The Sum-Check Protocol over Fields of Small Characteristic

Suyash Bagad* suyash@ingonyama.com
Yuval Domb† yuval@ingonyama.com
Justin Thaler‡ justin.thaler@georgetown.edu

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 are likely to 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 a 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 paper, let $n = 2^\ell$ denote the number of terms in this sum. Suppose 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.

*Ingonyama
†Ingonyama
‡a16z 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 succinct non-interactive argument of knowledge (SNARK) is a cryptographic protocol allowing an untrusted prover to prove that it knows a witness satisfying a specified property. 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]. And indeed, Jolt's performance when using large fields and elliptic-curve-based commitments is competitive with that of comparable SNARKs that work over small fields and use hashing-based commitments.

Fortunately, new works by Diamond and Posen [DP23b, DP24] called Binius and FRI-Binius give 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 now the prover bottleneck by a significant margin.[4] 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 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

[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/` and `https://a16zcrypto.com/posts/article/building-jolt/`.

[4] Preliminary experimental results on the prover bottleneck in Binius are available at `https://youtu.be/rgRWcWOllOw?feature=shared&t=1548`.

multiplications that they are essentially *free*. One example is $\mathsf{GF}[2]$, as multiplying any field element x 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, 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 (i.e., there is actually an attack demonstrating that this security bound is tight [AFK22].[5]). 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 its introduction of potentially catastrophic security issues and because, when used securely, its performance is worse than our algorithms. Indeed, our work improves over the $O(nk \log n)$ base field operations obtained from parallel repetition, as well as over the existing linear-time sum-check prover algorithms.

Our results. We start by describing 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. We also describe a *fourth* algorithm, which further improves the number of base field multiplications, without any increase in extension-field multiplications.

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, DP24]. This will render the sum-check protocol the prover bottleneck in these SNARKs, and our results help mitigate this bottleneck.

Overview of costs. Let $n = 2^\ell$ be the number of terms being summed in Equation (1), and suppose that the polynomial g is a product of d multilinear polynomials each defined over the base field (in practice, d is typically between two and four). Suppose that one executes one of our new prover algorithms (Algorithms 3 or 4) for the first i rounds of the sum-check protocol, and that "switches over" to an existing algorithm like Algorithm 1 for the final $\ell - i$ rounds. Then the prover incurs three costs:

[5] See `https://a16zcrypto.com/posts/article/17-misconceptions-about-snarks/#section--13` for an exposition of this attack. This attack is surprising and not widely known: projects have released code that is vulnerable to it, such as `https://github.com/PolyhedraZK/Expander/issues/15`. We hope that our manuscript helps raise awareness of the attack.

- In the first i rounds, Algorithm 3 performs $O(d \cdot 2^{d \cdot i} n)$ base field multiplications.
- In order to "switch over" to a pre-existing algorithm at the end of round i, the prover incurs $O(n)$ many base-field-times-extension-field multiplications. We refer to this procedure as "binding the first i variables".
- Once the switchover is complete, the prover performs $O(n/2^i)$ extension field multiplications across the final $\ell - i$ rounds.

Adapting ideas from the Toom-Cook multiplication algorithm, Algorithm 4 reduces the number of base field multiplications from $O(d \cdot 2^{d \cdot i})$ to $O(d^{i+1})$.

Comparison to independent work of Gruen [Gru24]. Independent work of Gruen [Gru24] also seeks to optimize the sum-check protocol prover when working over fields of small characteristic. Most relevant to our work is a technique Gruen calls the "univariate skip". This refers to applying sum-check to a multivariate polynomial g that has high degree in its first variable–roughly $d \cdot 2^i$ for some $i \geq 1$ and some constant d, and degree d in each of its remaining variables.

The sum-check protocol can then be used to compute the sum of g's evaluations over a set of size n (say, the set $\{0, \ldots, 2^i\} \times \{0,1\}^{\ell - i}$). In this setting, simple variations of existing linear-time sum-check prover implementations naturally perform $O((d \cdot 2^{d \cdot i}) \cdot n)$ base field multiplications in the first round, and $O(n/2^i)$ extension field multiplications across all other rounds. This has very similar prover costs to our Algorithm 3 (combined with Algorithm 1). Over FFT-friendly fields, the number of base field multiplications of the univariate-skip approach can even be asymptotically lowered, though this is unlikely to yield major speedups in practice because these FFTs would only be applied to vectors of length 2^i (and in practice we expect i to be between, say, 2 and 8).

However, the univariate-skip technique is not directly comparable to our results for several reasons. First, our approach leads to lower proof size than the univariate-skip approach, stemming from the high degree that the univariate-skip technique introduces in the first variable of g, and the fact that the sum-check protocol's proof size grows with the degree in each variable. Second, in SNARK applications the polynomial g is often derived from various committed polynomials, and the commitment scheme used would need to be modified if the polynomials being committed have large degree in their first variable and constant degree in their remaining variables. These modifications may introduce additional costs. Our techniques apply to existing sum-check-based SNARKs "as is".

One final remark is that Gruen's work is targeted at settings where the polynomial g being summed is not necessarily a product of multilinear polynomials, but rather is obtained by composing some low-degree univariate polynomial q with several mutlilinear polynomials. Of course, our techniques can also be applied to this general setting, by expressing q as a sum of monomials. In general this transformation may introduce overheads. However, in perhaps our main motivating application, namely the zkVM Jolt [AST23], the sum-check protocol is only ever applied to a product of multilinear polynomials.

Implementation and Experiments. We implemented and benchmarked all four sum-check prover algorithms[6] to compare their run times. Using the degree-4 extension of the Babybear [BG23] field, we tested single-core and six-core configurations on an Intel i7 processor. Algorithm 4 performs better than Algorithm 3 due to fewer base-field multiplications.

In comparison with the existing algorithms, Algorithms 3 and 4 require fewer extension-field multiplications but many more base-field multiplications. Naturally, if extension-field multiplications are much slower than the base-field multiplications, Algorithms 3 and 4 are expected to perform much better than the existing algorithms. In our implementation, for degree-four extensions of the Babybear field, we observe that the running times of the existing algorithms are better than the Algorithms 3 and 4. However, we expect substantial speedups in key applications like Jolt [AST23] with binary tower fields, in which extension-field multiplications are orders of magnitude slower than the base-field multiplications. We present careful analytical estimates of the relative speedup of Algorithms 3 and 4 over Algorithm 1 (Section 7) when using a

[6]Our code is open-sourced at `https://github.com/ingonyama-zk/smallfield-super-sumcheck`.

degree-128 extension of the binary field $\mathsf{GF}[2]$ that support this claim. One reason our current experimental results are limited is that we use Arkworks [ac22] as the backend for finite field arithmetic. This library is optimized for fields with characteristic higher than 64, which leads to inefficiencies when working with fields with smaller characteristic (like the 31-bit Babybear field). The library also does not support field extensions of degree more than four. Therefore, our implementation has significant potential for improvement, especially for Algorithms 3 and 4, ranging from efficient field arithmetic for smaller fields to hand-rolled multi-threading optimizations.

A companion work [DT24]: addressing an "equality" factor. Let $\mathsf{eq}(y, x)$ denote the $2n$-variate multilinear polynomial defined as follows.

$$\mathsf{eq}(y, x) = \prod_{i=0}^{n-1} (x_i y_i + (1 - x_i)(1 - y_i)) .$$

This polynomial is the multilinear extension of the "equality function" that takes two n-bit inputs x and y and outputs 1 if and only if $x = y$. In most applications of the sum-check protocol to SNARK design, we actually have that the protocol is applied to a polynomial g of the form:

$$g(x) = \mathsf{eq}(r, x) \cdot \prod_{i=1}^{d} p_i(x) \tag{3}$$

for some multilinear polynomials p_i, where $p_i(x)$ is in a small base field for all $x \in \{0, 1\}^{\log n}$, and where $r \in \mathbb{F}^n$ is chosen at random from a larger extension field $\mathbb{F}$ by the verifier.

Our results in this manuscript substantially reduce the extension-field multiplications incurred by the sum-check prover in order to process the polynomials $p_1, \ldots, p_d$. However, in this work we do not address extension-field multiplications introduced by the $\mathsf{eq}(r, x)$ factor. The main source of such multiplications is the prover's need to compute the values

$$A = \{\mathsf{eq}(r, x) \colon x \in \{0, 1\}^n\}.$$

This requires about n $\mathbb{F}$-multiplications using standard memoization-based procedures [VSBW13] (see [Tha22, Lemma 3.8] for details).

In a companion work, Dao and Thaler [DT24] reduce this cost for the prover when $\mathbb{F}$ has characteristic two and is constructed as a tower extension of a smaller base field, which is exactly the setting considered in Binius [DP23b] and FRI-Binius [DP24]. Specifically, let $\mathbb{F}$ be a degree-2^k extension of $\mathbb{B}$. Dao and Thaler show how to reduce the number of extension-field multiplications to compute the set A of $\mathsf{eq}(r, x)$ evaluations from roughly n down to $n/2^k$. In practice, this 2^k factor savings can be 128 (e.g., when the base field is $\mathsf{GF}[2]$ and the extension field is $\mathsf{GF}[2^{128}]$). In this setting, our work, when combined with the result of Dao and Thaler [DT24], reduces the sum-check prover time by an order of magnitude or more.

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 $z > 0$. A degree-k extension field $\mathbb{F}$ of $\mathbb{B}$ is said to be 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 on 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 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 subfield-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 paper, 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 subfield-by-subfield 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 $\mathbb{F}$.

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 $O(k^{\log_2(3)}) = O(k^{1.58496\ldots})$ times more expensive than multiplications in the base field $\mathbb{F}$.[7]

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).

