Constraint-Packing and the Sum-Check Protocol over Binary Tower Fields

Quang Dao* Justin Thaler†

Abstract

SNARKs based on the sum-check protocol often invoke the "zero-check PIOP". This reduces the vanishing of many constraints to a single sum-check instance applied to an n-variate polynomial of the form $g(x) = \mathsf{eq}(r, x) \cdot p(x)$, where p is a product of multilinear polynomials, r is a random vector, and eq is the multilinear extension of the equality function. In recent SNARK designs, $p(x)$ is defined over a "small" base field, while r is drawn from a large extension field $\mathbb{F}$ for security.

Recent papers (Bagad, Domb, and Thaler 2024; Gruen 2024) have optimized the sum-check protocol prover for this setting. However, these works still require the prover to "pre-compute" all evaluations of $\mathsf{eq}(r, x)$ as x ranges over $\{0,1\}^n$, and this computation involves about n multiplications over the extension field $\mathbb{F}$.

In this note, we describe a modification to the zero-check PIOP in the case of binary tower fields that reduces this "pre-computation" cost by a factor of close to $\log |\mathbb{F}|$, which is 128 in important applications. We show that our modification is sound, and that it strictly generalizes a (possibly folklore) technique of constraint-packing over field extensions.

1 Introduction

The sum-check protocol computes

$$\sum_{x \in \{0,1\}^{\log n}} g(x) \tag{1}$$

for some low-degree n-variate polynomial g. Here, g is an n-variate polynomial defined over some finite field $\mathbb{F}$, and the sum is also defined over $\mathbb{F}$.

From the verifier's perspective, the sum-check protocol acts as a reduction from the task of summing up g's evaluations over the 2^n inputs in $\{0,1\}^n$ to the (hopefully easier) task of evaluating g at a *single* point in $\mathbb{F}^n$.

A prior work of Bagad, Domb, and Thaler [BDT24] considered how to optimize the sum-check prover when

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

where each $p_i(x)$ is a multilinear polynomial such that $p_i(x) \in \mathbb{B}$, where $\mathbb{F}$ is an *extension field* of $\mathbb{B}$. Throughout this note, we refer to $\mathbb{F}$ as the "big field" or "extension field" and $\mathbb{B}$ as the "small field" or "base field".

However, in most applications of the sum-check protocol to SNARK design, g does not *quite* satisfy this property. Often, there is an extra factor in g, defined as follows. Let $\mathsf{eq}(y, x)$ denote the $2n$-variate multilinear polynomial defined as:

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

*Carnegie Mellon University. Work done while at a16z crypto research

†a16z crypto research and Georgetown University

Notice that eq is the multilinear extension of the equality function, meaning that if both x and y are in $\{0,1\}^n$ then

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

When $y \in \{0,1\}^n$ is fixed, we denote $\mathsf{eq}(y,x)$ by $\chi_y(x)$, and call χ_y the y'th multilinear Lagrange basis polynomial. Then:

$$\chi_y(x) = \left(\prod_{j \in \{0,\ldots,n-1\}\colon y_j = 0} (1 - x_j) \right) \cdot \left(\prod_{j \in \{0,\ldots,n-1\}\colon y_j = 1} x_j \right) \tag{3}$$

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:

$$g(x) = \mathsf{eq}(r,x) \cdot \prod_{i=1}^{\ell} p_i(x) \tag{4}$$

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

Roughly speaking, Bagad, Domb, and Thaler [BDT24], as well as Gruen [Gru24] show that when g satisfies Equation (2), then the sum-check protocol prover only needs to perform additions and multiplications over $\mathbb{B}$ for the first few rounds of the protocol. This leads to speedups compared to earlier prover implementations [CTY11, CMT12, Tha13], because in those earlier prover implementations, after round one the prover has to perform many multiplications over the "big field" $\mathbb{F}$.

The results of [BDT24] do apply when g has the form of Equation (4), but the resulting prover is not as fast as one might hope. This is because the prover has to perform many big-field operations over $\mathbb{F}$ throughout the whole protocol (though fewer than if one directly applied older prover implementations). The main source of extension-field 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 Section 7 and [Tha22, Lemma 3.8] for details).

In this note, we drastically reduce this cost for the prover when $\mathbb{F}$ has characteristic two and is constructed as a tower extension of $\mathbb{B}$ (see Section 2 for details of tower extension fields). This is exactly the setting that arises in applications of the Binius [DP23] and FRI-Binius [DP24] commitment schemes and associated polynomial IOPs.

Specifically, let $\mathbb{F}$ be a degree-2^k extension of $\mathbb{B}$. That is, $\mathbb{F}$ is constructed by first taking a degree-two extension of $\mathbb{B}$, and then a degree-2 extension of that, and so on for k iterations.

