HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates

Binyi Chen[1](✉), Benedikt Bünz[1,2], Dan Boneh[2], and Zhenfei Zhang[1]

[1] Espresso Systems, Menlo Park, USA
binyi@espressosys.com
[2] Stanford University, Stanford, USA

Abstract. Plonk is a widely used succinct non-interactive proof system that uses univariate polynomial commitments. Plonk is quite flexible: it supports circuits with low-degree "custom" gates as well as circuits with lookup gates (a lookup gate ensures that its input is contained in a predefined table). For large circuits, the bottleneck in generating a Plonk proof is the need for computing a large FFT.

We present HyperPlonk, an adaptation of Plonk to the boolean hypercube, using multilinear polynomial commitments. HyperPlonk retains the flexibility of Plonk but provides several additional benefits. First, it avoids the need for an FFT during proof generation. Second, and more importantly, it supports custom gates of much higher degree than Plonk without harming the running time of the prover. Both of these can dramatically speed up the prover's running time. Since HyperPlonk relies on multilinear polynomial commitments, we revisit two elegant constructions: one from Orion and one from Virgo. We show how to reduce the Orion opening proof size to less than 10 KB (an almost factor 1000 improvement) and show how to make the Virgo FRI-based opening proof simpler and shorter (This is an extended abstract. The full version is available on EPRINT[22]).

1 Introduction

Proof systems [4,31] have a long and rich history in cryptography and complexity theory. In recent years, the efficiency of proof systems has dramatically improved and this has enabled a multitude of new real-world applications that were not previously possible. In this paper, we focus on succinct non-interactive arguments of knowledge, also called SNARKs [13]. Here, succinct refers to the fact that the proof is short and verification time is fast, as explained below. Recent years have seen tremendous progress in improving the efficiency of the prover [17,27,32, 43,48–50,55].

Let us briefly review what a (preprocessing) SNARK is. We give a precise definition in the full version. Fix a finite field $\mathbb{F}$, and consider the relation $\mathcal{R}(\mathcal{C}, \mathbb{x}, \mathbb{w})$ that is true whenever $\mathbb{x} \in \mathbb{F}^n$, $\mathbb{w} \in \mathbb{F}^m$, and $\mathcal{C}(\mathbb{x}, \mathbb{w}) = 0$, where $\mathcal{C}$ is the description of an arithmetic circuit over $\mathbb{F}$ that takes $n + m$ inputs. A SNARK enables a prover $\mathcal{P}$ to non-interactively and succinctly convince a verifier $\mathcal{V}$ that $\mathcal{P}$ knows

C. Hazay and M. Stam (Eds.): EUROCRYPT 2023, LNCS 14005, pp. 499–530, 2023.
https://doi.org/10.1007/978-3-031-30617-4_17

a witness $\mathbb{w} \in \mathbb{F}^m$ such that $\mathcal{R}(\mathcal{C}, \mathbb{x}, \mathbb{w})$ holds, for some public circuit $\mathcal{C}$ and $\mathbb{x} \in \mathbb{F}^n$.

In more detail, a SNARK is a tuple of four algorithms $(\mathsf{Setup}, \mathcal{I}, \mathcal{P}, \mathcal{V})$, where $\mathsf{Setup}(1^\lambda)$ is a randomized algorithm that outputs parameters gp, and $\mathcal{I}(\mathsf{gp}, \mathcal{C})$ is a deterministic algorithm that pre-processes the circuit $\mathcal{C}$ and outputs prover parameters pp and verifier parameters vp. The prover $\mathcal{P}(\mathsf{pp}, \mathbb{x}, \mathbb{w})$ is a randomized algorithm that outputs a proof π, and the verifier $\mathcal{V}(\mathsf{vp}, \mathbb{x}, \pi)$ is a deterministic algorithm that outputs 0 or 1. The SNARK must be *complete*, *knowledge sound*, and *succinct*. Here *succinct* means that if $\mathcal{C}$ contains s gates, and $\mathbb{x} \in \mathbb{F}^n$, then the size of the proof should be $O_\lambda(\log s)$ and the verifier's running time should be $\tilde{O}_\lambda(n + \log s)$. A SNARK is often set in the random oracle model where all four algorithms can query the oracle. If the Setup algorithm is randomized, then we say that the SNARK requires a *trusted setup*; otherwise, the SNARK is said to be *transparent* because Setup only has access to public randomness via the random oracle. Optionally, we might want the SNARK to be zero-knowledge, in which case it is called a zkSNARK.

Modern SNARKs are constructed by compiling an information-theoretic object called an Interactive Oracle Proof (IOP) [11] to a SNARK using a suitable cryptographic commitment scheme. There are several examples of this paradigm. Some SNARKs use a univariate polynomial commitment scheme to compile a Polynomial-IOP to a SNARK. Examples include Marlin [24], and Plonk [27]. Other SNARKs use a multivariate linear (multilinear) commitment scheme to compile a multilinear-IOP to a SNARK. Examples include Hyrax [48], Libra [49], Spartan [43], Quarks [44], and Gemini [17]. Yet other SNARKs use a vector commitment scheme (such as a Merkle tree) to compile a vector-IOP to a SNARK. The STARK system [8] is the prime example in this category, but other examples include Aurora [10], Virgo [55], Brakedown [32], and Orion [51]. While STARKs are post-quantum secure, require no trusted setup, and have an efficient prover, they generate a relatively long proof (tens of kilobytes in practice). The paradigm of compiling an IOP to a SNARK using a suitable commitment scheme lets us build *universal* SNARKs where a single trusted setup can support many circuits. In earlier SNARKs, such as [14,30,34], every circuit required a new trusted setup.

The Plonk System. Among the IOP-based SNARKs that use a Polynomial-IOP, the Plonk system [27] has emerged as one of the most widely adopted in industry. This is because Plonk proofs are very short (about 400 bytes in practice) and fast to verify. Moreover, Plonk supports custom gates, as we will see in a minute. An extension of Plonk, called PlonKup [41], further extends Plonk to incorporate lookup gates using the Plookup IOP of [27].

One difficulty with Plonk, compared to some other schemes, is the prover's complexity. For a circuit $\mathcal{C}$ with s arithmetic gates, the Plonk prover runs in time $O_\lambda(s \log s)$. The primary bottlenecks come from the fact that the prover must commit to and later open several degree $O(s)$ polynomials. When using the KZG polynomial commitment scheme [36], the prover must (i) compute a multi-exponentiation of size $O(s)$ in a pairing-friendly group where discrete log is hard, and (ii) compute several FFTs and inverse-FFTs of dimension $O(s)$.

When using a FRI-based polynomial commitment scheme [7,37,55], the prover computes an $O(cs)$-sized FFT and $O(cs)$ hashes, where $1/c$ is the rate of a certain Reed-Solomon code. The performance further degrades for circuits that contain *high-degree* custom gates, as some FFTs and multi-exponentiations have size proportional to the degree of the custom gates.

In practice, when the circuit size s is bigger than 2^{20}, the FFTs become a significant part of the running time. This is due to the quasi-linear running time of the FFT algorithm, while other parts of the prover scale linearly in s. The reliance on FFT is a direct result of Plonk's use of *univariate* polynomials. We note that some proof systems eliminate the need for an FFT by moving away from Plonk altogether [17,25,32,43,51].

Hyperplonk. In this paper, we introduce HyperPlonk, an adaptation of the Plonk IOP and its extensions to operate over the boolean hypercube $B_\mu := \{0,1\}^\mu$. We present HyperPlonk as a multilinear-IOP, which means that it can be compiled using a suitable multilinear commitment scheme to obtain a SNARK (or a zkSNARK) with an efficient prover.