On real hardware, multiplication of extension field elements can be faster when using the standard monomial basis rather than a tower basis (though using 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. Particularly important examples are the so-called POLYVAL basis for $\mathsf{GF}[2^{128}]$, which is used by AES and hence natively supported by many CPUs[8], and Intel's Galois Field instruction set

[7] 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.

[8] See `https://docs.rs/polyval/latest/polyval/`.

(GFNI) which has native support for $\mathsf{GF}[2^8]$ multiplication. However, with FPGAs, the difference between multiplication in the tower vs. monomial bases is much smaller, with recent estimates indicating that multiplications in the monomial basis uses only 20% fewer resources than tower bases [DP23a].

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 $\mathsf{GF}[2]$, 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 for $\mathrm{GF}[2^8]$ 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. We are 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 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 $\mathbb{B}$. 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 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

$$s_j(c) = \sum_{x \in \{0,1\}^{\ell - j}} g(r_1, \ldots, r_{j-1}, c, x), \tag{4}$$

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 (5).
- 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

$$\sum_{x\in\{0,1\}^\ell} p(x) \cdot q(x). \tag{5}$$

2.3 Key lemmas 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 $(r_1, x') \in \mathbb{F} \times \mathbb{F}^{\ell-1}$,*

$$p(r_1, x') = r_1 \cdot p(1, x') + (1 - r_1) \cdot p(0, x'). \tag{6}$$

Proof. The right hand side of Equation (6) 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 $((r_1,\ldots,r_i),x') \in \mathbb{F}^i \times \mathbb{F}^{\ell-i}$,*

$$p(r_1,\ldots,r_i,x') = \sum_{S \subseteq i} \chi_S(r_1,\ldots,r_i)\cdot p(S,x'). \tag{7}$$

Proof. The right hand side of Equation (7) 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. □

Lemma 3. *Suppose $f\colon \mathbb{F}^\ell \to \mathbb{F}$ is an ℓ-variate multilinear polynomial over $\mathbb{F}$, and let $P\colon \mathbb{F}\times\mathbb{F} \to \mathbb{F}$ be a linear map. Then, for $y \in \mathbb{F}^{i-1}$ and given $x \in \mathbb{F}^{\ell-i}$, the polynomial $F(y) := P(f(y,0,x), f(y,1,x))$ is also a multilinear polynomial.*

Proof. Since f is a multilinear polynomial, $f(y,0,x)$ and $f(y,1,x)$ are multilinear polynomials in $y \in \mathbb{F}^{i-1}$ for given $x \in \mathbb{F}^{n-i}$. As the linear combination of multilinear polynomials is also a multilinear polynomial, and $F(y)$ is a linear combination of $f(y,0,x)$ and $f(y,1,x)$, F is a multilinear polynomial in y. □

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: $\widetilde{\mathsf{eq}}_\ell(x,y) = \prod_{j=1}^{\ell}(x_jy_j + (1-x_j)(1-y_j))$. $\widetilde{\mathsf{eq}}_\ell$ 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 The case of $d = 2$.

Consider applying the sum-check polynomial to $g(x) = p(x)\cdot q(x)$. 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,

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

and similarly for

$$s_1(1) = \sum_{x\in\{0,1\}^{\ell-1}} p(1,x)\cdot q(1,x) = \sum_{x\in\{0,1\}^{\ell-1}} A(1,x)\cdot B(1,x). \tag{9}$$

To compute $s_1(2)$, by Lemma 1,

$$s_1(2) = \sum_{x \in \{0,1\}^{\ell-1}} p(2,x) \cdot q(2,x) = \sum_{x \in \{0,1\}^{\ell-1}} ((1-2) \cdot p(0,x) + 2 \cdot p(1,x)) \cdot ((1-2) \cdot q(0,x) + 2 \cdot q(1,x)). \tag{10}$$

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

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

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

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,

$$\begin{aligned} s_2(2) &= \sum_{x \in \{0,1\}^{\ell-2}} p(r_1,2,x) \cdot q(r_1,2,x) \\ &= \sum_{x \in \{0,1\}^{\ell-2}} ((1-2) \cdot p(r_1,0,x) + 2 \cdot p(r_1,1,x)) \cdot ((1-2) \cdot q(r_1,0,x) + 2 \cdot q(r_1,1,x)) \\ &= \sum_{x \in \{0,1\}^{\ell-2}} ((1-2) \cdot A[0,x] + 2 \cdot A[1,x]) \cdot ((1-2) \cdot B[0,x] + 2 \cdot B[1,x]). \end{aligned} \tag{11}$$

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 for each $x \in \{0,1\}^{\ell-2}$ as follows.

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

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 $n/2^i$ 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:

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

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 $g(x) = p_1(x) \cdot p_2(x) \cdot \cdots \cdot p_d(x)$. 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 $d(d-1)n/2^i$ 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 sum of $n/2^i$ terms (one for each input in $\{0,1\}^{\ell-i}$), with each term equal to a product of d ee elements. At the end of round i, updating all d arrays costs $d \cdot n/2^i$ ee multiplications. Hence, the cost of Algorithm 1 across all ℓ rounds is:

$$\left(\frac{d(d-1)n}{2} \cdot \mathsf{bb} + \frac{dn}{2} \cdot \mathsf{be}\right) + \sum_{i=2}^{\ell} \frac{d^2 n}{2^i} \cdot \mathsf{ee} \;\leq\; \left(\frac{(d^2-1)n}{2} \cdot \mathsf{bb} + \frac{dn}{2} \cdot \mathsf{be}\right) + \frac{d^2 n}{2} \cdot \mathsf{ee}. \tag{12}$$

In Expression (12), on the left side of the inequality, the expression before the sum accounts for computing the round 1 message $s_1(0)$, $s_1(1)$, ... $s_1(d)$,[9] and the following array update. The sum in Expression (12) 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.[10]

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.[11] 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.

[9] Evaluating $s_1(i)$ requires $(d-1) \cdot n/2$ base field multiplications for any $i \in \{0, 1, \ldots, d\}$. Here, $n/2$ is the number of terms in the sum defining s_1, see Equation (4).

[10] More precisely, the number of field operations performed by the prover in each round is linear in the *sparsity* m of p and q, i.e., 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 paper.

[11] 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.

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

$$\begin{aligned}
C[y,0,x] &= \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \ldots, r_{i-1}) \cdot p(y,0,x),\\
C[y,1,x] &= \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \ldots, r_{i-1}) \cdot p(y,1,x),\\
D[y,0,x] &= \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \ldots, r_{i-1}) \cdot q(y,0,x),\\
D[y,1,x] &= \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \ldots, r_{i-1}) \cdot q(y,1,x),\\
E[y,x] &= \widetilde{\mathsf{eq}}_{i-1}(y, r_1, \ldots, r_{i-1}) \left(-p(y,0,x) + 2p(y,1,x)\right),\\
F[y,x] &= -\widetilde{\mathsf{eq}}_{i-1}(y, r_1, \ldots, r_{i-1}) \left(-q(y,0,x) + 2q(y,1,x)\right).
\end{aligned}$$

Lemma 2 implies that

$$\begin{aligned}
\sum_{y \in \{0,1\}^{i-1}} C[y,0,x] &= p(r_1, \ldots, r_{i-1}, 0, x),\\
\sum_{y \in \{0,1\}^{i-1}} C[y,1,x] &= p(r_1, \ldots, r_{i-1}, 1, x),\\
\sum_{y \in \{0,1\}^{i-1}} D[y,0,x] &= q(r_1, \ldots, r_{i-1}, 0, x),\\
\sum_{y \in \{0,1\}^{i-1}} D[y,1,x] &= q(r_1, \ldots, r_{i-1}, 1, x),\\
\sum_{y \in \{0,1\}^{i-1}} E[y,x] &= p(r_1, \ldots, r_{i-1}, 2, x),\\
\sum_{y \in \{0,1\}^{i-1}} F[y,x] &= q(r_1, \ldots, r_{i-1}, 2, x).
\end{aligned}$$

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 $\widetilde{\mathsf{eq}}_{i-2}(y, r_1, \ldots, r_{i-2})$ 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. 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

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

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

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

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 Equation (13). 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 to be computed from $d+1$ to d.

4 Optimized provers for extension fields: Algorithms 3 and 4

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 (11), although $p(r_1, x)$ and $q(r_1, x)$ for $x \in \{0,1\}^{\ell-1}$ are extension field elements, 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 $p(0,x) \cdot q(0,x)$, $p(1,x) \cdot q(1,x)$, $p(0,x) \cdot q(1,x)$, $p(1,x) \cdot q(0,x)$. Using schoolbook multiplication, we compute

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

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. Further, we can use the Karatsuba trick to combine the cross-terms $p(0,x) \cdot q(1,x)$ and $p(1,x) \cdot q(0,x)$ into a single product $(p(0,x) + q(0,x)) \cdot (p(1,x) + q(1,x))$.

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

Since the first two products have already been computed, we only need to compute the new product $(p(0,x) + q(0,x)) \cdot (p(1,x) + q(1,x))$. We call this Algorithm 4.