Our technique is simple. We show that in key applications of the sum-check protocol, it is not necessary to choose $r_0, \ldots, r_{k-1}$ at random. Rather, they can be chosen deterministically to equal very special elements $z_0, \ldots, z_{k-1}$ of the tower basis for $\mathbb{F}$. For each z_i, multiplication of any element of the extension field $\mathbb{F}$ by the field element z_i is extremely fast (roughly equivalent to the cost of a single *addition* in $\mathbb{F}$). This reduces the number of $\mathbb{F}$-multiplications required to compute the values in A from n to roughly $n/2^k$.

2 Overview of Wiedemann's binary tower field construction

For simplicity, let us focus on $\mathbb{B} = \mathsf{GF}(2)$ and $\mathbb{F} = \mathsf{GF}(2^{128})$.

We can associate the 2^{128} elements of $\mathbb{F}$ with 7-variate multilinear polynomials with coefficients over $\mathbb{B}$. Note that any such multilinear polynomial has $2^7 = 128$ coefficients, and hence the total number of such multilinear

polynomials is 2^{128}, allowing for a natural one-to-one correspondence between elements of $\mathsf{GF}(2^{128})$ and seven-variate multilinear polynomials. Let us write $z_0, \ldots, z_6$ for the seven variables of any such multilinear polynomial.

On an actual computer, we can uniquely represent any element of $\mathsf{GF}(2^{128})$ (viewed as a multilinear polynomial $q(z_0, \ldots, z_6)$) by simply listing its 128 coefficients. But the coefficients in what basis? People would typically use the standard monomial basis. So, the polynomial (i.e., $\mathsf{GF}(2^{128})$ element)

$$\sum_{S \subseteq \{0,\ldots,6\}} c_S \cdot \prod_{i \in S} z_i$$

would be represented via a list of the the c_S values.

Adding two $\mathsf{GF}(2^{128})$ elements represented in this basis simply amounts to computing bitwise XOR. Multiplication is more interesting. Given two 7-variate multilinear polynomials (i.e., $\mathsf{GF}(2^{128})$ field elements) $q_1(z_0, \ldots, z_6)$ and $q_2(z_0, \ldots, z_6)$, their product *as polynomials* is not multilinear, as it has degree up to two in each variable. Let $q_3(z_0, \ldots, z_6)$ denote this product polynomial. According to Wiedemann's tower construction [Wie88] of $\mathsf{GF}(2^{128})$ [Wie88], the multiplication rule for $\mathsf{GF}(2^{128})$ declares q_3 to be equivalent to the multilinear polynomial obtained by replacing, for $i = 0, \ldots, 6$, any instance of z_i^2 with

$$z_i \cdot z_{i-1} + 1, \tag{5}$$

where here we understand that z_{-1} is equal to 1.

So, for example, if $q_3(z_0, \ldots, z_6) = z_0^2$, then this is the same as the multilinear polynomial (a.k.a., $\mathsf{GF}(2^{128})$ field element) $z_0 \cdot z_{-1} + 1 = z_0 + 1$.

As a more involved example, if $q_3(z_0, \ldots, z_6) = z_2^2 \cdot z_1^2$, then via (5) this is equivalent to

$$(z_2 \cdot z_1 + 1) \cdot (z_1 \cdot z_0 + 1) = z_2 z_1^2 z_0 + z_2 z_1 + z_1 z_0 + 1,$$

which, via another application of (5), is in turn equivalent to

$$z_2(z_1 z_0 + 1) z_0 + z_2 z_1 + z_1 z_0 + 1 = z_2 z_1 z_0^2 + z_2 z_0 + z_2 z_1 + z_1 z_0 + 1.$$

Via yet another application of (5), this is, at last, equivalent to

$$z_2 z_1 (z_0 + 1) + z_2 z_0 + z_2 z_1 + z_1 z_0 + 1.$$

Since the field has characteristic two, this simplifies to

$$z_2 z_1 z_0 + z_2 z_0 + z_1 z_0 + 1.$$

This is a multilinear polynomial in the standard monomial basis and hence represents a field element in $\mathsf{GF}(2^{128})$.

We refer to $z_0, \ldots, z_6$ as the seven "special" field elements of $\mathsf{GF}(2^{128})$. As shown by Fan and Paar [FP97], multiplying *any* field element in $\mathsf{GF}(2^{128})$ by one of the seven special field elements can be done extremely quickly. In Section 8, we work out this algorithm in detail, including "unrolling" Fan and Paar's recursive multiplication procedure into an iterative form that may be implemented more efficiently in hardware. Ultimately, multiplying any $\mathsf{GF}(2^{128})$ element by a special field element z_i requires computing a bitwise-XOR over at most 128 bits (the same cost as a $\mathsf{GF}(2^{128})$ *addition*).

