	4.71	4.64	4.59
Factor improvement over Algorithm 1 alone	1.37	1.89	2.66	3.9	5.4

Table 2: Cost of Algorithm 3 (combined with Algorithm 1) for $d = 3$ in terms of number of ee multiplications, assuming extension field multiplications are $k^{1.585}$ times more expensive than bb multiplications and k times more expensive than be multiplications.

7 Applications and Extensions

A major issue not yet discussed is whether the most important applications of the sum-check protocol satisfy the assumption under which we analyzed the costs of Algorithms 1-3 above, namely that the sum-check protocol is applied to a polynomial $g(x)$ of the form

$$\prod_{i=1}^{d} p_i(x),$$

where $p_i(x)$ is a base-field element for all $x \in \{0,1\}^\ell$.

This is the case for at least some applications of sum-check. One example is the *super-efficient*[13] protocol for counting triangles of [Tha13] (see [Tha22, Section 4.5.1] for an exposition), if that protocol is instantiated over an extension field.[14] Specifically, if A is the adjacency matrix of a graph, A^2 is the squared adjacency matrix, and $\widetilde{A}$ and $\widetilde{A^2}$ their multilinear extension polynomials respectively, then the counting triangles protocol applies the sum-check protocol to the polynomial

$$g(x) = \widetilde{A}(x) \cdot \widetilde{A^2}(x).$$

However, in the most important applications of the sum-check protocol to SNARK design, the sum-check protocol is applied to a polynomial $g(x)$ that does not quite satisfy the assumption. In this section we explain these deviations from the assumption, and how to address them.

7.1 First generalization: g has one factor that is not in the base field

Suppose that

$$g(x) = \prod_{i=1}^{d} p_i(x)$$

where for $i = 2, \dots, d$, $p_i(x) \in \mathbb{B}$ for all $x \in \{0,1\}^\ell$ but this is not the case for $p_1(x)$. For example, suppose that $g_1(x) = \widetilde{\mathsf{eq}}(r, x)$ for some value $r \in \mathbb{F}^\ell$ chosen by the verifier from the full field $\mathbb{F}$. This situation arises in important applications of the GKR protocol such as in the Lasso lookup argument [Tha13, STW23], as well as in sum-check-based SNARKs like Spartan [Set20], BabySpartan [ST23], Hyperplonk [CBBZ23], and Binius [DP23b].

Cost analysis of Algorithm 1. Algorithm 1 (applied to a polynomial of degree d) directly handles this case, but its cost changes from Expression (12)

[13] Super-efficient means the honest prover runs the fastest known algorithm for the problem and then performs a low-order amount of additional work to prove the answer is correct.

[14] Since the number of triangles in a graph with N nodes can be as large as N^3, one would need to work over a field of characteristic at least N^3 to avoid "wrap around" issues. This is not necessary in settings where the number of triangles is a priori bounded by a number much less than N^3. One could also apply the protocol over several field of characteristic $O(\log N)$ and apply the Chinese remainder theorem to compute the actual number of triangles.

The accounting for this expression is as follows. Computing the round 1 message $s_1(0)$, $s_1(1)$, ... $s_1(d)$ now costs $d-2$ bb multiplications and one be multiplication for each of the $d+1$ evaluation points and each of the $n/2$ terms of the sum defining s_1 (see Equation (3)). The cost to update all arrays at the end of round one is now $(d-1)\cdot(n/2)$ be multiplications and $n/2$ ee multiplications. The cost of Algorithm 1 in all remaining rounds is unchanged. Hence, roughly speaking, when one factor of g is not in the base field the cost of Algorithm 1 goes up by $dn/2$ be multiplications and $n/2$ ee multiplications.

However, recall that the Algorithm 1 cost analysis assumes that the prover has already computed an array storing all the quantities $p_1(x)$. When

$$p_1(x) = \widetilde{\mathsf{eq}}(r,x) \colon x \in \{0,1\}^\ell,$$

computing these quantities can be done with n ee multiplications via a standard recurrence. So in total, the fact that in applications of the GKR protocol and elsewhere,

$$p_1(x) = \widetilde{\mathsf{eq}}(r,x) \colon x \in \{0,1\}^\ell,$$

causes the cost of Algorithm 1 to go up by roughly $dn/2$ be multiplications and $3n/2$ ee multiplications.

Cost analysis of Algorithm 3. Algorithm 3 also easily generalizes to the case that p_1 does not output base field elements even when evaluated at inputs in $\{0,1\}^\ell$, by combining with the technique of Algorithm 1. Specifically, using standard dynamic programming techniques (see [Tha22, Lemma 3.8]), one first spends n ee multiplications to compute an array A storing, for each $x \in \{0,1\}^\ell$,

$$A[x] = p_1(x) = \widetilde{\mathsf{eq}}(r,x).$$

As per Algorithm 1, at the end of each round j, with $n/2^j$ ee multiplications one can update A to ensure that for each $x \in \{0,1\}^{\ell-j}$, $A[x] = p_1(r_1,\dots,r_j,x)$.