4.1 Algorithm 3

4.1.1 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

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

To compute $s_1(2)$, as per Equation (10),

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

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 $p(y) \cdot q(y)$ 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:

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

Expanding the x'th term of this sum, using schoolbook multiplication, yields:

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

Hence, every term equals a previously-computed product, times either r_1^2, $r_1(1-r_1)$, or $(1-r_1)^2$.[12]

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

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

$2n \cdot \mathsf{bb}$ 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,

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

This expression in turn equals

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

where

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

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:

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

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 $p(x) \cdot q(\bar{x})$ 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

$$\widetilde{\mathsf{eq}}((y_1, \ldots, y_i), (r_1, \ldots, r_{i-1}, 2)) \cdot \widetilde{\mathsf{eq}}((y'_1, \ldots, y'_i), (r_1, \ldots, r_{i-1}, 2)). \tag{14}$$

For each $j = 1, \ldots, i-1$, 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 $z = (z_1, \ldots, z_{i-1}) \in \{0,1,2\}^{i-1}$, 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[13]

$$D[z] \leftarrow r_{i-1}^{z_{i-1}} \cdot D[z_1, \ldots, z_{i-2}] + (1 - r_{i-1})^{2 - z_{i-1}} \cdot D[z_1, \ldots, z_{i-2}]. \tag{15}$$

Thus, 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. In Sections 5 and 6, we work out the optimal rounds at which one should switch over from Algorithm 3 to Algorithm 1 and/or Algorithm 2.

4.1.2 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.

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

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

The prover computes $s_1(2)$ as follows. Per Equation (10),

$$\begin{aligned} s_1(2) &= \sum_{x \in \{0,1\}^{\ell-1}} p(2,x) \cdot q(2,x) \cdot h(2,x) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} ((1-2) \cdot p(0,x) + 2 \cdot p(1,x)) \cdot ((1-2) \cdot q(0,x) + 2 \cdot q(1,x)) \cdot ((1-2) \cdot h(0,x) + 2 \cdot h(1,x)) \\ &= \sum_{x \in \{0,1\}^{\ell-1}} z(x), \end{aligned} \tag{16}$$

where

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

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

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).

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 $p(y) \cdot q(y)$ 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 (11),

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

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', y'' may or may not differ from y in their first bit.

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

Expanding the x'th term of this sum yields:

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

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

$16n \cdot \mathsf{bb}$ 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,

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

This expression can be written as

$$s_2(2) := \sum_{x \in \{0,1\}^{\ell-2}} F(x) \cdot G(x) \cdot H(x)$$

where

$$\begin{aligned} F(x) &= ((1-r_1) \cdot ((1-2)p(0,0,x) + 2p(0,1,x)) + r_1 \cdot ((1-2)p(1,0,x) + 2p(1,1,x))) \,, \\ G(x) &= ((1-r_1) \cdot ((1-2)q(0,0,x) + 2q(0,1,x)) + r_1 \cdot ((1-2)q(1,0,x) + 2q(1,1,x))) \,, \\ H(x) &= ((1-r_1) \cdot ((1-2)h(0,0,x) + 2h(0,1,x)) + r_1 \cdot ((1-2)h(1,0,x) + 2h(1,1,x))) \,. \end{aligned}$$

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[14] 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 $p(y) \cdot q(y')$, 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'.

[14] $4n$ out of the remaining $12n$ terms agree on their second bit.

4.1.3 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 number of bb multiplications is

$$\left((d-1) + 2^j + 4^j + 8^j + \cdots + 2^{(d-1)j}\right) n. \tag{17}$$

4.2 Algorithm 4

The round polynomial computation for sum-check involves multiplication of multilinear polynomials. Due to multilinearity, each of the polynomials being multiplied can be expressed as a linear combination of base-field elements. To multiply these multilinear polynomials, Algorithm 3 uses the schoolbook multiplication algorithm. The main idea of Algorithm 4 is to use the Toom-Cook multiplication algorithm to further minimize the number of base-field multiplications.

Toom-Cook Multiplication. Let $p(x)$ and $q(x)$ be two univariate polynomials of the form

$$p(x) = (1-x) \cdot a_0 + x \cdot a_1,$$
$$q(x) = (1-x) \cdot b_0 + x \cdot b_1,$$

such that the coefficients $a_0, a_1, b_0, b_1 \in \mathbb{B}$ are base-field elements. The objective is to compute $r(y) = p(y) \cdot q(y)$ for some $y \in \mathbb{F}$. The degree d of the resulting polynomial $r(x)$ is 2. The Toom-Cook approach is to evaluate $p(\cdot)$ and $q(\cdot)$ at $(d+1) = 3$ points, multiply those evaluations for each point and then interpolate to get the monomial form of the resulting polynomial $r(x)$. For $d = 2$, we choose the set of evaluation points to be $E = \{0, 1, \infty\}$[15]. Thus, we compute the evaluations of r as:

$$\begin{aligned} r(0) &= p(0) \cdot q(0) &&= a_0 \cdot b_0, \\ r(1) &= p(1) \cdot q(1) &&= a_1 \cdot b_1, \\ r(\infty) &= p(\infty) \cdot q(\infty) &&= (a_1 - a_0) \cdot (b_1 - b_0). \end{aligned}$$

Computing each evaluation requires $(d-1) = 1$ multiplications in the base-field. So computing all evaluations of r on the given evaluation set requires $(d+1)(d-1) = 3$ multiplications in the base-field. To get the monomial form of $r(x) = r_0 + r_1 \cdot x + r_2 \cdot x^2$, we multiply the evaluations with an interpolation matrix $I_d \equiv I_2$.

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} := \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}}_{I_2} \begin{bmatrix} r(0) \\ r(1) \\ r(\infty) \end{bmatrix}$$

[15] To "evaluate" a polynomial $p(x)$ at infinity actually means to compute $\lim_{x \to \infty} \frac{p(x)}{x^{\deg(p)}}$. This implies that $p(\infty)$ is always the value of its highest-degree coefficient.

We can evaluate the polynomial r on $y \in \mathbb{F}$ by linearly combining its evaluations without having to compute the monomial form, i.e.,

$$r(y) = \begin{bmatrix}1 & y & y^2\end{bmatrix} \begin{bmatrix}r_0 \\ r_1 \\ r_2\end{bmatrix} = \begin{bmatrix}1 & y & y^2\end{bmatrix} \begin{bmatrix}1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}r(0) \\ r(1) \\ r(\infty)\end{bmatrix} = \begin{bmatrix}(y-1) & y & (y^2-y)\end{bmatrix} \begin{bmatrix}r(0) \\ r(1) \\ r(\infty)\end{bmatrix}.$$

To simplify notation, we define the interpolation maps

$$L_1(y) := (y-1), \qquad L_2(y) := y, \qquad L_3(y) := (y^2-y), \tag{18}$$

and define evaluation maps $P_j(a,b) := (1-e_j)\cdot a + e_j \cdot b$ for $e_j \in E$ for each $j \in \{1,2,3\}$. Using these evaluation maps, we can write the evaluations at $x=0$ as $p(0) = P_1(a_0,a_1)$ and $q(0) = P_1(b_0,b_1)$ and thus $r(0) = p(0)\cdot q(0) = P_1(a_0,a_1)\cdot P_1(b_0,b_1)$. Similarly, we have $r(1) = P_2(a_0,a_1)\cdot P_2(b_0,b_1)$ and $r(\infty) = P_3(a_0,a_1)\cdot P_3(b_0,b_1)$. Therefore, we can write $r(y)$ as

$$r(y) = \begin{bmatrix}(y-1) & y & (y^2-y)\end{bmatrix} \begin{bmatrix}r(0) \\ r(1) \\ r(\infty)\end{bmatrix} = \sum_{j=1}^{3} L_j(y)\cdot\big(P_j(a_0,a_1)\cdot P_j(b_0,b_1)\big). \tag{19}$$

Therefore, we can compute $r(y)$ with 3 base-field multiplications. For the same setting, the Karatsuba method also requires 3 base-field multiplications. The advantage of the Took-Cook[16] algorithm is that it neatly generalizes for multiplication of more than two polynomials of arbitrary degree. In fact, the Karatsuba algorithm is a special case of the Toom-Cook algorithm for $d=2$.

On multiplying d univariate polynomials $p_1, p_2, \ldots, p_d$ each of degree 1, the degree of the resulting polynomials is d. In this case, we need to choose an evaluation set E of size $(d+1)$. The total number of base-field multiplications required to compute the polynomial product is $(d+1)(d-1)$. Expression (19) trivially generalises[17] for the multiplying d polynomials as

$$r(y) = \frac{1}{\Delta_d}\cdot\sum_{j=1}^{d} L_j(y)\cdot\prod_{k=1}^{m} P_j(a_{k,0},a_{k,1}). \tag{20}$$

where $p_k(x) = (1-x)\cdot a_{k,0} + x\cdot a_{k,1}$ for $k \in \{1,2,\ldots,d\}$ and $\Delta_d = (d-1)!$ is an integer constant.

4.2.1 Details of Algorithm 4 when $d=2$.

Let $p_1(x)$ and $p_2(x)$ be multilinear polynomials defined over the base-field $\mathbb{B}$. Similarly to algorithm 3, we start by pre-computing the product $p_1(x)\cdot p_2(x)$ for all $x \in \{0,1\}^\ell$ and store it in an array C. This costs $n\cdot \mathsf{bb}$ multiplications.

Round 1. To compute the first round polynomial s_1, we define $G_1(c,x)$ for $x \in \{0,1\}^{\ell-1}$ as

$$G_1(c,x) := \prod_{k=1}^{2} p_k(c,x) = \prod_{k=1}^{2}\left((1-c)\cdot p_k(0,x) + c\cdot p_k(1,x)\right).$$

Using Expression (19) for multiplying two polynomials using Toom-Cook multiplication,

$$G_1(c,x) = \sum_{j_1=1}^{3} L_{j_1}(c)\cdot\prod_{k=1}^{2} P_{j_1}\left(p_k(0,x),p_k(1,x)\right).$$

[16]Conventionally, Toom-Cook is a multiplication algorithm for multiplying two large integers. In our case, we extend the Toom-Cook framework to efficiently multiply more than two linear polynomials.