HyperPlonk inherits the flexibility of Plonk to support circuits with custom gates, but presents several additional advantages. First, by moving to the boolean hypercube we eliminate the need for an FFT during proof generation. We do so by making use of the classic SumCheck protocol [39], and this reduces the prover's running time from $O_\lambda(s \log s)$ to $O_\lambda(s)$. The efficiency of SumCheck is the reason why many of the existing multilinear SNARKs [17,43,44,48,49] use the boolean hypercube. Here we show that Plonk can similarly benefit from the SumCheck protocol.

Second, and more importantly, we show that the hypercube lets us incorporate custom gates more efficiently into HyperPlonk. A custom gate is a function $G : \mathbb{F}^\ell \to \mathbb{F}$, for some ℓ. An arithmetic circuit $\mathcal{C}$ with a custom gate G, denoted $\mathcal{C}[G]$, is a circuit with addition and multiplication gates along with a custom gate G that can appear many times in the circuit. The circuit may contain multiple types of custom gates, but for now, we will restrict to one type to simplify the presentation. These custom gates can greatly reduce the circuit size needed to compute a function, leading to a faster prover. For example, if one needs to implement the S-box in a block cipher, it can be more efficient to implement it as a custom gate.

Custom gates are not free. Let $G : \mathbb{F}^\ell \to \mathbb{F}$ be a custom gate that computes a multivariate polynomial of total degree d. Let $\mathcal{C}[G]$ be a circuit with a total of s gates. In the Plonk IOP, the circuit $\mathcal{C}[G]$ results in a prover that manipulates univariate polynomials of degree $O(s \cdot d)$. Consequently, when compiling Plonk using KZG [36], the prover needs to do a group multi-exponentiation of size $O(sd)$ as well as FFTs of this dimension. This restricts custom gates in Plonk to gates of low degree.

We show that the prover's work in HyperPlonk is much lower. Let $G : \mathbb{F}^\ell \to \mathbb{F}$ be a custom gate that can be evaluated using k arithmetic operations. In HyperPlonk, the bulk of the prover's work when processing $\mathcal{C}[G]$ is only $O(sk \log^2 k)$ field operations. Moreover, when using KZG multilinear commitments [40], the total number of group exponentiations is only $O(s + d \log s)$, where d is the total degree of G. This is much lower than Plonk's $O(sd)$ group exponentiations. It lets us use custom gates of much higher degree in HyperPlonk.

Making Plonk and its Plonkup extension work over the hypercube raises interesting challenges, as discussed in Sect. 1.1. In particular, adapting the Plookup IOP [27], used to implement table lookups, requires changing the protocol to make it work over the hypercube (see Sect. 3.6). The resulting version of HyperPlonk that supports lookup gates is called HyperPlonk+ and is described in the full version.

Batch Openings and Commit-and-Prove SNARKs. he prover in HyperPlonk needs to open several multilinear polynomials at random points. We present a new sumcheck-based batch-opening protocol (Sect. 3.7) that can batch many openings into one, significantly reducing the prover time, proof size, and verifier time. Our protocol takes $O(k \cdot 2^{\mu})$ field operations for the prover for batching k of μ-variate polynomials, compared to $O(k^2 \mu \cdot 2^{\mu})$ for the previously best protocol [47]. Under certain conditions, we also obtain a more efficient batching scheme with complexity $O(2^{\mu})$, which yields a very efficient commit-and-prove protocol.

Improved Multilinear Commitments. Since HyperPlonk relies on a multilinear commitment scheme, we revisit two approaches to constructing multilinear commitments and present significant improvements to both.

First, in Sect. 5 we use our commit-and-prove protocol to improve the Orion multilinear commitment scheme [51]. Orion is highly efficient: the prover time is strictly linear, taking only $O(2^{\mu})$ field operations and hashes for a multilinear polynomial in μ variables (no group exponentiations are used). The proof size is $O(\lambda\mu^2)$ hash and field elements, and the verifier time is proportional to the proof size. In Sect. 5 we describe Orion+, that has the same prover complexity, but has $O(\mu)$ proof size and $O(\mu)$ verifier time, with good constants. In particular, for security parameter $\lambda = 128$ and $\mu = 25$ the proof size with Orion+ is only about 7 KBs, compared with 5.5 MB with Orion, a nearly 1000x improvement. Using Orion+ in HyperPlonk gives a strictly linear time prover.

Second, in the full version, we show how to generically transform a univariate polynomial commitment scheme into a multilinear commitment scheme using the tensor-product *univariate* Polynomial-IOP from [17]. This yields a new construction for multilinear commitments from FRI [7] by applying the transformation to the univariate FRI-based commitment scheme from [37]. This approach leads to a more efficient FRI-based multilinear commitment scheme compared to the prior construction in [55], which uses recursive techniques. Using this commitment scheme in HyperPlonk gives a quantum-resistant quasilinear-time prover.

Evaluation Results. When optimized and instantiated with the pairing-based multilinear commitment scheme of [40], the proof size of Hyperplonk is $\mu + 5$ group elements and $4\mu + 29$ field elements[1]. Using BLS12-381 as the pairing

[1] The constants depend linearly on the degree of the custom gates. These numbers are for simple degree 2 arithmetic circuits.

group, we obtain 4.7 KB proofs for $\mu = 20$ and 5.5 KB proofs for $\mu = 25$. For comparison, Kopis [44] and Gemini [17], which also have linear-time provers, report proofs of size 39 KB and 18 KB respectively for $\mu = 20$. In Table 1 we show that our prototype HyperPlonk implementation outperforms an optimized commercial-strength Plonk system for circuits with more than 2^{14} gates. It also shows the effects of PLONK arithmetization compared to R1CS by comparing the prover runtime for several important applications. Hyperplonk outperforms Spartan [43] for these applications by a factor of over 60. We discuss the evaluation further in the full version.

Table 1. The prover runtime of Hyperplonk, Spartan [43], and Jellyfish Plonk, for popular applications. The first column (next to the column of the applications) shows the number of R1CS constraints for each application. The third column shows the corresponding number of constraints in HyperPlonk+. Note that the Zexe and the Rollup applications are using the BW6-761 curve.

Application	$\mathcal{R}_{\text{R1}CS}$	Spartan	$\mathcal{R}_{\text{PLONK+}}$	Jellyfish	HyperPlonk
3-to-1 Rescue Hash	288 [1]	422 ms	144 [45]	40 ms	88 ms
Zexe's recursive circuit	2^{22} [52]	6 min	2^{17} [52]	13.1 s	5.1 s
Rollup of 50 private tx	2^{25}	39 min	2^{20} [45]	110 s	38.2 s

1.1 Technical Overview

In this section we give a high level overview of how to make Plonk and its extensions work over the hybercube. We begin by describing Plonk in a modular way, breaking it down into a sequence of elementary components. In Sect. 3 we show how to instantiate each component over the hybercube.

Some components of Plonk rely on the simple linear ordering of the elements of a finite cyclic group induced by the powers of a generator. On the hypercube there is no natural simple ordering, and this causes a problem in the Plookup protocol [27] that is used to implement a lookup gate. To address this we modify the Plookup argument in Sect. 3.6 to make it work over the hypercube. We give an overview of our approach below.