Therefore, in this work, for each $i \in \{0, \ldots, 6\}$, we consider multiplication by the special $\mathsf{GF}(2^{128})$-element z_i or $(1 - z_i)$ to be "free".

3 Constraint Checking

Say you want to confirm that $g(x) = 0$ for all $x \in \{0,1\}^n$, where g is defined over $\mathbb{B} = \mathsf{GF}(2)$. Letting $\mathbb{F} = \mathsf{GF}(2^{128})$ the standard way to do this is the standard zero-check PIOP [BFL91, BTVW14, CFQ19, Set20]: $\mathcal{V}$ picks a random r in $\mathbb{F}^n$, and applies sum-check to confirm

$$0 = \sum_{x \in \{0,1\}^n} \mathsf{eq}(r, x) \cdot g(x). \tag{6}$$

It's not hard to show that if $g(x) \neq 0$ for any $x \in \{0,1\}^{\log n}$, then with probability at least $1 - \log(n)/|\mathbb{F}|$, Equation (6) will fail to hold. In other words, if the verifier is convinced via sum-check that Equation (6) holds, then it is safe for the verifier to conclude that $g(x) = 0$ for all $x \in \{0,1\}^n$.

Instead, let's exploit the following derivation.

Let h be the unique multilinear polynomial that agrees with g at all $x \in \{0,1\}^n$. Then clearly

$$g(x) = 0 \text{ for all } x \in \{0,1\}^n \iff h(x) = 0 \text{ for all } x \in \{0,1\}^n. \tag{7}$$

It is easy to see the following polynomial identity:

$$h(w) = \sum_{x \in \{0,1\}^n} \mathsf{eq}(w, x) \cdot g(x). \tag{8}$$

Indeed, the right-hand side is multilinear in w and agrees with $g(w)$ whenever w is in $\{0,1\}^n$. Hence, by Schwartz–Zippel, the right-hand side must be the same polynomial as g.

Claim 1. *Let $z_0, \ldots, z_6$ denote the seven "special" elements of $\mathsf{GF}(2^{128})$. Equation (7) holds if and only if*

$$h(z_0, \ldots, z_6, y) = 0 \text{ for all } y \in \{0,1\}^{n-7}. \tag{9}$$

Proof. If Equation (7) holds, then since h is multilinear, it follows that h is identically zero, and hence Equation (9) holds.

Conversely, assume Equation (9) holds. Fix some $y \in \{0,1\}^{n-7}$. Then, as we explain in Equation (11) below, since h is multilinear, $h(z_0, \ldots, z_6, y)$ is a linear combination of 128 evaluations of h, namely all those of the form $h(y', y)$ with $y' \in \{0,1\}^7$. Specifically, for *any* $w \in \mathbb{F}^7$,

$$h(w, y) = \sum_{y' \in \{0,1\}^7} \chi_{y'}(w) \cdot h(y', y), \tag{10}$$

where recall that $\chi_{y'}$ denotes the multilinear Lagrange basis polynomial corresponding to y' (see Equation (3)). Indeed, the right-hand side of Equation (10) is multilinear in w and y and agrees with h whenever w and y are both in $\{0,1\}^n$. Hence the right-hand side must *equal* h.

It follows from Equation (10) that

$$h(z_0, \ldots, z_6, y) = \sum_{y' \in \{0,1\}^7} \chi_{y'}(z_0, \ldots, z_6) \cdot h(y', y). \tag{11}$$

Now, since h is defined over $\mathbb{B}$, $h(y', y) \in \mathbb{B}$ for each $y', y \in \{0,1\}^7 \times \{0,1\}^{n-7}$. We now look at the right-hand side of Equation (11) as a multilinear polynomial in the variables $z_0, \ldots, z_6$. Since

$$\{\chi_{y'}(z_0, \ldots, z_6) \colon y' \in \{0,1\}^7\}$$

form a multilinear basis over $\mathbb{B}$, and because the left-hand side is zero, it follows that the coefficients of $\chi_{y'}(z_0, \ldots, z_6)$ in the right-hand side of Equation (11), which are $h(y', y)$ for all $y \in \{0,1\}^7$, must all be zero. This finishes the claim. □

By Claim 1 and the discussion proceeding it, checking that $g(x) = 0$ for all $x \in \{0,1\}^n$ is equivalent to checking that $h(z_0, \ldots, z_6, y) = 0$ for all $y \in \{0,1\}^{n-7}$. By Schwartz–Zippel, up to soundness error $(n-7)/|\mathbb{F}|$, it is enough for the verifier to choose a random value $r \in \mathbb{F}^{n-7}$ and confirm that $h(z_0, \ldots, z_6, r) = 0$. By Equation (8), this is equivalent to confirming that

$$0 = \sum_{x \in \{0,1\}^n} \mathsf{eq}((z_0, \ldots, z_6, r), x) \cdot g(x). \tag{12}$$