[17]For $d>2$, the interpolation matrix I_d typically contains rational numbers. Dealing with rational numbers in fields requires inversion operations. To avoid any rational numbers in the interpolation maps, we multiply the interpolation matrix by a constant $\Delta_d := (d-1)!$ and thus a term of $\frac{1}{\Delta_d}$ appears in the expression of $r(y)$.

The interpolation maps L_1, L_2, L_3 are defined in the Expression (18) and the evaluation maps are $P_1(a,b) = a,\ P_2(a,b) = b$ and $P_3(a,b) = (b-a)$. Since P_1 and P_2 are identity maps over a and b, we have

$$\prod_{k=1}^{2} P_1(p_k(0,x), p_k(1,x)) = \prod_{k=1}^{2} p_k(0,x) \equiv C[0,x],$$
$$\prod_{k=1}^{2} P_2(p_k(0,x), p_k(1,x)) = \prod_{k=1}^{2} p_k(1,x) \equiv C[1,x].$$

Thus, we do not need to recompute the products corresponding to $j_1 \in \{1,2\}$ for the first round. The only additional product we need to compute is for $j_1 = 3$ and we store it in a new array S_1 as

$$S_1(x) := \prod_{k=1}^{2} (p_k(1,x) - p_k(0,x))$$

for each $x \in \{0,1\}^{\ell-1}$. This requires $n/2$ bb multiplications. Given $G_1(c,x)$ for $x \in \{0,1\}^{\ell-1}$, the first round polynomial can be computed as

$$s_1(c) := \sum_{x \in \{0,1\}^{\ell-1}} G_1(c,x).$$

Round 2. We define $G_2(c,x)$ for $x \in \{0,1\}^{\ell-2}$ as

$$\begin{aligned}
G_2(c,x) &:= \prod_{k=1}^{2} p_k(r_1,c,x), \\
&= \prod_{k=1}^{2} \left((1-c)\cdot p_k(r_1,0,x) + c \cdot p_k(r_1,1,x)\right), && \text{(using Lemma 1)} \\
&= \sum_{j_1=1}^{3} L_{j_1}(c) \cdot \prod_{k=1}^{2} P_{j_1}\left(p_k(r_1,0,x), p_k(r_1,1,x)\right). && \text{(using Equation (19) with } y = c)
\end{aligned}$$

We can then unroll the evaluation maps $P_{j_1}\left(p_k(r_1,0,x), p_k(r_1,1,x)\right)$ recursively as

$$\begin{aligned}
P_{j_1}\left(p_k(r_1,0,x), p_k(r_1,1,x)\right) &= \bar{e}_{j_1} \cdot p_k(r_1,0,x) + e_{j_1} \cdot p_k(r_1,1,x) \\
&= \bar{e}_{j_1} \cdot (\bar{r}_1 \cdot p_k(0,0,x) + r_1 \cdot p_k(1,0,x)) + e_{j_1} \cdot (\bar{r}_1 \cdot p_k(0,1,x) + r_1 \cdot p_k(1,1,x)) \\
&= \bar{r}_1 \cdot (\bar{e}_{j_1} \cdot p_k(0,0,x) + e_{j_1} \cdot p_k(0,1,x)) + r_1 \cdot (\bar{e}_{j_1} \cdot p_k(1,0,x) + e_{j_1} \cdot p_k(1,1,x)) \\
&= \bar{r}_1 \cdot P_{j_1}(p_k(0,0,x), p_k(0,1,x)) + r_1 \cdot P_{j_1}(p_k(1,0,x), p_k(1,1,x))
\end{aligned}$$

where $\bar{b} := (1-b)$. Plugging this back into the expression of $G_2(c,x)$, we get

$$\begin{aligned}
G_2(c,x) &= \sum_{j_1=1}^{3} L_{j_1}(c) \cdot \prod_{k=1}^{2} P_{j_1}\left(p_k(r_1,0,x), p_k(r_1,1,x)\right) \\
&= \sum_{j_1=1}^{3} L_{j_1}(c) \cdot \prod_{k=1}^{2} \left(\bar{r}_1 \cdot P_{j_1}(p_k(0,0,x), p_k(0,1,x)) + r_1 \cdot P_{j_1}(p_k(1,0,x), p_k(1,1,x))\right)
\end{aligned}$$

Again using Expression (19) for multiplying two polynomials and setting $y = r_1$,

$$G_2(c,x) = \sum_{j_1=1}^{3} \sum_{j_2=1}^{3} L_{j_1}(c) \cdot L_{j_2}(r_1) \cdot \prod_{k=1}^{2} \mathsf{merkle}_2(p_k, j_1, j_2), \tag{21}$$

where $\mathsf{merkle}_2(p_k, j_1, j_2)$ denotes the result of recursive application of the linear maps P_{j_1}, P_{j_2} in a merkle-tree like fashion on p_k (see Figure 1).

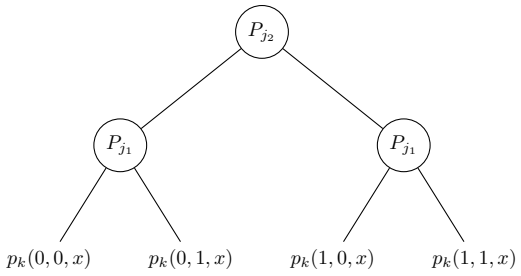


Figure 1: Merkle-tree structure for round 2 computation with linear maps P_{j_1} and P_{j_2} at respective levels.

Similarly to the first round, we can reuse some of the computation from the previous rounds. For $j_2 \in \{1,2,3\}$ and $j_1 = \{1,2\}$ we can simplify the products as shown in Table 1. Hence, we compute the remainder of $9-6=3$ products and store them in a new array S_2. This costs $3 \cdot 2^{\ell-2} = 3n/4$ bb multiplications. The second round polynomial can be computed as

$$s_2(c) := \sum_{x \in \{0,1\}^{\ell-2}} G_2(c,x)$$

.

Table 1: Product terms in round 2 that also appeared in round 1 for each $x \in \{0,1\}^{\ell-2}$.

j_2	j_1	Product	Equals
1	1	$\prod_{k=1}^{2} p_k(0,0,x)$	$C[0,0,x]$
1	2	$\prod_{k=1}^{2} p_k(0,1,x)$	$C[0,1,x]$
2	1	$\prod_{k=1}^{2} p_k(1,0,x)$	$C[1,0,x]$
2	2	$\prod_{k=1}^{2} p_k(1,1,x)$	$C[1,1,x]$
3	1	$\prod_{k=1}^{2} (p_k(0,0,x) - p_k(1,0,x))$	$S_1[0,x]$
3	2	$\prod_{k=1}^{2} (p_k(0,1,x) - p_k(1,1,x))$	$S_1[1,x]$

Round i. Following the expression (23) of $G_2(c,x)$, define $G_i(c,x)$ for round i

$$\begin{aligned} G_i(c,x) &:= \prod_{k=1}^{2} p_k(r_1, r_2, \ldots, r_{i-1}, c, x) \\ &= \prod_{k=1}^{2} (\bar{c} \cdot p_k(r_1, \ldots, r_{i-1}, 0, x) + c \cdot p_k(r_1, \ldots, r_{i-1}, 1, x)) \\ &= \sum_{j_1=1}^{3} L_{j_1}(c) \cdot \prod_{k=1}^{2} P_{j_1}\left(p_k(r_1, \ldots, r_{i-1}, 0, x), p_k(r_1, \ldots, r_{i-1}, 1, x)\right). \end{aligned}$$

Owing to Lemma 3, the polynomial

$$F_k(r_1, \ldots, r_{i-1}) := P_{j_1}\left(p_k(r_1, \ldots, r_{i-1}, 0, x), p_k(r_1, \ldots, r_{i-1}, 1, x)\right)$$

is multilinear in $(r_1, r_2, \ldots, r_{i-1}) \in \mathbb{F}^{i-1}$ for $k \in \{1,2\}$. Using equation (19), we write

$$\prod_{k=1}^{2} F_k(r_1, r_2, \ldots, r_{i-1}) = \sum_{j_2=1}^{3} L_{j_2}(r_{i-1}) \cdot \prod_{k=1}^{2} P_{j_2}(F_k(r_1, \ldots, r_{i-2}, 0), F_k(r_1, \ldots, r_{i-2}, 1)).$$

Substituting this product in the expression of $G_i(c, x)$, we get

$$G_i(c,x) = \sum_{j_1=1}^{3} \sum_{j_2=1}^{3} L_{j_1}(c) L_{j_2}(r_{i-1}) \cdot \prod_{k=1}^{2} P_{j_2}\left(F_k(r_1, \ldots, r_{i-2}, 0), F_k(r_1, \ldots, r_{i-2}, 1)\right).$$

We can now repeat the same process of invoking Lemma 3 followed by Equation (19) sequentially on each of the remaining variables $(r_{i-2}, \ldots, r_1)$ to get:

$$G_i(c,x) = \sum_{j_1=1}^{3} \sum_{j_2=1}^{3} \cdots \sum_{j_i=1}^{3} L_{j_i}(r_1) \cdots L_{j_2}(r_{i-1}) \cdot L_{j_1}(c) \cdot \prod_{k=1}^{2} \mathsf{merkle}_i(p_k, j_1, \ldots, j_i) \tag{22}$$

where $\mathsf{merkle}_i(f_i, j_1, \ldots, j_i)$ denotes recursive application of the linear maps $P_{j_1}, P_{j_2}, \ldots, P_{j_i}$ in a merkle-tree like fashion on the polynomial p_k. See Figure 2 for an illustration.