A Review of Plonk. Let us briefly review the Plonk SNARK. Let $\mathcal{C}[G] : \mathbb{F}^{n+m} \to \mathbb{F}$ be a circuit with a total of s gates, where each gate has fan-in two and can be one of addition, multiplication, or a custom gate $G : \mathbb{F}^2 \to \mathbb{F}$. Let $\mathbb{x} \in \mathbb{F}^n$ be a public input to the circuit. Plonk represents the resulting computation as a sequence of $n + s + 1$ triples[2]:

$$\hat{M} := \left\{ \left(L_i, R_i, O_i\right) \in \mathbb{F}^3 \right\}_{i=0,\ldots,n+s}. \tag{1}$$

[2] A more general Plonkish arithmetization [54] supports wider tuples, but triples are sufficient here.

This $\hat{M}$ is a matrix with three columns and $n + s + 1$ rows. The first n rows encode the n public input; the next s rows represent the left and right inputs and the output for each gate; and the final row enforces that the final output of the circuit is zero. We will see how in a minute.

In basic (univariate) Plonk, the prover encodes the cells of $\hat{M}$ using a cyclic subgroup $\Omega \subseteq \mathbb{F}$ of order $3(n + s + 1)$. Specifically, let $\omega \in \Omega$ be a generator. Then the prover interpolates and commits to a polynomial $M \in \mathbb{F}[X]$ such that

$$M(\omega^{3i}) = L_i, \quad M(\omega^{3i+1}) = R_i, \quad M(\omega^{3i+2}) = O_i \quad \text{for } i = 0, \ldots, n + s.$$

Now the prover needs to convince the verifier that the committed M encodes a valid computation of the circuit $\mathcal{C}$. This is the bulk of Plonk system.

Hyperplonk. In $\mathsf{HyperPlonk}$ we instead use the boolean hypercube to encode $\hat{M}$. From now on, suppose that $n+s+1$ is a power of two, so that $n+s+1 = 2^{\mu}$. The prover interpolates and commits to a multilinear polynomial $M \in \mathbb{F}[X^{\mu+2}] = \mathbb{F}[X_1, \ldots, X_{\mu+2}]$ such that

$$M\big(0, 0, \langle i \rangle\big) = L_i, \quad M\big(0, 1, \langle i \rangle\big) = R_i, \quad M\big(1, 0, \langle i \rangle\big) = O_i, \quad \text{for } i = 0, \ldots, n + s. \tag{2}$$

Here $\langle i \rangle$ is the μ-bit binary representation of i. Note that a multilinear polynomial on $\mu + 2$ variables is defined by a vector of $2^{\mu+2} = 4 \times 2^{\mu}$ coefficients. Hence, it is always possible to find a multilinear polynomial that satisfies the $3 \times 2^{\mu}$ constraints in Eq. (2). Next, the prover needs to convince the verifier that the committed M encodes a valid computation of the circuit $\mathcal{C}$. To do so, we need to adapt Plonk to work over the hypercube.

Let us start with the pre-processing algorithm $\mathcal{I}(\mathsf{gp}, \mathcal{C})$ that outputs prover and verifier parameters pp and vp. The verifier parameters vp encode the circuit $\mathcal{C}[G]$ as a commitment to four multilinear polynomials (S_1, S_2, S_3, σ), where $S_1, S_2, S_3 \in \mathbb{F}[X^{\mu}]$ and $\sigma \in \mathbb{F}[X^{\mu+2}]$. The first three are called *selector polynomials* and σ is called the *wiring polynomial.* We will see how they are defined in a minute. There is one more auxiliary multilinear polynomial $I \in \mathbb{F}[X^{\mu}]$ that encodes the input $\mathbb{x} \in \mathbb{F}^n$. This polynomial is defined as $I(\langle i \rangle) = \mathbb{x}_i$ for $i = 0, \ldots, n - 1$, and is zero on the rest of the boolean cube B_{μ}. The verifier, on its own, computes a commitment to the polynomial I to ensure that the correct input $\mathbb{x} \in \mathbb{F}^n$ is being used in the proof. Computing a commitment to I can be done in time $O_{\lambda}(n)$, which is within the verifier's time budget.

With this setup, the Plonk prover $\mathcal{P}$ convinces the verifier that the committed M satisfies two polynomial identities:

The Gate Identity: Let $S_1, S_2, S_3 : \mathbb{F}^{\mu} \to \{0, 1\}$ be the three selector polynomials that the pre-processing algorithm $\mathcal{I}(\mathsf{gp}, \mathcal{C})$ committed to in vp. To prove that all gates were evaluated correctly, the prover convinces the verifier that the following identity holds for all $\mathbf{x} \in B_\mu := \{0,1\}^\mu$:

$$\begin{aligned} 0 = S_1(\mathbf{x}) \cdot \Big(\underbrace{M(0,0,\mathbf{x})}_{L_{[\mathbf{x}]}} + \underbrace{M(0,1,\mathbf{x})}_{R_{[\mathbf{x}]}}\Big) \quad + \quad S_2(\mathbf{x}) \cdot \underbrace{M(0,0,\mathbf{x})}_{L_{[\mathbf{x}]}} \cdot \underbrace{M(0,1,\mathbf{x})}_{R_{[\mathbf{x}]}} \\ + \quad S_3(\mathbf{x}) \cdot \boldsymbol{G}\Big(\underbrace{M(0,0,\mathbf{x})}_{L_{[\mathbf{x}]}}, \underbrace{M(0,1,\mathbf{x})}_{R_{[\mathbf{x}]}}\Big) \quad - \quad \underbrace{M(1,0,\mathbf{x})}_{O_{[\mathbf{x}]}} \quad + \quad I(\mathbf{x}) \end{aligned} \tag{3}$$

where $[\mathbf{x}] = \sum_{i=0}^{\mu-1} \mathbf{x}_i 2^i$ is the integer whose binary representation is $\mathbf{x} \in B_\mu$. For each $i = 0, \ldots, n+s$, the selector polynomials S_1, S_2, S_3 are defined to do the "right" thing:

– addition gate:	$S_1(\langle i \rangle) = 1, \quad S_2(\langle i \rangle) = S_3(\langle i \rangle) = 0$	$(O_i = L_i + R_i\ )$
– multiplication gate:	$S_1(\langle i \rangle) = S_3(\langle i \rangle) = 0, \quad S_2(\langle i \rangle) = 1$	$(O_i = L_i \cdot R_i\ )$
– for a G gate:	$S_1(\langle i \rangle) = S_2(\langle i \rangle) = 0, \quad S_3(\langle i \rangle) = 1$	$(O_i = G(L_i, R_i)\ )$
– if $i < n$ or $i = n + s$:	$S_1(\langle i \rangle) = S_2(\langle i \rangle) = S_3(\langle i \rangle) = 0$	$(O_i = I(\langle i \rangle)\ )$.

The last bullet ensures that O_i is equal to the i-th input for $i = 0, \ldots, n-1$, and that the final output of the circuit, O_{n+s}, is equal to zero.

The Wiring Identity: Every wire in the circuit $\mathcal{C}$ induces an equality constraint on two cells in the matrix $\hat{M}$. In HyperPlonk, the wiring constraints are captured by a permutation $\hat{\sigma} : B_{\mu+2} \to B_{\mu+2}$. The prover needs to convince the verifier that

$$M(\mathbf{x}) = M(\hat{\sigma}(\mathbf{x})) \quad \text{for all } \mathbf{x} \in B_{\mu+2} := \{0,1\}^{\mu+2}. \tag{4}$$

To do so, the pre-processing algorithm $\mathcal{I}(\mathsf{gp}, \mathcal{C})$ commits to a multilinear polynomial $\sigma : \mathbb{F}^{\mu+2} \to \mathbb{F}$ that satisfies $\sigma(\mathbf{x}) = [\hat{\sigma}(\mathbf{x})]$ for all $\mathbf{x} \in B_{\mu+2}$ (recall that $[\hat{\sigma}(\mathbf{x})]$ is the integer whose binary representation is $\hat{\sigma}(\mathbf{x}) \in B_{\mu+2}$). The prover then convinces the verifier that the following two sets are equal (both sets are subsets of $\mathbb{F}^2$):

$$\Big\{\big([\mathbf{x}], M(\mathbf{x})\big)\Big\}_{\mathbf{x} \in B_{\mu+2}} = \Big\{\big([\hat{\sigma}(\mathbf{x})], M(\mathbf{x})\big)\Big\}_{\mathbf{x} \in B_{\mu+2}}. \tag{5}$$

This equality of sets implies that Eq. (4) holds.