This is exactly the standard "zero-check PIOP" except that the first 7 entries of $(z_0, \ldots, z_6, r)$ are deterministically fixed to the special field elements $z_0, \ldots, z_6$, rather than chosen at random from $\mathsf{GF}(2^{128})$. Since multiplying any $\mathsf{GF}(2^{128})$-element by z_i or $(1 - z_i)$ can be done extremely quickly [FP97],[1] it is easy to see that the standard memoization procedure [VSBW13, Tha13] for computing $\{\mathsf{eq}((z_0, \ldots, z_6, r), x) \colon x \in \{0,1\}^n\}$ can be done with about 2^{n-7} $\mathsf{GF}(2^{128})$-multiplications, in contrast to the standard zero-check PIOP that requires 2^n such multiplications. This is a speedup factor of about $2^7 = 128$. Note that in order to get this speedup factor, we need to process $z_0, \ldots, z_6$ *last*, not first, in the memoization procedure. We discuss this detail in Section 7.

4 An optimization when g has degree one in some variables

Suppose that g has degree one in its first ℓ variables. Then Equation (8) can be simplified to

$$h(w_0, \ldots, w_{\ell-1}, w') = \sum_{x \in \{0,1\}^{n-\ell}} \mathsf{eq}(w', x) \cdot g(w_0, \ldots, w_{\ell-1}, x). \tag{13}$$

Indeed, the right-hand side Equation (13) is multilinear in $(w_0, \ldots, w_{\ell-1}, w')$ and agrees with h whenever $(w_0, \ldots, w_{\ell-1}, w') \in \{0,1\}^n$, so the right-hand side must equal h as a formal polynomial. Note that the sum on the right-hand side is only over $2^{n-\ell}$ terms rather than 2^n terms. This is a factor of 2^ℓ fewer terms being summed than in Equations (8) and (12).

Accordingly, to check that $h(z_0, \ldots, z_{\ell-1}, r) = 0$ for any $r \in \mathbb{F}^{n-\ell}$, it suffices to apply the sum-check protocol to compute

$$\sum_{x \in \{0,1\}^{n-\ell}} \mathsf{eq}(r, x) \cdot g(z_0, \ldots, z_{\ell-1}, x),$$

rather than to

$$\sum_{x \in \{0,1\}^n} \mathsf{eq}((z_0, \ldots, z_\ell, r)), x) \cdot g(z_0, \ldots, z_{\ell-1}, x),$$

as per the unoptimized protocol captured in Equation (12).

5 Showing this protocol generalizes "simple constraint packing"

Simple constraint packing. Suppose that the verifier has evaluation access to six n-variate polynomials p_1, q_1, s_1, p_2, q_2, and s_2, all defined over $\mathbb{B}$, and the prover claims that for all $x \in \{0,1\}^n$, it holds that

$$p_1(x) \cdot q_1(x) - s_1(x) = 0 \tag{14}$$

and

$$p_2(x) \cdot q_2(x) - s_2(x) = 0. \tag{15}$$

These are $2 \cdot 2^n$ rank-one constraints in total, with the first 2^n due to Equation (14) and the second 2^n constraints due to Equation (15). We can reduce the number of constraints by half, to 2^n, by replacing the

[1] As discussed earlier, we consider these multiplications to be "free".

separate requirements that $p_1(x) \cdot q_1(x) - s_1(x) = 0$ and $p_2(x) \cdot q_2(x) - s_2(x) = 0$ with the single constraint (over the extension $\mathbb{B}(z_0) \subset \mathbb{F}$) that

$$z_0 \cdot (p_1(x) \cdot q_1(x) - s_1(x)) + (1 - z_0) \cdot (p_2(x) \cdot q_2(x) - s_2(x)) = 0. \tag{16}$$

Indeed, because z_0 and $(1 - z_0)$ are linearly independent over $\mathbb{B}$, Equations (14) and (15) hold if and only if Equation (16) holds.

Applying the standard sum-check-based zero-check PIOP to Equation (16) amounts to applying sum-check to confirm that

$$0 = \sum_{x \in \{0,1\}^n} \mathsf{eq}(r, x) \cdot (z_0 \cdot (p_1(x) \cdot q_1(x) - s_1(x)) + (1 - z_0) \cdot (p_2(x) \cdot q_2(x) - s_2(x))) \,. \tag{17}$$

We call the above application of the sum-check protocol, in order to check that Equations (14) and (15) both hold, the *simple constraint-packing approach.*