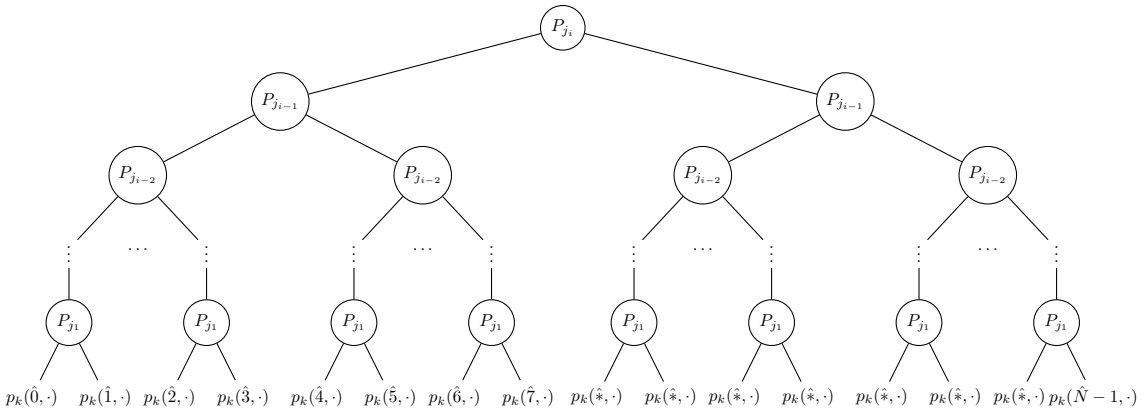


Figure 2: Merkle-tree structure for round i computation. Note $\hat{m}$ denotes the i-bit representation of an integer m.

4.2.2 Details of Algorithm 4 when $d = 3$.

Let $p_1(x), p_2(x), p_3(x)$ be the multilinear polynomials. We start by computing the the product $p_1(x) \cdot p_2(x) \cdot p_3(x)$ for all $x \in \{0,1\}^\ell$ and store it in an array C. This costs $2n \cdot \mathsf{bb}$ multiplications.

The algorithm for $d = 3$ closely follows the $d = 2$ case with the evaluation set now being $E = \{0, 1, -1, \infty\}$. The evaluation maps P_j and the interpolation maps L_j for each $j \in \{1, 2, 3, 4\}$ change accordingly.

$$P_1(a,b) = a, \qquad P_2(a,b) = b, \qquad P_3(a,b) = 2a - b, \qquad P_4(a,b) = b - a,$$
$$L_1(y) := 2 \cdot (1 - y^2), \qquad L_2(y) := (y^2 + y), \qquad L_3(y) := (y^2 - y), \qquad L_4(y) := 2 \cdot (y^3 - y).$$

Round 1. We define $G_1(c, x)$ for $x \in \{0,1\}^{\ell-1}$ as

$$\begin{aligned} G_1(c,x) &:= \prod_{k=1}^{3} p_k(c,x) = \prod_{k=1}^{3} \left((1-c) \cdot p_k(0,x) + c \cdot p_k(1,x)\right). \\ &= \frac{1}{2} \cdot \sum_{j_1=1}^{4} L_{j_1}(c) \cdot \prod_{k=1}^{3} P_{j_1}\left(p_k(0,x), p_k(1,x)\right). \qquad \text{(from Equation (20))} \end{aligned}$$

As the evaluation maps P_1 and P_2 are identity maps, we don't need to recompute the products corresponding to $j_1 \in \{1, 2\}$. For $j_1 \in \{3, 4\}$, we compute two 3-way products and store them in arrays A_1 and S_1 respectively

$$A_1(x) := \prod_{k=1}^{3} (2p_k(0,x) - p_k(1,x)), \qquad S_1(x) := \prod_{k=1}^{3} (p_k(1,x) - p_k(0,x)),$$

for each $x \in \{0,1\}^{\ell-1}$. This requires $2 \cdot 2 \cdot 2^{\ell-1} = 2n \cdot \mathsf{bb}$ multiplications. Since $G_1(c,x)$ has a multiplicand of $1/2$, the first round polynomial $s_1(c)$ would also carry over the same multiplicand. Multiplication by a factor of 2^{-1} could become an overhead cost for the prover. The prover can avoid this cost by sending a modified round polynomial $s_1'(c) = 2 \cdot s_1(c)$.

Round 2. We define $G_2(c,x)$ for $x \in \{0,1\}^{\ell-2}$ as

$$\begin{aligned} G_2(c,x) &:= \prod_{k=1}^{3} p_k(r_1, c, x) = \prod_{k=1}^{3} \left((1-c) \cdot p_k(r_1, 0, x) + c \cdot p_k(r_1, 1, x)\right) \\ &= \frac{1}{2} \cdot \sum_{j_1=1}^{4} L_{j_1}(c) \cdot \prod_{k=1}^{3} P_{j_1}\left(p_k(r_1,0,x), p_k(r_1,1,x)\right) \qquad \text{(from Equation (20))} \end{aligned}$$

Similarly to the second round of the $d = 2$ case, unrolling $P_{j_1}\left(p_k(r_1,0,x), p_k(r_1,1,x)\right)$ recursively, we get

$$G_2(c,x) = \frac{1}{2^2} \sum_{j_1=1}^{4} \sum_{j_2=1}^{4} L_{j_1}(c) \cdot L_{j_2}(r_1) \cdot \prod_{k=1}^{3} \mathsf{merkle}_2(p_k, j_1, j_2) \tag{23}$$

where $\mathsf{merkle}_2(p_k, j_1, j_2)$ denotes the result of recursive application of the linear maps P_{j_1}, P_{j_2} in a merkle-tree like fashion on p_k (see Figure 1).

We can reuse some of the computation from the previous rounds, i.e., for $j_2 \in \{1,2,3,4\}$ and $j_1 = \{1,2\}$ we can simplify the products as shown in Table 2. Hence, we compute the remainder of $16 - 8 = 8$ products and store them in arrays A_2 and S_2. This costs $8 \cdot 2 \cdot 2^{\ell-2} \cdot \mathsf{bb}$ multiplications. The second round polynomial can be computed as

$$s_2(c) := \sum_{x \in \{0,1\}^{\ell-2}} G_2(c,x).$$

The prover again sends a modified round polynomial $s_2'(c) = 2^2 \cdot s_2(c)$ to avoid multiplying by the scaling factor 2^{-2} as per equation (23).

Table 2: Product terms in round 2 that also appeared in round 1 for $x \in \{0,1\}^{\ell-2}$.

j_2	j_1	Product	Equals
1	1	$\prod_{k=1}^{3} p_k(0,0,x)$	$C[0,0,x]$
1	2	$\prod_{k=1}^{3} p_k(0,1,x)$	$C[0,1,x]$
2	1	$\prod_{k=1}^{3} p_k(1,0,x)$	$C[1,0,x]$
2	2	$\prod_{k=1}^{3} p_k(1,1,x)$	$C[1,1,x]$
3	1	$\prod_{k=1}^{3} (2p_k(0,0,x) - p_k(1,0,x))$	$A_1[0,x]$
3	2	$\prod_{k=1}^{3} (2p_k(0,1,x) - p_k(1,1,x))$	$A_1[1,x]$
4	1	$\prod_{k=1}^{3} (p_k(1,0,x) - p_k(0,0,x))$	$S_1[0,x]$
4	2	$\prod_{k=1}^{3} (p_k(1,1,x) - p_k(0,1,x))$	$S_1[1,x]$

Round i. Following the expression (23) of $G_2(c,x)$, define $G_i(c,x)$ for round i

$$\begin{aligned} G_i(c,x) &:= \prod_{k=1}^{3} p_k(r_1, r_2, \ldots, r_{i-1}, c, x), \\ &= \prod_{k=1}^{3} \left(\bar{c} \cdot p_k(r_1, \ldots, r_{i-1}, 0, x) + c \cdot p_k(r_1, \ldots, r_{i-1}, 1, x)\right), \end{aligned}$$

Using Equation (20), we get

$$G_i(c,x) = \frac{1}{2}\sum_{j_1=1}^{4} L_{j_1}(c)\cdot\prod_{k=1}^{3} P_{j_1}\left(p_k(r_1,\ldots,r_{i-1},0,x), p_k(r_1,\ldots,r_{i-1},1,x)\right).$$

Similarly to the round i of the $d=2$ case, on recursively applying Lemma 3 followed by the Toom-Cook multiplication from Equation (20) for each of the variables $r_{i-1}, r_{i-2}, \ldots, r_1$, we get

$$G_i(c,x) = \frac{1}{2^i}\sum_{j_1=1}^{4}\sum_{j_2=1}^{4}\cdots\sum_{j_i=1}^{4} L_{j_i}(r_1)\cdots L_{j_2}(r_{i-1})\cdot L_{j_1}(c)\cdot\prod_{k=1}^{3}\mathsf{merkle}_i(p_k, j_1,\ldots,j_i) \tag{24}$$

where $\mathsf{merkle}_i(p_k, j_1,\ldots,j_i)$ denotes recursive application of the linear maps $P_{j_1}, P_{j_2},\ldots,P_{j_i}$ in a merkle-tree like fashion. See Figure 2 for an illustration. Lastly, the prover computes the p-th round polynomial as

$$s_i(c) := \sum_{x\in\{0,1\}^{\ell-i}} G_i(c,x).$$

To avoid multiplying with this scalar $2^{-i}\in\mathbb{B}$ in Equation (24), the prover sends a modified round polynomial $s'_i(c) := 2^i\cdot s_i(c)$. The verifier check for round i is modified as follows.

$$\begin{aligned} s_i(r_i) &\stackrel{?}{=} s_{i-1}(0)+s_{i-1}(1),\\ \implies 2^i\cdot s_i(r_i) &\stackrel{?}{=} 2^i\cdot(s_{i-1}(0)+s_{i-1}(1)),\\ \implies s'_i(r_i) &\stackrel{?}{=} 2\cdot(s'_{i-1}(0)+s'_{i-1}(1)). \end{aligned} \tag{25}$$

where s'_{i-1} and s'_i are the round polynomials sent by the prover in rounds $(i-1)$ and i respectively, and $r_i\in\mathbb{F}$ is the verifier challenge in round i. Note that the only additional work the verifier has to do in equation (25) is multiplication by the constant 2.