Proving the Gate Identity. The prover convinces the verifier that the Gate identity holds by proving that the polynomial defined by the right hand side of Eq. (3) is zero for all $\mathbf{x} \in B_\mu$. This is done using a ZeroCheck IOP, defined in Sect. 3.2. If the custom gate G has total degree d and there are s gates in the circuit, then the total number of terms of the polynomial in Eq. (3) is $(d+1)(s+n+1)$ which is about $(d \cdot s)$. If this were a univariate polynomial, as in Plonk, then a ZeroCheck would require a multi-exponentiation of dimension $(d \cdot s)$ and an FFT of the same dimension. When the polynomial is defined over the hypercube, the ZeroCheck is implemented using the SumCheck protocol in Sect. 3.1, which requires no FFTs. In that section we describe two optimizations to the SumCheck protocol for the settings where the multivariate polynomial has a high degree d in every variable:

- First, in every round of SumCheck the prover sends a polynomial commitment to a univariate polynomial of degree d, instead of sending the polynomial in the clear as in regular SumCheck. This greatly reduces the proof size.
- Second, in standard SumCheck, the prover opens the univariate polynomial commitment at three points: at 0, 1, and at a random $r \in \mathbb{F}$. We optimize this step by showing that opening the commitment at a *single* point is sufficient. This further shortens the final proof.

The key point is that the resulting ZeroCheck requires the prover to do only about $s + d \cdot \mu$ group exponentiations, which is much smaller than $d \cdot s$ in Plonk. The additional arithmetic work that the prover needs to do depends on the number of multiplication gates in the circuit implementing the custom gate G, not on the total degree of G, as in Plonk. As such, we can support much larger custom gates than Plonk.

In summary, proof generation time is reduced for two reasons: (i) the elimination of the FFTs, and (ii) the better handling of high-degree custom gates.

Proving the Wiring Identity. The prover convinces the verifier that the Wiring identity holds by proving the set equality in Eq. (5). We describe a set equality protocol over the hypercube in Sect. 3.4. Briefly, we use a technique from Bayer and Groth [6], that is also used in Plonk, to reduce this problem to a certain ProductCheck over the hypercube (Sect. 3.3). We then use an idea from Quarks [44] to reduce the hypercube ProductCheck to a ZeroCheck, which then reduces to a SumCheck. Again, no FFTs are needed.

Table Lookups. An important extension to Plonk supports circuits with table lookup gates. The table is represented as a fixed vector $\mathbf{t} \in \mathbb{F}^{2^{\mu}-1}$. A table lookup gate ensures that a specific cell in the matrix $\hat{M}$ is contained in $\mathbf{t}$. For example, one can set $\mathbf{t}$ to be the field elements in $\{0, 1, \ldots, B\}$ for some B (padding the vector by 0 as needed). Now, checking that a cell in $\hat{M}$ is contained in $\mathbf{t}$ is a simple way to implement a range check.

Let $f, t : B_{\mu} \to \mathbb{F}$ be two multilinear polynomials. Here the polynomial t encodes the table $\mathbf{t}$, where the table values are $t(B_{\mu})$. The polynomial f encodes the cells of $\hat{M}$ that need to be checked. An important step in supporting lookup gates in Plonk is a way for the prover to convince the verifier that $f(B_{\mu}) \subseteq t(B_{\mu})$, when the verifier has commitments to f and t. The Plookup proof system by Gabizon and Williamson [27] is a way for the prover to do just that. More recently preprocessed alternatives to lookup have been developed[42, 53]. These perform better if the table is known, e.g. a range of values but are in general orthogonal to Plookup.

The problem is that Plookup is designed to work when the polynomials are defined over a cyclic subgroup $\mathbb{G} \subseteq \mathbb{F}^*$ of order q with generator $\omega \in \mathbb{G}$. In particular, Plookup requires a function $\mathsf{next} : \mathbb{F} \to \mathbb{F}$ that induces an ordering of $\mathbb{G}$. This function must satisfy two properties: (i) the sequence

$$\omega, \quad \mathsf{next}(\omega), \quad \mathsf{next}\big(\mathsf{next}(\omega)\big), \quad \ldots, \quad \mathsf{next}^{(q-1)}(\omega) \tag{6}$$

should traverse all of $\mathbb{G}$, and (ii) the function next should be a *linear* function. This is quite easy in a cyclic group: simply define $\mathsf{next}(x) := \omega x$.

To adapt Plookup to the hypercube we need a *linear* function $\mathsf{next} : \mathbb{F}^\mu \to \mathbb{F}^\mu$ that traverses all of B_μ as in Eq. (6), starting with some element $\mathbf{x}_0 \in B_\mu$. However, such an $\mathbb{F}$-linear function does not exist. Nevertheless, we construct in Sect. 3.6 a quadratic function from $\mathbb{F}^\mu$ to $\mathbb{F}^\mu$ that traverses B_μ. The function simulates B_μ using a binary extension and has a beautiful connection to similar techniques used in early PCP work [12]. We then show how to linearize the function by modifying some of the building blocks that Plookup uses. This gives an efficient Plookup protocol over the hypercube. In the full version we use this hypercube Plookup protocol to support lookup gates in HyperPlonk. The resulting protocol is called HyperPlonk+.

2 Preliminaries

Notation: We use λ to denote the security parameter. For $n \in \mathbb{N}$ let $[n]$ be the set $\{1, 2, \ldots, n\}$; for $a, b \in \mathbb{N}$ let $[a, b)$ denote the set $\{a, a+1, \ldots b-1\}$. A function $f(n)$ is $\mathsf{poly}(\lambda)\,(n)$ if there exists a $c \in \mathbb{N}$ such that $f(n) = O(n^c)$. If for all $c \in \mathbb{N}$, $f(n)$ is $o(n^{-c})$, then $f(n)$ is in $\mathsf{negl}(\lambda)\,(n)$ and is said to be **negligible**. A probability that is $1 - \mathsf{negl}(\lambda)(n)$ is **overwhelming**. We use $\mathbb{F}$ to denote a field of prime order p such that $\log(p) = \Omega(\lambda)$.

A *multiset* is an extension of the concept of a set where every element has a positive multiplicity. Two finite multisets are equal if they contain the same elements with the same multiplicities.

A **relation** is a set of pairs $(\mathbb{x}, \mathbb{w})$. An **indexed relation** is a set of triples $(\mathbb{i}, \mathbb{x}; \mathbb{w})$. The index $\mathbb{i}$ is fixed at setup time.

In defining the syntax of the various protocols, we use the following convention concerning public values (known to both the prover and the verifier) and secret ones (known only to the prover). In any list of arguments or returned tuple $(a, b, c; d, e)$, those variables listed before the semicolon are public, and those listed after it are secret. When there is no secret information, the semicolon is omitted.

2.1 Proofs and Arguments of Knowledge

We refer to the full version for more detailed definitions of proofs, arguments, and polynomial interactive oracle proofs. We briefly overview the primitives used and constructed in this paper.