Algorithm 3 can be applied in each round j to

$$g'(x) \coloneqq \prod_{i=2}^{d} p_i(x)$$

in order to compute computing $g'(r_1,\dots,r_{j-1},z,x) \colon x \in \{0,1\}^{\ell-j}$, where z ranges over $\{0,1,\dots,d\}$. To obtain $g(x) = g'(x)\cdot p_1(x)$, each such quantity $g(x)$ is multiplied by $p_1(r_1,\dots,r_{j-1},z,x)$, which is a linear combination of elements of the array A (with coefficients given by products as differences of elements in the interpolating set $\{0,1,\dots,d\}$, per univariate Langrange interpolation over this interpolating set).

Thus, the costs in round j are the same as the cost of Algorithm 3 applied to the degree-$(d-1)$ polynomial g', plus an extra $2n$ ee multiplications to compute and update A over the course of the protocol, and at most n extra be multiplications over the course of the protocol.

Savings over prior work bb and be multiplications are free. If the goal is to minimize the number of ee multiplications, then the cost of Algorithm 1 alone is roughly the following number of ee multiplications:

$$(d^2+3)\cdot n/2.$$

Meanwhile, Algorithm 3 combined with Algorithm 1 costs $(2+o(1))n$ ee multiplications. Accordingly, the savings is a factor of roughly

$$(d^2+3)/4.$$

For example, if $d=3$, the savings is a factor of $15/4 = 3.75$, while if $d=16$ the savings is a much more substantial factor of 64.75.

7.2 Second Generalization: Translations of Base-Field Elements

In Lasso [STW23], an optimized variant of the GKR protocol [GKR15], due to Thaler [Tha13], is applied to a circuit (specifically, a tree of multiplication gates) whose inputs $w = (w_1, \dots, w_m)$ are not themselves base-field elements, but rather each w_i is of the form $y_i - r'$, where each y_i is a base field element and r' is an extension-field element chosen at random by the verifier. We provide two different techniques for addressing this setting.

Warmup: Modifying the GKR protocol to mitigate the cost of the most expensive layer. When the circuit has fan-in two (i.e., a binary tree of multiplication gates), the prover spends about half of its work processing the layer of gates adjacent to the input layer of the circuit. Let $\widetilde{W}$ denote the multilinear extension of the gate values at the circuit layer below the inputs. When the GKR protocol reaches this layer, the verifier needs to ascertain $\widetilde{W}(r)$ for a random values $r \in \mathbb{F}^{\log(m)-1}$.

To do so, the sum-check protocol is applied to the $(\log(m) - 1)$-variate polynomial

$$g(x) = \widetilde{\mathsf{eq}}(r, x) \cdot \tilde{w}(x, 0) \cdot \tilde{w}(x, 1). \tag{23}$$

This indeed computes $\widetilde{W}(r)$, owing to the fact that

$$\widetilde{W}(r) = \sum_{x \in \{0,1\}^{\log(m)-1}} \widetilde{\mathsf{eq}}(r, x) \cdot \tilde{w}(x, 0) \cdot \tilde{w}(x, 1). \tag{24}$$

To see that Equation (24) holds, observe that the right hand side is a multilinear polynomial in r and agrees with W at all inputs in $\{0, 1\}^{\log(m)-1}$. Hence, it must equal $\widetilde{W}(r)$.

In the application of the GKR protocol in Lasso, $w_i = y_i - r'$ for all $i = 1, \dots, m$. In this case, the following variant of Equation (24) holds:

$$\widetilde{W}(r) = \left(\sum_{x \in \{0,1\}^{\log(m)-1}} \widetilde{\mathsf{eq}}(r, x) \cdot \widetilde{y}(x, 0) \cdot \widetilde{y}(x, 1) \right) - r' \cdot (\widetilde{y}(r, 0) + \widetilde{y}(r, 1)) + (r')^2. \tag{25}$$

Again, this equation holds because the right hand side is a multilinear polynomial in r and agrees with W at all inputs in $\{0, 1\}^{\log(m)-1}$.

Accordingly, to compute rather than applying the sum-check protocol to the polynomial $g(x)$ defined in Equation (23), we can apply it to

$$g(x) = \widetilde{\mathsf{eq}}(r, x) \cdot \widetilde{y}(x, 0) \cdot \widetilde{y}(y, 1), \tag{26}$$

and set $\widetilde{W}(r)$ to the result, plus

$$-r' \cdot (\widetilde{y}(r, 0) + \widetilde{y}(r, 1)) + (r')^2. \tag{27}$$

The polynomial $g(x)$ in Equation (26) satisfies the assumptions of Section 7.1, and hence the techniques in that section substantially speed up the prover. At the end of the sum-check protocol, the verifier needs to evaluate $\widetilde{y}(r'', 0)$ and $\widetilde{y}(r'', 1)$ for some $r'' \in \mathbb{F}^{\log(m)-1}$. Hence, this modification increases the number of evaluation queries to $\widetilde{y}$ from two to four.

In Lasso, $\widetilde{y}$ is committed with any multilinear polynomial commitment scheme. With many polynomial commitment schemes, the cost of these multiple evaluations can be amortized, i.e., all four evaluations can be provided at roughly the cost (to both prover and verifier) of a single evaluation.