Relating simple constraint packing to our protocol. As we now explain, the simple constraint-packing approach is *identical* to what one obtains from our constraint-checking protocol of Section 3 when using the optimization of Section 4 with $\ell = 1$. Define an $(n+1)$-variate polynomial g as follows

$$g(x_0, x) = x_0 \cdot (p_1(x)q_1(x) - s_1(x)) + (1 - x_0) \cdot (p_2(x)q_2(x) - s_2(x)) \tag{18}$$

Then Equations (14) and (15) both hold if and only if

$$g(x_0, x) = 0 \text{ for all } (x_0, x) \in \{0,1\}^{1+n} \tag{19}$$

Notice that g has degree one in x_0. Applying the optimization of Section 4 (with $\ell = 1$) to our constraint-checking procedure from Section 3 leads to the following protocol. The verifier picks a random $r \in \mathbb{F}^n$, and invokes the sum-check protocol to confirm that

$$0 = \sum_{x \in \{0,1\}^n} \mathsf{eq}(r, x) \cdot g(z_0, x).$$

By the definition of $g(x)$ (see Equation (18)) and the optimization of Section 4, this application of the sum-check protocol is *equivalent* to the simple constraint-packing approach captured via Equation (17). Hence, our constraint-checking procedure from Section 3, optimized appropriately per Section 4, is a strict generalization of simple constraint-packing.

6 Some intuition for what's going on

Sum-check works with multilinear polynomials. Tower field elements *are* multilinear polynomials. Typically, extension field elements are regarded as *univariate* polynomials, modulo an irreducible polynomial over the base field. But $\mathsf{GF}(2^{128})$ elements under the tower construction are naturally viewed as (seven-variate) *multilinear* polynomials, which are exactly the polynomials that arise in key applications of the sum-check protocol. Hence, $\mathsf{GF}(2^{128})$ elements slot right into any multilinear polynomials being sum-checked, with the first seven variables of the polynomial whose evaluations are being summed regarded as the seven variables specifying a $\mathsf{GF}(2^{128})$ element.

For example, suppose the sum-check protocol is applied to an n-variate polynomial g that has degree 1 in its first 7 variables. Then fixing those seven variables to the "special" $\mathsf{GF}(2^{128})$-elements $z_0, \ldots, z_6$, one can treat any sum over all 128 possible products of those first seven variables, each multiplied by a base-field element, as specifying a single $\mathsf{GF}(2^{128})$ element. This lets one cut out the first seven rounds of the standard sum-check-based zero-check PIOP applied to g. In other words, g gets replaced with the polynomial $q(x) = g(z_0, \ldots, z_6, x)$ over seven fewer variables than g itself. Whereas g's evaluations are in the base field $\mathsf{GF}(2)$, q's are in the extension field $\mathbb{F} = \mathsf{GF}(2^{128})$.

If g is *not* of degree 1 in its first seven variables, we can still do something similar, because the sum-check-based zero-check PIOP considers the *multilinear* polynomial h that agrees with g over the Boolean hypercube, so everything in the previous paragraph can be said about h rather than g itself.

Performance, soundness, and tensor structure in the tower basis. The key observation of this note is that there are seven special field elements $z_0, \ldots, z_6$ such that one can deterministically fix the first seven variables of r to be $z_0, \ldots, z_6$ before having the prover compute $\mathsf{eq}(r, x)$ as x ranges over $\{0,1\}^n$. The soundness of this procedure relies on the fact that there are bases for $\mathsf{GF}(2^{128})$ as a vector space over the base field $\mathsf{GF}(2)$ that consist of tensor products of these seven special field elements $z_0, \ldots, z_6$. These bases are called *tower bases*. This "tensor product structure" in bases for $\mathsf{GF}(2^{128})$ seems unique to tower field constructions.

More precisely, for soundness in this note, it's essential that the following 128 elements of $\mathsf{GF}(2^{128})$ are linearly independent over $\mathsf{GF}(2)$ (and hence form a basis):

$$\begin{aligned}(z_0-1)(z_1-1)\ldots(z_6-1),\\ z_0(z_1-1)\ldots(z_6-1),\\ \vdots\\ z_0z_1\ldots z_6.\end{aligned}$$

The tensor product structure in tower bases is also implicitly exploited in the fact that multiplication by $z_0, \ldots, z_6$ is extremely fast. Indeed, the fast algorithm to multiply by z_i heavily exploits the fact that $\mathsf{GF}(2^{2^i})$ is a degree-two extension of $\mathsf{GF}(2^{2^{i-1}})$ for $i = 1, \ldots, 7$ (see Section 8 for the full description of the algorithm to multiply by z_i). This is exactly how tower fields are constructed, via a sequence of degree-two extensions.