Interactive proofs and arguments of knowledge consist of a non-interactive *preprocessing* phase run by an indexer and an interactive *online* phase between a prover and a verifier. They satisfy the notions of *completeness* and *knowledge soundness*, as well as optionally *zero-knowledge*. The protocols described in this paper are *public coin*, meaning that the verifier only sends random messages.

PolyIOPs. SNARKs can be constructed from information-theoretic proof systems that give the verifier oracle access to prover messages. The information-theoretic proof is then compiled using a cryptographic tool, such as a polynomial commitment. We now define a specific type of information-theoretic proof system called polynomial interactive oracle proofs (PIOPs). In a PIOP, the prover sends oracles to multi-variate polynomials as messages, and the verifier can query these polynomials at arbitrary points. The statement and the index can also consist of oracles to polynomials which the verifier can query. See the full version for formal definitions of PIOPs.

2.2 Multilinear Polynomial Commitments

Multilinear polynomial commitments are commitments where the message space is a multi-linear polynomial. It has the additional property that there exists an efficient argument of knowledge for convincing a verifier that the committed polynomial evaluates to a specific value at a given point.

Multi-linear polynomial commitments can be instantiated from random oracles using the FRI protocol [55], bilinear groups [40], groups of unknown order [3, 19] and discrete logarithm groups[18, 48]. We give a table of polynomial commitments with their different properties in Table 2:

Table 2. Multi-linear polynomial commitment schemes for μ-variate linear polynomials and $n = 2^{\mu}$. The prover time measures the complexity of committing to a polynomial and evaluating it once. The commitment size is constant for all protocols. Unless constants are mentioned, the metrics are assumed to be asymptotic. In the 4th row, ρ denotes the rate of Reed-Solomon codes. In the 5th and 6th rows, k denotes the number of rows of the matrix that represents the polynomial coefficients. The 6th column measures the concrete proof size for $n = 2^{25}$, i.e. $\mu = 25$ and 128-bit security. Legend: BL = Bilinear Group, DL = Discrete Logarithm, RO = Random Oracle, H = Hashes, P = pairings, $\mathbb{G}$ = group scalar multiplications, rec. = Recursive circuit size, univ. = universal setup, trans. = transparent setup, Add. = Additive

Scheme		Prover time: Commit+ Eval	Verifier time	Proof size	$n = 2^{25}$	Setup	Add
KZG-based [40]	BL	$n\ \mathbb{G}_1$	$\log(n)\ P$	$\log(n)\ \mathbb{G}_1$	0.8 KB	Univ	Yes
Dory [38]	BL	$n\mathbb{G}_1 + \sqrt{n}P$	$\log(n)\ \mathbb{G}_T$	$6\log(n)\ \mathbb{G}_T$	30 KB	Trans	Yes
Bulletproofs [18]	DL	$n\ \mathbb{G}$	$n\ \mathbb{G}$	$2\log(n)\ \mathbb{G}$	1.6 KB	Trans	Yes
FRI-based [22]	RO	$n\log(n)/\rho\mathbb{F} + n/\rho H$	$\log^2(n)\frac{\lambda}{-\log\rho}\ H$	$\log^2(n)\frac{\lambda}{-\log\rho}\ H$	250 KB	Trans	No
Orion	RO	$nH + \frac{n}{k} + k$ rec	$\lambda\log^2 nH$	$\lambda\log^2 n\ H$	5.5 MB	Trans	No
Orion + (§5)	BL	$n/k\mathbb{G}_1 + nH +$ $(k\lambda H + \frac{n}{k}\mathbb{F})$ rec.	$\log(n)P$	$4\log n\ \mathbb{G}_1$	7 KB	Univ	No

Virtual Oracles and Commitments. Given multiple polynomial oracles, we can construct virtual oracles to the functions of these polynomials. An oracle to $g([[f_1]], \dots, [[f_k]])$ for some function g is simply the list of oracles $\{[[f_1]], \dots, [[f_k]]\}$ as well as a description of g. In order to evaluate $g([[f_1]], \dots, [[f_k]])$ at some point $\mathbf{x}$ we compute $y_i = f_i(\mathbf{x}) \forall i \in [k]$ and output

$g(y_1, \ldots y_k)$. Equivalently given commitments to polynomials, we can construct a virtual commitment to a function of these polynomials similarly. If g is an additive function and the polynomial commitment is additively homomorphic, then we can use the homomorphism to evaluate.

2.3 PIOP Compilation

PIOP compilation transforms the interactive oracle proof into an interactive argument of knowledge (without oracles) Π. The compilation replaces the oracles with polynomial commitments. Every query by the verifier is replaced with an invocation of the Eval protocol at the query point $\mathbf{z}$. The compiled verifier accepts if the PIOP verifier accepts and if the output of all Eval invocations is 1. If Π is public-coin, it can further be compiled into a non-interactive argument of knowledge (or NARK) using the Fiat-Shamir transform.

Theorem 2.1 (PIOP Compilation [19,24]**).** *If the polynomial commitment scheme Γ has witness-extended emulation, and if the t-round Polynomial IOP for $\mathcal{R}$ has negligible knowledge error, then Π, the output of the PIOP compilation, is a secure (non-oracle) argument of knowledge for $\mathcal{R}$. The compilation also preserves zero knowledge. If Γ is hiding and Eval is honest-verifier zero-knowledge, then Π is honest-verifier zero-knowledge. The efficiency of the resulting argument of knowledge Π depends on the efficiency of both the PIOP and Γ:*

- *Prover time The prover time is equal to the prover time of the PIOP plus the oracle length times the commitment time plus the query complexity times the prover time of Γ.*
- *Verifier time The verifier time is equal to the verifier time of the PIOP plus the verifier time for Γ times the query complexity of the PIOP.*
- *Proof size The proof size is equal to the message complexity of the PIOP times the commitment size plus the query complexity times the proof size of Γ. We say the proof is succinct if the proof size is $O(\log^c(|\mathrm{w}|))$.*

Batching. The prover time, verifier time, and proof size can be significantly reduced using batch openings of the polynomial commitments. For example, the proof size only depends on the number of oracles plus a single batch opening.

3 A Toolbox for Multivariate Polynomials

We begin by reviewing several important PolyIOPs that will serve as building blocks for HyperPlonk. Some are well-known, and some are new.

Notation. From here on, we let $B_\mu := \{0,1\}^\mu \subseteq \mathbb{F}^\mu$ be the boolean hypercube. We use $\mathcal{F}_\mu^{(\leq d)}$ to denote the set of multivariate polynomials in $\mathbb{F}[X_1, \ldots, X_\mu]$ where the degree in each variable is at most d; moreover, we require that each polynomial in $\mathcal{F}_\mu^{(\leq d)}$ can be expressed as a virtual oracle to $c = O(1)$ multilinear polynomials. that is, with the form $f(\mathbf{X}) := g(h_1(\mathbf{X}), \ldots, h_c(\mathbf{X}))$ where $h_i \in$

$\mathcal{F}_\mu^{(\leq 1)}$ $(1 \leq i \leq c)$ is multilinear and g is a c-variate polynomial of total degree at most d. Looking ahead, we restrict ourselves to this kind of polynomials so that we can have sumchecks for the polynomials with linear-time provers.
For polynomials $f, g \in \mathcal{F}_\mu^{(\leq d)}$, we denote $\mathsf{merge}(f, g) \in \mathcal{F}_{\mu+1}^{(\leq d)}$ as

$$\mathsf{merge}(f, g) := h(\mathbf{X}_0, \ldots, \mathbf{X}_\mu) := (1 - \mathbf{X}_0) \cdot f(\mathbf{X}_1, \ldots, \mathbf{X}_\mu) + \mathbf{X}_0 \cdot g(\mathbf{X}_1, \ldots, \mathbf{X}_\mu) \quad (7)$$

so that $h(0, \mathbf{X}) = f(\mathbf{X})$ and $h(1, \mathbf{X}) = g(\mathbf{X})$. In the following definitions, we omit the public parameters $\mathsf{gp} := (\mathbb{F}, \mu, d)$ when the context is clear. We use $\delta_{\text{check}}^{d,\mu}$ to denote the soundness error of the PolyIOP for relation $\mathcal{R}_{\text{check}}$ with public parameter $(\mathbb{F}, d, \mu)$, where check $\in \{\text{sum}, \text{zero}, \text{prod}, \text{mset}, \text{perm}, \text{lkup}\}$.