Have the prover apply the distributive law. The above modification of the GKR protocol can be viewed as requiring the verifier to apply the sum-check protocol to a *different polynomial*, compared to prior work that did not leverage the polynomial's structure as a (product of translations of) multilinear polynomials outputting small values. This changes the verification procedure of the protocol. It also led to the verifier making extra evaluating queries to the base-field polynomial $\widetilde{y}$.

We can avoid modification to the verification procedure with a different approach. Suppose the sum-check protocol is applied to an ℓ-variate polynomial $g(x) = \prod_{i=1}^{d} p_i(x)$ where each $p_i(x) = q_i(x) - r'$ where q_i is a multilinear polynomial that maps $\{0,1\}^\ell$ to $\mathbb{B}$. Roughly speaking, the prover can apply the distributive law to express

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

as a *sum of "sub-sums"*, where each sub-sum applies to a product of a subset of the q_j's (multiplied by a power of r').

One can then apply the algorithms of this manuscript to each of these sub-sums individually. The prover's i'th-round message in the sum-check protocol applied to g is simply the appropriate sum of the i'th-round message in the sum-check protocol applied to each sub-sum.

In more detail,

$$g(x) = \sum_{S \subseteq \{1,2,\dots,d\}} (-r)^{|S|} \cdot \prod_{j \in S} q_j(x).$$

Let $g_S := \prod_{j \in S} q_j(x)$ and let $s_{i,S}(x)$ be the i'th round message in the sum-check protocol applied to g_S. Then the i'th round message s_i in the sum-check protocol applied to g is

$$\sum_{S \subseteq \{1,2,\dots,d\}} (-r)^{|S|} s_{i,S}(x).$$

Accordingly, the prover can compute $s_{i,S}(x)$ for each $S \subseteq \{1,\dots,d\}$ using the algorithms of this manuscript, and then obtain s_i via $O(d)$ additional ee multiplications.

Cost analysis. Naively, one can bound the increase in prover costs relative to the case where each p_i is untranslated (i.e., $r' = 0$) by at most a factor of 2^d, but in fact a much stronger bound holds.

Recall from Section 5 that in settings where bb and be multiplications are "free", the total prover work to compute $s_{i,S}$ is $O(n^{1-1/(1+\log_2(d+1))})$. For constant d, this means that the cost of computing $s_{i,S}$ for $S = \{1,\dots,d\}$ asymptotically dominates the *total* cost of computing $s_{i,S'}$ for all other $S' \subset \{1,\dots,d\}$.

Even in the case where bb multiplications are not free, the number of bb multiplications required to compute $s_{i,S}$ grows exponentially with $|S|$, and so we expect that the cost of computing $s_{i,S}$ for $S = \{1,\dots,d\}$ dominates the total cost of computing $s_{i,S'}$ for all other $S' \subset \{1,\dots,d\}$.

In addition, for many pairs of sets S', S'', all of the multiplications required to process $g_{S'}$ via Algorithm 3 are already computed by Algorithm 3 when applied to compute $g_{S''}$. For example, for $S'' = \{1,\dots,d\}$, Algorithm 3 performs all of the multiplications necessary to compute $s_{i,S'}$ for any S' equal to a "prefix" of $\{1,\dots,d-1\}$, meaning any set of the form $\{1,2,3,\dots,j\}$ for $j < d$. Hence, applying Algorithm 3 to $g_{S'}$ adds no multiplications to the cost of Algorithm 3, beyond what was already necessary to process $g_{S''}$.

Acknowledgments. I am grateful to Ben Diamond and Jim Posen for patiently explaining the computational implications of tower field constructions, for identifying the benefits of Algorithm 2 when working over fields of small characteristic, and for many conversations surrounding this work and their recent manuscript [DP23b].

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/`.)

References

[AST23] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. *Cryptology ePrint Archive*, 2023.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2018.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2023.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS)*, 2012.

[CTY11] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. *Proc. VLDB Endow.*, 5(1):25–36, 2011.

[DP23a] Benjamin E. Diamond and Jim Posen. Personal communication, 2023.

[DP23b] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023. `https://eprint.iacr.org/2023/1784`.

[EMGI11] Nadia El Mrabet, Aurore Guillevic, and Sorina Ionica. Efficient multiplication in finite field extensions of degree 5. In *Progress in Cryptology–AFRICACRYPT 2011: 4th International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings 4*, pages 188–205. Springer, 2011.

[FP97] John L Fan and Christof Paar. On efficient inversion in tower fields of characteristic two. In *Proceedings of IEEE International Symposium on Information Theory*, page 20. IEEE, 1997.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)*, 62(4):1–64, 2015.

[GKR$^+$21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535, 2021.

[LFKN90] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, October 1990.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[ST23] Srinath Setty and Justin Thaler. BabySpartan: Lasso-based SNARK for non-uniform computation, 2023.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with Lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023. `https://eprint.iacr.org/2023/1216`.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends in Privacy and Security*, 4(2–4):117–660, 2022.

[Wie88] Doug Wiedemann. An iterated quadratic extension of gf (2). *Fibonacci Quart*, 26(4):290–295, 1988.