7 Memoization for evaluating the eq polynomial at many points

In this section, we describe in detail how to fully utilize the benefits of constraint packing (see Section 3). The optimization is applicable to the prover's "pre-computation" phase, before the first round of the sum-check protocol applied to g (see Equation (4)), where the prover wishes to compute

$$\mathsf{eq}(w, y) = \prod_{i=0}^{n-1} (w_i \cdot y_i + (1-w_i)(1-y_i)) \qquad \text{for all} \quad y \in \{0,1\}^n,$$

and store these evaluations. Following this pre-computation, the sum-check prover can be implemented in the manner described in [BDT24].

In our setting, we have $w = (z_0, \ldots, z_6, r)$, where $r \leftarrow \mathbb{F}^{n-7}$ is a vector of random field elements, and $z_0, \ldots, z_6$ are the special field elements for the tower field $\mathbb{B} \subset \mathbb{F}$.

First, we recall a simple algorithm for general $w \in \mathbb{F}^n$ which costs roughly 2^n field multiplications in $\mathbb{F}$:

Algorithm 1 Computation of $\mathsf{eq}(w, y)$ for all $y \in \{0,1\}^n$. In the comment on Line 7, $\mathsf{bin}(k)$ denotes the n-bit binary representation of integer $k \in \{0, \ldots, 2^n - 1\}$.

1: Initialize an all-one vector $\mathbf{v} = (1, 1, \ldots, 1) \in \{0,1\}^n$
2: **for** $i = 0, \ldots, n-1$ **do**
3: **for** $j = 2^i - 1, \ldots, 0$ **do**
4: Set $\mathbf{v}[2j+1] := \mathbf{v}[j] \cdot w_i$ and $\mathbf{v}[2j] := \mathbf{v}[j] - \mathbf{v}[2j+1]$
5: **end for**
6: **end for**
7: **return v** ▷ we have $\mathbf{v}[k] = \mathsf{eq}(w, \mathsf{bin}(k))$ for all $k = 0, \ldots, 2^n - 1$.

For example, here is the algorithm in action for $w = (w_0, w_1, w_2)$, where we denote $\overline{w}_i := 1 - w_i$:

Index	7	6	5	4	3	2	1	0
Init:	1	1	1	1	1	1	1	1
Round 1:	$\overline{w}_0$	w_0	1	1	1	1	1	1
Round 2:	$\overline{w}_1\overline{w}_0$	$\overline{w}_1 w_0$	$w_1\overline{w}_0$	$w_1 w_0$	1	1	1	1
Round 3:	$\overline{w}_2\overline{w}_1\overline{w}_0$	$\overline{w}_2\overline{w}_1 w_0$	$\overline{w}_2 w_1\overline{w}_0$	$\overline{w}_2 w_1 w_0$	$w_2\overline{w}_1\overline{w}_0$	$w_2\overline{w}_1 w_0$	$w_2 w_1\overline{w}_0$	$w_2 w_1 w_0$

From this example, it is clear that the number of multiplications is not the same for each entry of w. Indeed, we do 2 multiplications (in $\mathbb{F}$) with w_0, 4 multiplications with w_1, and so on, up to 2^{n-1} multiplications with w_{n-1}. When $w = (z_0, \ldots, z_6, r)$ as in our setting, we want to make sure that most of the multiplications involve a "special" field element z_i, since such multiplications are cheap. Hence, we should evaluate w in the *opposite* order, going from $r_{n-8}, \ldots, r_0$ to $z_6, \ldots, z_0$. In other words, we should use the following algorithm:

Algorithm 2 Computation of $\mathsf{eq}(w, y)$ for all $y \in \{0,1\}^n$, where $w = (z_0, \ldots, z_6, r)$

1: Initialize an all-one vector $\mathbf{v} = (1, 1, \ldots, 1) \in \{0,1\}^n$
2: **for** $i = n-1, \ldots, 0$ **do**
3: **for** $j = 2^{n-1-i} - 1, \ldots, 0$ **do**
4: Set $\mathbf{v}[2^{i+1}j + 2^i] := \mathbf{v}[2^{i+1}j] \cdot w_i$ and $\mathbf{v}[2^{i+1}j] := \mathbf{v}[2^{i+1}j] - \mathbf{v}[2^{i+1}j + 2^i]$
5: **end for**
6: **end for**
7: **return v** ▷ we have $\mathbf{v}[k] = \mathsf{eq}(w, \mathsf{bin}(k))$ for all $k = 0, \ldots, 2^n - 1$

Once again, here's the modified algorithm in action for $w = (z_0, z_1, r)$:

Index	7	6	5	4	3	2	1	0
Init:	1	1	1	1	1	1	1	1
Round 1:	$\overline{r}$	1	1	1	r	1	1	1
Round 2:	$\overline{r}\,\overline{z}_1$	1	$\overline{r}z_1$	1	$r\overline{z}_1$	1	rz_1	1
Round 3:	$\overline{r}\,\overline{z}_1\overline{z}_0$	$\overline{r}\,\overline{z}_1 z_0$	$\overline{r}\,z_1\overline{z}_0$	$\overline{r}z_1 z_0$	$r\overline{z}_1\overline{z}_0$	$r\overline{z}_1 z_0$	$rz_1\overline{z}_0$	$rz_1 z_0$

This will result in roughly 2^{n-7} total multiplications in $\mathbb{F}$, and 2^{n-k-1} multiplications with z_k for all $k = 6, \ldots, 0$. Since the latter computations with $z_6, \ldots, z_0$ are basically free, this way of evaluating $\{\mathsf{eq}(w, y) : y \in \{0,1\}^n\}$ saves a factor of close to $2^7 = 128$ in the time required to compute all of these evaluations.