We defer all proofs to the full version[22] (Table 3).

Table 3. The complexity of PIOPs. d and μ denote the degree and the number of variables of the multivariate polynomials; k in MsetCheck is the length of each element in the multisets; k in BatchEval is the number of evaluations.

Scheme	$\mathcal{P}$ time	$\mathcal{V}$ time	Num of queries	Num of rounds	Proof oracle size	Witness size
SumCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 1$	μ	$d\mu$	$O(2^\mu)$
ZeroCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 1$	μ	$d\mu$	$O(2^\mu)$
ProdCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 2$	$\mu + 1$	$O(2^\mu)$	$O(2^\mu)$
MsetEqChk	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 2$	$\mu + 1$	$O(2^\mu)$	$O(k2^\mu)$
PermCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 2$	$\mu + 1$	$O(2^\mu)$	$O(2^\mu)$
Plookup	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 3$	$\mu + 2$	$O(2^\mu)$	$O(2^\mu)$
BatchEval	$O(2^\mu k)$	$O(k\mu)$	1	$\mu + \log k$	$O(\mu + \log k)$	$O(k2^\mu)$

3.1 SumCheck PIOP for High Degree Polynomials

In this section, we describe a PIOP for the sumcheck relation using the classic sumcheck protocol [39]. However, we modify the protocol and adapt it to our setting of high-degree polynomials.

Definition 3.1 (SumCheck relation). *The relation $\mathcal{R}_{SUM}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = \big((v, [[f]]); f\big)$ where $f \in \mathcal{F}_\mu^{(\leq d)}$ and $\sum_{\boldsymbol{b} \in B_\mu} f(\boldsymbol{b}) = v$.*

Construction. The classic SumCheck protocol [39] is a PolyIOP for the relation $\mathcal{R}_{\text{SUM}}$. When applying the protocol to a polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, the protocol runs in μ rounds where in every round, the prover sends a univariate polynomial of degree at most d to the verifier. The verifier then sends a random challenge point for the univariate polynomial. At the end of the protocol, the verifier checks the consistency between the univariate polynomials and the multi-variate polynomial using a single query to f.
Given a tuple $(\mathbb{x}; \mathbb{w}) = (v, [[f]]; f)$ for μ-variate degree d polynomial f such that $\sum_{\boldsymbol{b} \in B_\mu} f(\boldsymbol{b}) = v$:

- For $i = \mu, \mu - 1, \ldots, 1$:

  - The prover computes $r_i(X) := \sum_{\boldsymbol{b} \in B_{i-1}} f(\boldsymbol{b}, X, \alpha_{i+1}, \ldots, \alpha_\mu)$ and sends the oracle $[[r_i]]$ to the verifier. r_i is univariate and of degree at most d.
  - The verifier checks that $v = r_i(0) + r_i(1)$, samples $\alpha_i \leftarrow \mathbb{F}$, sends α_i to the prover, and sets $v \leftarrow r_i(\alpha_i)$.
- Finally, the verifier accepts if $f(\alpha_1, \ldots, \alpha_\mu) = v$.

Theorem 3.2. *The PIOP for $\mathcal{R}_{SUM}$ is perfectly complete and has knowledge error $\delta_{sum}^{d,\mu} := d\mu/|\mathbb{F}|$.*

We refer to [47] for the proof of the theorem.

Sending r as an Oracle. Unlike in the classic sumcheck protocol, we send an oracle to r_i, in each round, instead of the actual polynomial. This does not change the soundness analysis, as the soundness is still proportional to the degree of the univariate polynomials sent in each round. However, it reduces the communication and verifier complexity, especially if the degree of r is large, as in our application of Hyperplonk with custom gates.

Moreover, the verifier has to evaluate r_i at three points: 0, 1, and α_i. As a useful optimization, the prover can instead send an oracle for the degree $d-2$ polynomial

$$r_i'(X) := \frac{r_i(X) - (1-X) \cdot r_i(0) - X \cdot r_i(1)}{X \cdot (1-X)},$$

along with $r_i(0)$. The verifier then computes $r_i(1) \leftarrow v - r_i(0)$ and

$$r_i(\alpha_i) \leftarrow r_i'(\alpha_i) \cdot (1-\alpha_i) \cdot \alpha_i + (1-\alpha_i) \cdot r_i(0) + \alpha_i \cdot r_i(1).$$

This requires only one query to the oracle of r_i' at α_i and one field element per round.

Computing Sumcheck for High-Degree Polynomials. Consider a multi-variate polynomial $f(\mathbf{X}) := h(g_1(\mathbf{X}), \ldots, g_c(\mathbf{X}))$ such that h is degree d and can be evaluated through an arithmetic circuit with $O(d)$ gates. In the sumcheck protocol, the prover has to compute a univariate polynomial $r_i(X)$ in each round using the previous verifier messages $\alpha_1, \ldots, \alpha_{i-1}$. We adapt the algorithm by [46,49] that showed how the sumcheck prover can be run in time linear in 2^μ using dynamic programming. The algorithm takes as input a description of f as well as the sumcheck round challenges $\alpha_1, \ldots, \alpha_\mu$. It outputs the round polynomials $r_1, \ldots, r_\mu$. The sumcheck prover runs the algorithm in parallel to the sumcheck protocol, taking each computed r_i as that rounds message:

Algorithm 1. Computing $r_1, \dots, r_\mu$ [46,49]

```
procedure SumCheck prover($h, g_1(\mathbf{X}), \dots, g_c(\mathbf{X})$)
    For each $g_j$ build table $A_j : \{0,1\}^\mu \to \mathbb{F}$ of all evaluations over $B_\mu$
    for $i \leftarrow \mu \dots 1$ do
        For each $\boldsymbol{b} \in B_{i-1}$ and each $j \in [c]$, define $r^{(j,\boldsymbol{b})}(X) := (1-X)A_j[\boldsymbol{b},0] + XA_j[\boldsymbol{b},1]$.
        Compute $r^{(\boldsymbol{b})}(X) \leftarrow h(r^{(1,\boldsymbol{b})}(X), \dots, r^{(c,\boldsymbol{b})}(X))$ for all $\boldsymbol{b} \in B_{i-1}$ using Algorithm 2 .
        $r_i(X) \leftarrow \sum_{b \in B_{i-1}} r_b(X)$.
        Send $r_i(X)$ to $\mathcal{V}$.
        Receive $\alpha_i$ from $\mathcal{V}$.
        Set $A_j[\boldsymbol{b}] \leftarrow r^{(j,\boldsymbol{b})}(\alpha_i)$ for each $\boldsymbol{b} \in B_{i-1}$.
    end for
end procedure
```

In [46,49], $r^{(\boldsymbol{b})}(X) := h(r^{(1,\boldsymbol{b})}(X), \dots, r^{(c,\boldsymbol{b})}(X))$ is computed by evaluating h on d distinct values for X, e.g. $X \in \{0, \dots, d\}$ and interpolating the output. This works as h is a degree d polynomial and each $r^{j,\boldsymbol{b}}$ is linear. Evaluating $r^{j,\boldsymbol{b}}$ on d points can be done in d steps. So the total time to evaluate all $r^{j,\boldsymbol{b}}$ for $j \in [c]$ is $c \cdot d$. Furthermore, the circuit has $O(d)$ gates, and evaluating it on d inputs, takes time $O(d^2)$. Assuming that $c \approx d$ the total time to compute $r^{(\boldsymbol{b})}$ with this algorithm is $O(d^2)$ and the time to run Algorithm 1 is $O(2^\mu d^2)$.