Total Computation. We start by computing the products $p_1(x)\cdot p_2(x)\cdot p_3(x)$ for each $x\in\{0,1\}^\ell$ which costs n 3-way products. In round i, we additionally compute $\frac{4^i}{2}\cdot\frac{n}{2^i}$ bb multiplications. Thus, the total number of bb multiplications is:

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

4.2.3 Algorithm 4 for general d

Let $p_1(x), p_2(x),\ldots,p_d(x)$ be d multilinear polynomials. The analysis for the round polynomial computation for a general d follows the $d=3$ case closely. Instead of 3-way products in the $d=3$ case, we need to pre-compute d-way products for general d. For example, the expression of $G_i(c,x)$ for round i for general d takes the form

$$G_i(c,x) = \frac{1}{\Delta_d^i}\cdot\sum_{j_1=1}^{d+1}\cdots\sum_{j_i=1}^{d+1} L_{j_k}(r_{i-k})\cdots L_{j_2}(r_{i-1})\cdot L_{j_1}(c)\cdot\prod_{k=1}^{d}\mathsf{merkle}_i(p_k,j_1,\ldots,j_i)$$

where $\mathsf{merkle}_i(p_k,j_1,\ldots,j_i)$ is the same recursive merkle tree structure with the layers (starting from leaves) that apply the linear maps $P_{j_1}, P_{j_2},\ldots,P_{j_i}$ respectively (see Figure 2). Similarly to the $d=3$ case, in round i, the prover sends a modified round polynomial $s'_i(c) := \Delta_d^i\cdot s_i(c)$ where $s_i(c) = \sum_{x\in\{0,1\}^{\ell-i}} G_i(c,x)$. The verification check in round i, therefore, changes to

$$s'_i(r_i) \stackrel{?}{=} \Delta_d\cdot(s'_{i-1}(0)+s'_{i-1}(1)). \tag{26}$$

Total Computation. In round i, given a set of indices $(j_1, j_2, \ldots, j_i) \in \{0, 1, \ldots, d\}^i$, the expression $\mathsf{merkle}_i(f_i, j_1, \ldots, j_i)$ is a d-way product, and hence it can be computed with $(d-1)$ bb multiplications. We need to compute such d-way products for each $(j_1, j_2, \ldots, j_i) \in \{0, 1, \ldots, d\}^i$ and $x \in \{0,1\}^{\ell-i}$, and hence the total number of bb multiplications needed in round i would be $(d-1) \cdot (d+1)^i \cdot 2^{\ell-i}$. But similarly to the $d = 3$ case, we can re-use computations from round $(i-1)$ in round i. The product terms corresponding to $j_1 \in \{0, d\}$ and $(j_2, \ldots, j_i) \in \{0, 1, \ldots, d\}^{i-1}$ and $x \in \{0,1\}^{\ell-i}$ can be re-used (see Table 2). Therefore, the total number of bb multiplications required in round i is

$$\begin{aligned}(d-1) \cdot \left((d+1)^i \cdot 2^{\ell-i} - 2(d+1)^{i-1} \cdot 2^{\ell-i}\right) &= (d-1) \cdot (d+1)^i \cdot \left(1 - \frac{2}{d+1}\right) \cdot 2^{\ell-i}, \\ &= (d-1) \cdot (d+1)^i \cdot \left(\frac{d-1}{d+1}\right) \cdot 2^{\ell-i}. \\ &= (d-1) \cdot \left(\frac{d+1}{2}\right)^i \cdot \left(\frac{d-1}{d+1}\right) \cdot n.\end{aligned}$$

and the total number of bb multiplications across all rounds would be

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

4.3 Comparison of Algorithms 3 and 4

The design of both algorithms 3 and 4 is based on the idea of minimizing extension-field multiplications at the cost of additional base-field multiplications. Algorithm 3 uses schoolbook multiplication to recursively expand the round polynomial while Algorithm 4 uses Toom-Cook multiplication for the same. Therefore, Algorithm 4 requires much less bb multiplications than Algorithm 3. The number of ee and be multiplications is the same for both these algorithms. An important advantage of Algorithm 4 is that it requires much less memory to store the witness products than Algorithm 3. This could become crucial when we need sum-check provers in memory-limited environments. We discuss the key differences of the two algorithms in terms of the memory requirements and the computational costs.

Memory. Algorithms 3 and 4 both demand significant memory to store the arrays containing products of the base-field elements. To avoid the storage requirements of these algorithms becoming prohibitive, we assume that we run these algorithms only upto round $i < \ell$. For degree d, Algorithm 3 maintains $(d-1)$ arrays $C_2, \ldots, C_d$ and the array C_k contains $2^{(k-1)\cdot i} \cdot n$ base-field elements at the end of round i for $k \in \{2, \ldots, d\}$. Therefore, at the end of round i with Algorithm 3, the total number of base-field products we need to store is:

$$\sum_{k=2}^{d} \left(2^i\right)^{(k-1)} \cdot n \leq 2^{i \cdot d} \cdot n.$$

For Algorithm 4, in round i, given a set of indices $(j_1, \ldots, j_i) \in \{0, 1, \ldots, d\}^i$ and $x \in \{0,1\}^{\ell-i}$, the witness product is $\prod_{k=0}^{d} \mathsf{merkle}_i(f_i, j_1, \ldots, j_i)$. Thus, the total number of base-field products we need to store is

$$(d+1)^i \cdot \frac{n}{2^i}.$$

Clearly, the memory requirement of both the algorithms grows linearly with the sum-check instance size n. Next, we discuss a simple but important optimization to make the memory requirement independent of n and only depend on the round i and degree d.

Optimizing Memory. For both Algorithms 3 and 4, the key idea is to compute round polynomials as a linear combination between the pre-computed base-field products (computed using witness polynomials) and the extension-field products (computed using round challenges). We can reduce the memory required to store the pre-computed products by computing inner products instead of hadamard products of the witness terms.

In round i, the round polynomial $s_i(c)$ typically can be expressed as

$$s_i(c) := \sum_{x \in \{0,1\}^{\ell-i}} \sum_{j_1} \cdots \sum_{j_i} \underbrace{C_i(j_1, \ldots, j_i, r_1, \ldots, r_{i-1}, c)}_{\text{Challenge term}} \cdot \underbrace{W_i(j_1, \ldots, j_i, x)}_{\text{Witness term}},$$

for $(j_1, j_2, \ldots, j_i) \in \{0, 1, \ldots, d\}^i$. The challenge term $C_i(j_1, \ldots, j_i, r_1, \ldots, r_{i-1}, c)$ does not depend on x and therefore, we can write

$$s_i(c) := \sum_{j_1} \cdots \sum_{j_i} C_i(j_1, \ldots, j_i, r_1, \ldots, r_{i-1}, c) \cdot \underbrace{\sum_{x \in \{0,1\}^{\ell-i}} W_i(j_1, \ldots, j_i, x)}_{\text{New witness term}}.$$

Instead of storing $W_i(j_1, \ldots, j_i, x)$ for each $x \in \{0,1\}^{\ell-i}$, we can store the inner-product[18]

$$W'_i(j_1, \ldots, j_i) = \sum_{x \in \{0,1\}^{\ell-i}} W_i(j_1, \ldots, j_i, x).$$

Thus, the number of products we need to store in Algorithms 3 and 4 is $(2^d)^i$ and $(d+1)^i$ respectively. Notice that the new memory requirement of both algorithms is independent of n. Furthermore, the products computed for round i can be reused (without new multiplications) in all rounds prior to round i.

We plot the theoretical memory requirements of both algorithms in Figure 3. We consider the Babybear field [BG23] for our analysis, so one base-field element requires 4 bytes of storage. For $p = 10$, we observe that Algorithm 4 requires a mere 4 MB memory while Algorithm 3 requires 4.2 GB of memory to store the pre-computed products. As i increases, Algorithm 4 outperforms Algorithm 3 in terms of the memory required to store the pre-computed witness products. Thus, for memory-constrained provers, Algorithm 4 allows for many more rounds to be processed using pre-computation as compared to Algorithm 3.