8 Multiplication by special elements in binary tower fields

In this section, we give an explicit description for the multiplication of any field element $a \in \mathsf{GF}(2^{2^k})$, defined by the tower field construction (Section 2), by each of the special elements $z_0, \ldots, z_{k-1}$. We first present the recursive algorithm as in [FP97], and then "unroll" this algorithm to give explicit formulas for updating each of the 2^k bits of a's representation in the tower basis.

Recall that the tower construction identifies any $a \in \mathsf{GF}(2^{2^k})$ with a k-variate multilinear polynomial. Hence, we can index the bits of a by subsets $S \subseteq [\![k-1]\!] := \{0, 1, \ldots, k-1\}$. In other words, we have

$$a = \sum_{S \subseteq [\![k-1]\!]} a[S] \cdot z_S, \qquad \text{where} \qquad z_S := \prod_{i \in S} z_i.$$

For any $i \in [\![k-1]\!]$, we define

$$a_i := \sum_{T \subseteq [\![k-1]\!] \setminus \{i\}} a[T \cup \{i\}] \cdot z_T, \qquad \text{and} \qquad a_{\overline{i}} := \sum_{T \subseteq [\![k-1]\!] \setminus \{i\}} a[T] \cdot z_T.$$

This partitions the bits of a depending on whether the corresponding index S contains i. We have that

$$a = a_i \cdot z_i + a_{\bar{i}},$$

and that a_i, and $a_{\bar{i}}$ both consist of 2^{k-1} bits.

The recursive algorithm for multiplying a by z_i [FP97] is as follows:

1. For $i = 0$, we have

$$\begin{aligned} a \cdot z_0 = (a_0 \cdot z_0 + a_{\overline{0}}) \cdot z_0 = a_0 \cdot z_0^2 + a_{\overline{0}} \cdot z_0 = a_0 \cdot (z_0 + 1) + a_{\overline{0}} \cdot z_0 \\ = (a_0 + a_{\overline{0}}) \cdot z_0 + a_0. \end{aligned}$$

   In other words, we simply compute $a := (a_0, a_{\overline{0}}) \mapsto (a_0 + a_{\overline{0}}, a_0)$, which costs one addition in the field $\mathsf{GF}(2^{2^{k-1}})$, i.e., computing the bitwise XOR of two vectors of length 2^{k-1}.

2. For $i > 0$, we have

$$\begin{aligned} a \cdot z_i = (a_i \cdot z_i + a_{\bar{i}}) \cdot z_i = a_i \cdot z_i^2 + a_{\bar{i}} \cdot z_i = a_i \cdot (z_i \cdot z_{i-1} + 1) + a_{\bar{i}} \cdot z_i \\ = (a_i \cdot z_{i-1} + a_{\bar{i}}) \cdot z_i + a_i. \end{aligned}$$

   In other words, multiplication by z_i can be expressed as the map

$$a = (a_i, a_{\bar{i}}) \mapsto (a_i \cdot z_{i-1} + a_{\bar{i}}, a_i). \tag{20}$$

   We now recursively compute $a_i \cdot z_{i-1}$ and substitute the result into Equation (20) to obtain $a \cdot z_i$. Importantly, since a_i can be seen as a multilinear polynomial in $z_0, \ldots, z_{i-1}, z_{i+1}, \ldots, z_{k-1}$, the recursive multiplication $a_i \cdot z_{i-1}$ can be seen as an operation in $\mathsf{GF}(2^{2^{k-1}})$ (up to "shifting" $z_{i+1}, \ldots, z_{k-1}$ to $z_i, \ldots, z_{k-2}$), and hence only half as expensive as $a \cdot z_i$.

We may calculate the total cost of the above procedure as follows. Let $N_{k,i}$ be the total lengths of the vectors that need to be bitwise XORed together to compute $a \cdot z_i$ with the method above. Then

$$N_{k,0} = 2^{k-1} \quad \text{and} \quad N_{k,i} = N_{k-1,i-1} + 2^{k-1}.$$

A simple inductive analysis then shows

$$N_{k,i} = 2^{k-1} + 2^{k-2} + \cdots + 2^{k-i-1} = 2^k - 2^{k-i-1} \quad \text{for all } 0 \le i < k.$$

Interestingly, this means that multiplying $a \in \mathsf{GF}(2^{2^k})$ by z_i requires slightly *fewer* bit additions than the 2^k bit additions that are required to add an arbitrary $\mathsf{GF}(2^{2^k})$ element to a, though multiplication by z_i also rearranges the bits of a.