We show how this can be reduced to $O(2^\mu \cdot d \log^2 d)$ for certain low depth circuits, such as $h := \prod_c r_c(\mathbf{X})$. The core idea is that evaluating the circuit for h *symbolically*, instead of at d individual points, is faster if fast polynomial multiplication algorithms are used.

We will present the algorithm for computing $h(X) := \prod_{j=1}^{d} r_j(X)$, then we will discuss how to extend this for more general h. Assume w.l.o.g. that d is a power of 2.

Algorithm 2. Evaluating $h := \prod_{j=1}^{d} r_j$

```
Require: $r_1, \dots, r_d$ are linear functions
procedure $h(r_1(X), \dots, r_d(X))$
    $t_{1,j} \leftarrow r_j$ for all $j \in [d]$.
    for $i \leftarrow 1 \dots \log d$ do
        for $j \in [d/2^i]$ do
            $t_{i+1,j}(X) \leftarrow t_{i,2j-1}(X) \cdot t_{i,2j}(X)$    ▷ Using fast polynomial multiplication
        end for
    end for
    return $h = t_{\log_2(d),1}$
end procedure
```

In round i there are $d/2^i$ polynomial multiplications for polynomials of degree 2^{i-1}. In FFT-friendly[3] fields, polynomial multiplication can be per-

[3] These are fields where there exists an element that has a smooth order of at least d.

formed in time $O(d\log(d))$.[4] The total running time of the algorithm is therefore $\sum_{i=1}^{\log_2(d)} \frac{d}{2^i} 2^{i-1}\log(2^{i-1}) = \sum_{i=1}^{\log_2(d)} O(d\cdot i) = O(d\log^2(d))$.

Algorithm 2 naturally extends to more complicated, low-depth circuits. Addition gates are performed directly through polynomial addition, which takes $O(d)$ time for degree d polynomials. As long as the circuit is low-depth and has $O(d)$ multiplication gates, the complexity remains $O(d\log^2(d))$. Furthermore, we can compute $r^k(X)$ for $k \le d$ using only a single FFT of length $\deg(r)\cdot k$ for an input polynomial r. The FFT evaluates r at $\deg(r)\cdot k$ points. Then we raise each point to the power of k. This takes time $O(\deg(r)\cdot k(\log(\deg(r)) + \log(k)))$ and saves a factor of $\log(k)$ over a repeated squaring implementation.

Batching. Multiple sumcheck instances, e.g. $(s, [[f]])$ and $(s', [[g]])$ can easily be batched together. This is done using a random-linear combination, i.e. showing that $(s+\alpha s', [[f]]+\alpha[[g]]) \in \mathcal{L}(\mathcal{R}_{\text{SUM}})$ for a random verifier-generated α [23,48]. The batching step has soundness $\frac{1}{\mathbb{F}}$.

3.2 ZeroCheck PIOP

In this section, we describe a PIOP showing that a multivariate polynomial evaluates to zero everywhere on the boolean hypercube. The PIOP builds upon the sumcheck PIOP in Sect. 3.1 and is a key building block for product-check PIOP in Sect. 3.3. The zerocheck PIOP is also helpful in HyperPlonk for proving the gate identity.

Definition 3.3 (ZeroCheck relation). *The relation $\mathcal{R}_{ZERO}$ is the set of all tuples $(\mathbb{x};\mathbb{w}) = \big(([[f]]); f\big)$ where $f \in \mathcal{F}_\mu^{(\le d)}$ and $f(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\mu$.*

We use an idea from [43] to reduce a ZeroCheck to a SumCheck.

Construction. Given a tuple $(\mathbb{x};\mathbb{w}) = \big(([[f]]); f\big)$, the protocol is the following:

- $\mathcal{V}$ sends $\mathcal{P}$ a random vector $\mathbf{r} \xleftarrow{\$} \mathbb{F}^\mu$
- Let $\hat{f}(\mathbf{X}) := f(\mathbf{X})\cdot eq(\mathbf{X},\mathbf{r})$ where $eq(\mathbf{x},\mathbf{y}) := \prod_{i=1}^{\mu}\big(x_i y_i + (1-x_i)(1-y_i)\big)$.
- Run a sumcheck PolyIOP to convince the verifier that $\big((0,[[\hat{f}]]);\hat{f}\big) \in \mathcal{R}_{\text{SUM}}$.

Batching. It is possible to batch two instances $\big(([[f]]); f\big) \in \mathcal{R}_{\text{ZERO}}$ and $\big(([[g]]); g\big) \in \mathcal{R}_{\text{ZERO}}$ by running a zerocheck on $\big(([[f+\alpha g]]); f+\alpha g\big)$ for a random $\alpha \in \mathbb{F}$. The soundness error of the batching protocol $\frac{1}{\mathbb{F}}$.

Theorem 3.4. *The PIOP for $\mathcal{R}_{ZERO}$ is perfectly complete and has knowledge error $\delta_{zero}^{d,\mu} := d\mu/|\mathbb{F}| + \delta_{sum}^{d+1,\mu} = \mathcal{O}(d\mu/|\mathbb{F}|)$.*

[4] Recent breakthrough results have shown that polynomial multiplication is $O(d\log(d))$ over arbitrary finite fields [35] and there have been efforts toward building practical, fast multiplication algorithms for arbitrary fields [9]. In practice, and especially for low-degree polynomials, using Karatsuba multiplication might be faster.

3.3 ProductCheck PIOP

We describe a PIOP for the product check relation, that is, for a rational polynomial (where both the nominator and the denominator are multivariate polynomials), the product of the evaluations on the boolean hypercube is a claimed value s. The PIOP uses the idea from the Quark system [44, §5], we adapt it to build upon the zerocheck PIOP in Sect. 3.2. Product check PIOP is a key building block for the multiset equality check PIOP in Sect. 3.4.