Figure 3: Pre-computed array memory (KB)

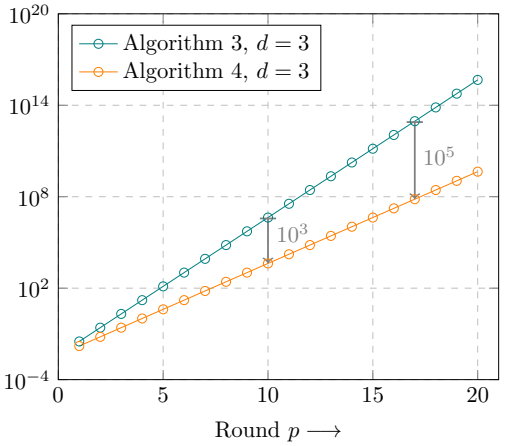
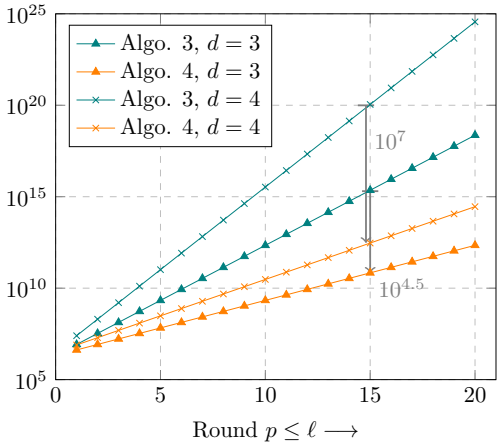


Figure 4: Number of bb multiplications



bb multiplications. The number of base-field multiplications is determined by the number of witness products that we need to pre-compute for both Algorithms 3 and 4. As discussed in the memory optimization section, we can pre-compute all the witness products for the round polynomial computation in round i and use them for rounds 1 through i. Therefore, the total number of such witness products is $2^{i \cdot d}$ and $(d+1)^i$ for Algorithms 3 and 4 respectively. Each such witness product is a result of inner-product of size $2^{\ell-i}$ consisting of d witness terms, requiring a total of $(d-1) \cdot 2^{\ell-i}$ bb multiplications. Hence, the total number of bb multiplications required for Algorithms 3 and 4 is $(d-1) \cdot 2^{i \cdot d} \cdot 2^{\ell-i}$ and $(d-1) \cdot (d+1)^i \cdot 2^{\ell-i}$ respectively.

[18] Typically, inner-product refers to the sum of element-wise products of *two* vectors. In this case, we use the term "inner-product" more generally to denote the sum of element-wise products of *more than two* vectors.

We plot the analytical number of bb multiplications for both algorithms for a fixed instance size $\ell = 20$ in Figure 4. As i increases, the number of bb multiplications grow faster for Algorithm 3 than Algorithm 4. For degree $d = 3$ and $p = 15$, the number of bb multiplications in Algorithm 4 is four orders of magnitude lesser than that of Algorithm 3. As the degree d increases, the difference between the number of bb multiplications in both the algorithms widens further. For the same $p = 15$ but degree $d = 4$, the number of bb multiplications in Algorithm 4 is 7 orders of magnitude lesser than that of Algorithm 3.

ee and be multiplications. In round i of Algorithm 3 (and 4), given the challenge $r_{i-1} \in \mathbb{F}$ from the previous round, we first need to compute $(1 - r_{i-1})^{d-j} \cdot r_{i-1}^j$ for each $j \in \{0, 1, \ldots, d\}$. We use binomial expansion to efficiently compute them.

$$(1 - r_{i-1})^{d-j} \cdot r_{i-1}^j = r_{i-1}^j - \binom{d-j}{1} \cdot r_{i-1}^{j+1} - \ldots + \binom{d-j}{d-j} \cdot r_{i-1}^d.$$

Thus, the challenge terms can be computed as a linear combination of the powers of the challenges r_{i-1}^j for $j = 0, 1, \ldots, d$. The binomial expansion constants $\binom{d-j}{k}$ for $k \in \{0, 1, \ldots, d-j\},\ j \in \{0, 1, \ldots, d\}$ would typically be small integers for small degree d. So we ignore the cost of multiplying the challenge powers with these constants. Thus, computing the powers of a given challenge requires $(d-1) \cdot$ ee multiplications.

In Algorithm 4, we need to additionally compute the interpolation maps using the challenge terms. The magnitude of the integers in the interpolation matrix can also considered to be small integers for a small degree d. Thus, we consider the cost of computing interpolation maps to be negligible for the prover. Once we have computed the challenge terms in round i of Algorithm 3 (and 4), we need $(d+1)^{i-1} \cdot$ ee multiplications to derive every possible product across variables $1, \ldots, i-1$. Thus, the total number of ee multiplications in round i is $(d-1) + (d+1)^{i-1}$ for both Algorithm 3 and 4. Analogously, we need $(d+1)^{i-1} \cdot$ be multiplications to multiply the results of the challenge products with the corresponding witness terms in both the algorithms.

5 Optimizing costs when bb, 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 3 or 4 for the early rounds of sum-check, then switch to Algorithm 2, then to Algorithm 1.

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 $(n - n/2^i) \cdot$ 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 $x \in \{0,1\}^{\ell-1}$,

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

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

$$\begin{aligned} A[x] &= r_1 r_2 \cdot p(1,1,x) + r_1 \bar{r}_2 \cdot p(1,0,x) + \bar{r}_1 r_2 \cdot p(0,1,x) + \bar{r}_1 \bar{r}_2 \cdot p(0,0,x) \\ &= r_1 r_2 \cdot (p(1,1,x) - p(1,0,x) - p(0,1,x) + p(0,0,x)) \ + \\ &\quad r_1 \cdot (p(1,0,x) - p(0,0,x)) + r_2 \cdot (p(0,1,x) - p(0,0,x)) + p(0,0,x). \end{aligned}$$

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 (or 4) 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:

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

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 $d(d-1) \cdot n/2$ ee multiplications. For $d = 3$ 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×. For $n = 2^{24}$, our algorithm does 434,000 ee multiplications, versus $3 \cdot 2^{24}$ 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. For memory-constrained provers, switching from Algorithm 4 to Algorithm 1 can occur later than switching from Algorithm 3 to Algorithm 1 because Algorithm 4 requires significantly less memory than Algorithm 3.

6 Optimizing costs when bb, 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 or 4 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 The case of 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 $p(x) \cdot q(\bar{x})$ in round j that were not already computed by round $j-1$ is to make use of those cross terms in future rounds, so 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 $j = 2$, the round polynomial $s_2(2)$ can be expressed as

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

where $G(x)$ is of the form $w(x) + r_1 z(x)$ and $H(x)$ is of the form $w'(x) + r_1 z'(x)$ 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 four bb multiplications for each of the $n/4$ terms of the sum, a factor-2 improvement over the $2n$ bb 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 multiplications to compute the A and B arrays used in Algorithm 1 at the end of round j.[19] Thus, the total prover cost will be:

$$(3/4) \cdot 2^j \cdot n \cdot \mathsf{bb} + (2 \cdot n - n/2^j + 3^j) \cdot \mathsf{be} + ((4/2^j) \cdot n + (j + 3^j)) \cdot \mathsf{ee}. \tag{27}$$

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

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

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 3 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 (17), if one switches from Algorithm 3 to Algorithm 1 at the end of round j, the total prover cost will be:

[19] Algorithm 3 also incurs at most an additional $j + 3^j$ ee and be multiplications in total over the first j rounds, per Equation (15). We include these terms in Expression (27) 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 3, 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	4	8	16	32	64	128	256
Optimal switchover round j	3	4	4	5	6	7	8
Prover cost in ee mults per term of sum	2.0	1.27	0.78	0.483	0.302	0.192	0.122
Prover cost in 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	1.31	1.85	2.83	4.43	6.92	10.8	16.7

Table 3: Cost of our prover implementation combining Algorithms 1 and 3, for $d = 2$, in terms of number of ee multiplications. We assume that be and ee multiplications are respectively k and $k^{1.585}$ times costlier than bb multiplications.

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

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 4. 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 $k = 128$.

Extension degree	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$
Optimal switchover round j	2	3	3	4	4
Prover cost in 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 4: Cost of Algorithm 3 (combined with Algorithm 1) for $d = 3$ in terms of number of ee multiplications, assuming ee multiplications are $k^{1.585}$ times more expensive than bb multiplications, k times more expensive than be multiplications.

7 Implementation and Experiments

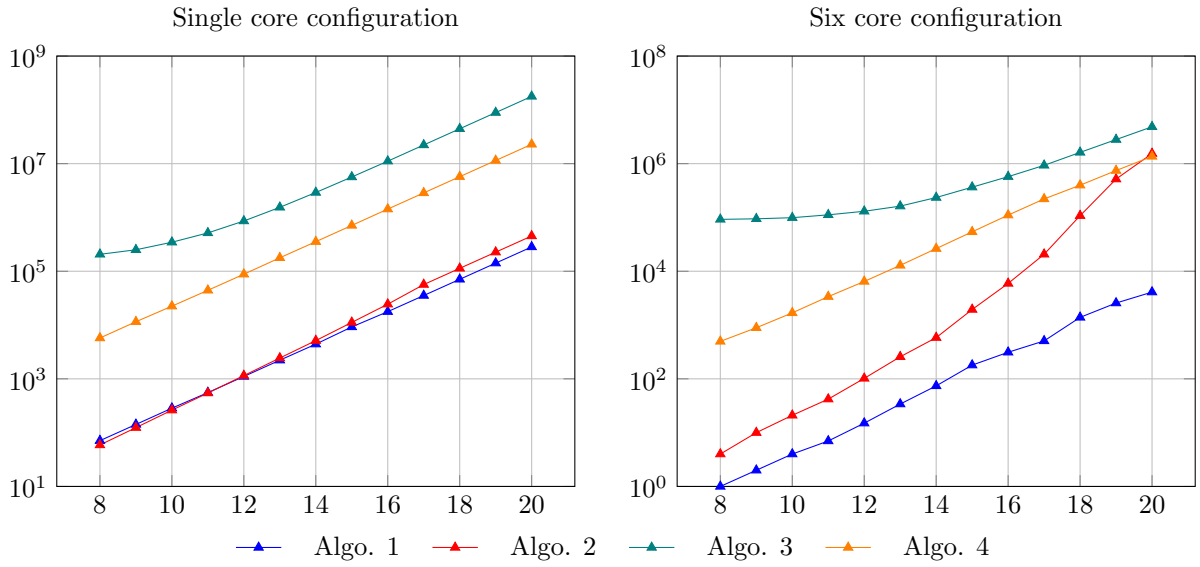
We implement the sum-check prover with all algorithms in our paper to compare the run times of the four algorithms[20]. Our implementation can be instantiated with any finite field and its extensions of any degree. However, we use arkworks [ac22] as the back-end for finite-field arithmetic and field-extension arithmetic. Arkworks is optimized to support finite-field arithmetic for fields with characteristic more than or equal to 64. In other words, field elements are represented as 64-bit integers. Therefore, although it is possible to define a field with characteristic 32 (or less) in Arkworks, each field element is cast as a 64-bit integer which leads to

[20]Our code is available at: `https://github.com/ingonyama-zk/smallfield-super-sumcheck`.

inefficiencies in field multiplications. Further, arkworks does not support field extensions of degree more than four. Considering these practical constraints, we benchmark our implementation with the Babybear field and its degree-4 extension. Our implementation also supports switching back to the Algorithm 1 after using Algorithm 3 or 4 for the first j rounds.