We now convert this recursive algorithm to an iterative one. Specifically, we want to derive a formula for how each bit $b[S]$ of $b := a \cdot z_i$ can be computed from the bits of a. Intuitively, Equation (20) states that multiplication of $a = (a_i, a_{\bar{i}})$ causes a_i and $a_{\bar{i}}$ to "switch places," but we also add (i.e., bitwise-XOR) $a_i \cdot z_{i-1}$ to $a_{\bar{i}}$. This has the following consequences:

- Since Equation (20) states that multiplication by z_i causes a_i and $a_{\bar{i}}$ to "switch places," we need to add $a[S\Delta\{i\}]$ to $b[S]$. Here Δ denotes the symmetric difference of sets. Equivalently, the algorithm will initialize $b[S]$ with $a[S\Delta\{i\}]$.
- If $i \notin S$, this completes the calculation of $b[S]$. If $i \in S$, we still need to add $a_i \cdot z_{i-1}$ to $b[S]$. This is done as follows.
- Since $(a_i)_{i-1}$ and $(a_i)_{\overline{i-1}}$ "switch places" in $a_i \cdot z_{i-1}$, we need to add $a[S\Delta\{i-1\}]$ to $b[S]$. If $i - 1 \notin S$, this completes the calculation of $b[S]$. Otherwise, we still need to add $a_{i-1} \cdot z_{i-2}$ to $b[S]$.
- This continues down to whether $0 \in S$. In this case, we make the convention that $S\Delta\{-1\} = S$.

To summarize, we present the iterative algorithm as Algorithm 3 below.

Algorithm 3 Iterative computation of $a \cdot z_i$ for any $a \in \mathsf{GF}(2^{2^k})$ and $i = 0, \ldots, k-1$.
$U \cup V$ corresponds to the set S considered in our prose description of the iterative algorithm.

Initialize $b[S] := a[S\Delta\{i\}]$ for all $S \subseteq [\![k-1]\!]$.
for $U \subseteq \{i+1, \ldots, k-1\}$ **do**
    for $V \subseteq \{0, \ldots, i\}$ **do**
        for $j = i, \ldots, 0$ **do**
            if $j \notin V$ **then**
                Break innermost **for** loop. ▷ We are done computing $b[U \cup V]$
            else
                Update $b[U \cup V] \leftarrow b[U \cup V] + a[U \cup (V\Delta\{j-1\})]$. ▷ Define $V\Delta\{-1\} := V$
            end if
        end for
    end for
end for
return b

9 Packing for lookup arguments and offline memory checking

A standard step in lookup arguments like Lasso [STW23] and other offline memory-checking procedures like Spice [SAGL18] is to take a sequence of n triples of field elements (a_i, b_i, c_i) and replace each triple with a single field element. The mapping from triples of field elements to one field element must be (with overwhelming probability, if the mapping is randomized) injective on the set of n triples. The standard way to do this is fingerprinting: the verifier picks a random $r \in \mathbb{F}$ and (a_i, b_i, c_i) is mapped to $a_i + b_i \cdot r + c_i \cdot r^2$. The probability over the random choice of r that two distinct tuples collide under this mapping is at most $n^2/|\mathbb{F}|$.

If a_i, b_i, and c_i are each in, say, $\mathsf{GF}(2^{32})$, and $\mathbb{F} = \mathsf{GF}(2^{128})$ as would typically be the case in offline memory-checking procedures that use the Binius commitment scheme [DP23, DP24], we can instead deterministically and injectively pack a_i, b_i, c_i into a single field element, via the map $(a_i, b_i, c_i) \mapsto a_i + z_5 \cdot b_i + z_6 \cdot c_i$. This map is "free" for the prover to compute when working in the tower basis. Compared to the standard fingerprinting approach, this saves the prover two field multiplications per tuple.

Acknowledgements

Justin Thaler is grateful to Jim Posen and Ben Diamond for sharing their work on constraint packing, and for valuable feedback and discussion.

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

References

[BDT24] Suyash Bagad, Yuval Domb, and Justin Thaler. The sum-check protocol over fields of small characteristic. 2024.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational complexity*, 1:3–40, 1991.

[BTVW14] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. ePrint Report 2014/846, 2014.

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2075–2092, 2019.

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

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

[DP23] 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*, 2024.

[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.

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

[SAGL18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

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

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

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

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

[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.