We plot the running times of the sum-check prover with the instance-size ℓ for all algorithms as shown in Figure 5. We use Algorithms 3 and 4 for the first $j = 6$ rounds and then switch back to Algorithm 1. We benchmark the prover run times for degree $d = 4$ on an Intel i7 processor with 6 cores running at 2.6 GHz. We analyze the running times of different algorithms on single-core and six-core configurations to understand performance differences and potential multi-threading improvements. First, we measure single-core running times to establish a baseline, showing each algorithm's inherent computational complexity without parallel processing. Then, we measure six-core running times. These are naturally lower due to parallel execution, achieved solely through compiler-level optimizations without manual adjustments.

Figure 5: Running times (in ms) of all four algorithms w.r.t ℓ for degree $d = 4$ and $j = 6$



In both configurations, the running times of all algorithms increase linearly with ℓ, except for Algorithm 2 in the six-core setup. Algorithm 4 outperforms Algorithm 3, being 5 to 10 times faster due to reduced bb multiplications. However, the existing Algorithms 1 and 2 surprisingly perform better than the optimized Algorithms 3 and 4. This can be attributed to two main factors:

- Parallelizability: Algorithms 3 and 4 are designed for parallel execution, but our implementation lacks manual parallel processing optimizations. About 90% of the runtime for these algorithms is spent on computing the base-field products for all rounds, which can be heavily parallelized. Conversely, Algorithms 1 and 2 benefit from Rust compiler optimizations for multi-core usage. For example, with $\ell = 16$, Algorithm 1's single-core runtime is approximately 60 times slower than its six-core runtime, whereas Algorithm 4's is only 13 times slower. This indicates a narrowing performance gap with multi-threading, suggesting significant potential for further improvements.
- Field choice: We benchmarked using the degree-4 extension of the Babybear field due to the arkworks backend constraints. Algorithms 3 and 4 are expected to perform better when extension-field multiplications are much costlier than base-field multiplications. For Babybear in Arkworks, an ee multiplication is 10 times slower than a bb multiplication. In binary fields, this disparity is much greater. For example, with GF[2] as the base field and a degree-128 extension, an ee multiplication can be $\approx 10^4$ times slower than a bb multiplication [tea23].

Analytical Runtime Analysis. Our implementation does not yet support binary fields. To compare the performance of Algorithms 3 and 4 against the existing algorithms for binary fields, we analytically estimate the running times of all four algorithms. Specifically, we record the unit running times of each of the bb, be and ee multiplications, i.e., the time required to run a single multiplication operation. Next, we calculate the exact number of bb, be and ee multiplications required in each algorithm for different configurations of the instance size ℓ, degree d and switchover round j. To compute the analytical runtimes, we simply multiply the number of multiplications with the unit time for that type of multiplication and sum over the timings for all three multiplication types: bb, be and ee.

We plot the analytically estimated running times of all four algorithms in Figure 6. On the left side, we plot the running times with respect to the instance size ℓ, for a constant degree $d = 4$ and switchover happens at round $j = 6$. Algorithm 4 performs better than all other algorithms by at least an order of magnitude, owing to significantly less number of bb and ee multiplications. Algorithm 3, on the other hand, performs worse than the existing algorithms. This is due to exhorbitantly high number of bb multiplications (in the range of billions) as against relatively fewer (in thousands) ee multiplications in the existing algorithms.

On the right side, we plot the factor improvement[21] of Algorithms 3 and 4 over Algorithm 1 with respect to the switchover round j for a fixed instance size $\ell = 20$ and degree $d \in \{3, 4, 5\}$. The solid lines represent the $d = 3$ case, the dashed lines represent $d = 4$ case and the dotted lines represent $d = 5$ case. The key insights from this plot are as follows.

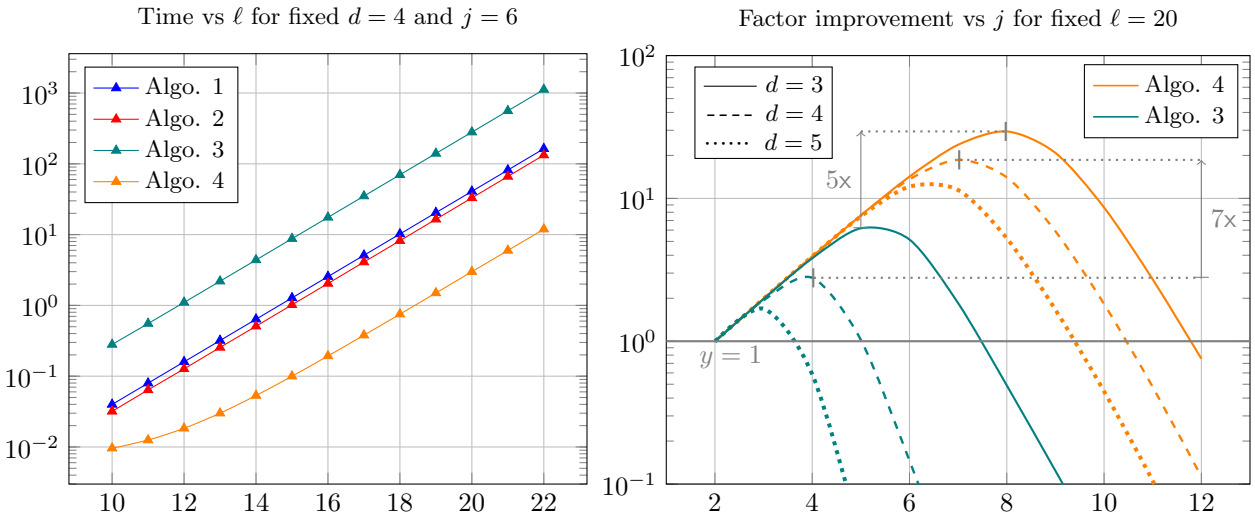
1. The runtimes of Algorithms 3 and 4 are sensitive to the switchover round j and perform better than Algorithm 1 only upto a specific switchover round j. The factor improvement plots exhibit a parabolic trend with respect to the switchover round j. For instance with degree $d = 3$, Algorithm 4 has a factor improvement greater than one only when $j \leq 11$. Analogously, this range reduces to $j \leq 7$ for Algorithm 3 for the same degree $d = 3$.
2. Algorithm 4 outperforms Algorithm 3 in terms of the factor improvement over Algorithm 1. For degree $d = 3$, the peak factor improvement (at $j = 8$) of Algorithm 4 is almost 5 times better than the peak factor improvement of Algorithm 3 (at $j = 5$). For $d = 4$, this ratio increases to almost 7 times. This can be attributed to the fact that the number of bb multiplications in Algorithm 3 grow exponentially with degree d while it grows "subexponentially" in Algorithm 4.
3. For a given degree d, the peak factor improvement for Algorithm 4 occurs at a later switchover round j than that of Algorithm 3. For example with $d = 4$, ideal switchover for Algorithm 4 is $j = 7$ while that for Algorithm 3 is $j = 4$. Thus, Algorithm 4 allows not just a better performance but also more rounds to be processed before we switch to Algorithm 1.
4. Lastly, we notice that as degree d inceases, the factor improvement plots shrink and the permissible range of switchover round j reduces. Algorithms 3 and 4 start giving diminishing returns when the increased number of bb multiplications they require far outweight the reduced ee multiplications.

Acknowledgments. We are 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/`.)

[21] The factor improvement of Algorithm 3 (or 4) over Algorithm 1 is the ratio of analytically estimated runtime of Algorithm 1 and Algorithm 3 (or 4). Higher the factor improvement, better is the performance of Algorithm 3 (or 4) relative to Algorithm 1.

Figure 6: Analytically estimated running times (in seconds) of all four algorithms for degree-128 extension of GF[2] in different configurations.



References

[ac22] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[AFK22] Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-shamir transformation of multi-round interactive proofs. In *Theory of Cryptography Conference*, pages 113–142. Springer, 2022.

[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 *ICALP*, 2018.

[BG23] Jeremy Bruestle and Paul Gafni. Risc zero zkvm: scalable, transparent arguments of riscv integrity, 2023. `https://dev.risczero.com/proof-system-in-detail.pdf`.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *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`.

[DP24] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Paper 2024/504, 2024. `https://eprint.iacr.org/2024/504`.

[DT24] Quang Dao and Justin Thaler. Constraint-packing and the sum-check protocol over binary tower fields. 2024.

[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.

[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.

[Gru24] Angus Gruen. Some improvements for the piop for zerocheck. *Cryptology ePrint Archive*, 2024.

[LFKN90] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *FOCS*, October 1990.

[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`.

[tea23] The Irreducible team. Binius: Rust implementation of Snark over towers of binary fields, 2023. `https://gitlab.com/IrreducibleOSS/binius`.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, 2013.

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

[VSBW13] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

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