Definition 3.5 (ProductCheck relation). *The relation $\mathcal{R}_{PROD}$ is the set of all tuples $(\mathbb{x};\mathbb{w}) = \big((s, [[f_1]], [[f_2]]); f_1, f_2\big)$ where $f_1 \in \mathcal{F}_\mu^{(\leq d)}$, $f_2 \in \mathcal{F}_\mu^{(\leq d)}$, $f_2(b) \neq 0\ \forall b \in B_\mu$ and $\prod_{\mathbf{x}\in B_\mu} f'(\mathbf{x}) = s$, where f' is the rational polynomial $f' := f_1/f_2$. In the case that $f_2 = c$ is a constant polynomial, we directly set $f := f_1/c$ and write $(\mathbb{x};\mathbb{w}) = \big((s,[[f]]); f\big)$.*

Construction. The Quark system [44, §5] constructs a proof system for the $\mathcal{R}_{\text{PROD}}$ relation. The proof system uses an instance of the $\mathcal{R}_{\text{ZERO}}$ PolyIOP on $\mu+1$ variables. Given a tuple $(\mathbb{x};\mathbb{w}) = \big((s, [[f_1]], [[f_2]]); f_1, f_2\big)$, we denote by $f' := f_1/f_2$. The protocol is the following:

- $\mathcal{P}$ sends an oracle $\tilde{v} \in \mathcal{F}_{\mu+1}^{(\leq 1)}$ such that for all $\mathbf{x} \in B_\mu$,
$$\tilde{v}(0,\mathbf{x}) = f'(\mathbf{x})\,, \qquad \tilde{v}(1,\mathbf{x}) = \tilde{v}(\mathbf{x},0)\cdot\tilde{v}(\mathbf{x},1)\,, \qquad \tilde{v}(\mathbf{1}) = 0.$$
- Define $\hat{h} := \mathsf{merge}(\hat{f},\hat{g}) \in \mathcal{F}_{\mu+1}^{(\leq \max(2,d+1))}$ where
$$\hat{f}(\mathbf{X}) := \tilde{v}(1,\mathbf{X}) - \tilde{v}(\mathbf{X},0)\cdot\tilde{v}(\mathbf{X},1)\,, \qquad \hat{g}(\mathbf{X}) := f_2(\mathbf{X})\cdot\tilde{v}(0,\mathbf{X}) - f_1(\mathbf{X})\,.$$
Run a ZeroCheck PolyIOP for $\big([[\hat{h}]];\hat{h}\big) \in \mathcal{R}_{\text{ZERO}}$, i.e., the polynomial $\tilde{v}$ is computed correctly.
- $\mathcal{V}$ queries $[[\tilde{v}]]$ at point $(1,\ldots,1,0) \in \mathbb{F}^{\mu+1}$, and checks that the evaluation is s.

Theorem 3.6. *Let $d' := \max(2, d+1)$. The PIOP for $\mathcal{R}_{PROD}$ is perfectly complete and has knowledge error $\delta_{prod}^{d,\mu} := \delta_{zero}^{d',\mu+1} = \mathcal{O}(d'\mu/|\mathbb{F}|)$.*

3.4 Multiset Check PIOP

We describe a multivariate PIOP checking that two multisets are equal. The PIOP builds upon the product-check PIOP in Sect. 3.3. The multiset check PIOP is a key building block for the permutation PIOP in Sect. 3.5 and the lookup PIOP in Sect. 3.6. A similar idea has been proposed in the univariate polynomial setting by Gabizon in a blogpost [26].

Definition 3.7 (Multiset Check relation). *For any $k \geq 1$, the relation $\mathcal{R}_{MSET}^k$ is the set of all tuples $(\mathbb{x};\mathbb{w}) = \big(([[f_1]],\ldots,[[f_k]],[[g_1]],\ldots,[[g_k]]); (f_1,\ldots,f_k,g_1,\ldots,g_k)\big)$ where $f_i, g_i \in \mathcal{F}_\mu^{(\leq d)}$ $(1 \leq i \leq k)$ and the following two multisets of tuples are equal: $\Big\{\mathbf{f_x} := \big[f_1(\mathbf{x}),\ldots,f_k(\mathbf{x})\big]\Big\}_{\mathbf{x}\in B_\mu} = \Big\{\mathbf{g_x} := \big[g_1(\mathbf{x}),\ldots,g_k(\mathbf{x})\big]\Big\}_{\mathbf{x}\in B_\mu}$.*

Basic Construction. We start by describing a PolyIOP for $\mathcal{R}^1_{\text{MSET}}$. The protocol can be obtained from a protocol for $\mathcal{R}_{\text{PROD}}$. Given a tuple $\big(([[f]],[[g]]);(f,g)\big)$, the protocol is the following:

- $\mathcal{V}$ samples and sends $\mathcal{P}$ a challenge $r \xleftarrow{\$} \mathbb{F}$.
- Set $f' := r + f$ and $g' := r + g$
- If $g' \neq 0 \forall b \in B_\mu$ run a ProductCheck PolyIOP for $\big((1,[[f']],[[g']]);f',g'\big) \in \mathcal{R}_{\text{PROD}}$.
- Else the prover sends b such that $g'(b) = 0$ and the verifier accepts if $g(b) = -r$ (this case happens with negligible probability).

Theorem 3.8. *The PIOP for $\mathcal{R}^1_{MSET}$ has perfect completeness and has knowledge error $\delta^{d,\mu}_{mset,1} := 2^{\mu+1}/|\mathbb{F}| + \delta^{d,\mu}_{prod} = \mathcal{O}\big((2^\mu + d\mu)/|\mathbb{F}|\big)$.*

3.5 Permutation PIOP

We describe a multivariate PIOP showing that for two multivariate polynomials $f, g \in \mathcal{F}^{(\leq d)}_\mu$, the evaluations of g on the boolean hypercube is a predefined permutation σ of f's evaluations on the boolean hypercube. The permutation PIOP is a key building block of HyperPlonk for proving the wiring identity.

Definition 3.9 (Permutation relation). *The indexed relation $\mathcal{R}_{PERM}$ is the set of tuples $(\mathbb{i};\mathbb{x};\mathbb{w}) = \big(\sigma;([[f]],[[g]]);(f,g)\big)$, where $\sigma : B_\mu \to B_\mu$ is a permutation, $f, g \in \mathcal{F}^{(\leq d)}_\mu$, and $g(\mathbf{x}) = f(\sigma(\mathbf{x}))$ for all $\mathbf{x} \in B_\mu$.*

Construction. Gabizon et al. [29] construct a permutation argument. We adapt their scheme into a multivariate PolyIOP. The construction uses a PolyIOP instance for $\mathcal{R}_{\text{MSET}}$. Given a tuple $\big(\sigma;([[f]],[[g]]);(f,g)\big)$ where σ is the predefined permutation, the indexer generates two oracles $[[s_{\text{id}}]]$, $[[s_\sigma]]$ such that $s_{\text{id}} \in \mathcal{F}^{(\leq 1)}_\mu$ maps each $\mathbf{x} \in B_\mu$ to $[\mathbf{x}] := \sum_{i=1}^{\mu} \mathbf{x}_i \cdot 2^{i-1} \in \mathbb{F}$, and $s_\sigma \in \mathcal{F}^{(\leq 1)}_\mu$ maps each $\mathbf{x} \in B_\mu$ to $[\sigma(\mathbf{x})]$.[5] The PolyIOP is the following:

- Run a Multiset Check PolyIOP for $\big(([[s_{\text{id}}]],[[f]],[[s_\sigma]],[[g]]);(s_{\text{id}},f,s_\sigma,g)\big) \in \mathcal{R}^2_{\text{MSET}}$.

Theorem 3.10. *The PIOP for $\mathcal{R}_{PERM}$ is perfectly complete and has knowledge error $\delta^{d,\mu}_{perm} := \delta^{d,\mu}_{mset,2} = \mathcal{O}\big((2^\mu + d\mu)/|\mathbb{F}|\big)$.*

3.6 Lookup PIOP

This section describes a multivariate PIOP checking the table lookup relation. The PIOP builds upon the multiset check PIOP (Sect. 3.4) and is a key building block for HyperPlonk+. Our construction is inspired by a univariate PIOP for the table lookup relation called Plookup [27]. However, it is non-trivial to adapt Plookup to the multivariate setting because their scheme requires the existence

[5] Here we further require $|\mathbb{F}| \geq 2^\mu$ so that $[\mathbf{x}]$ never overflow.

of a subdomain of the polynomial that is a cyclic subgroup $\mathbb{G}$ with a generator $\omega \in \mathbb{G}$. Translating to the multilinear case, we need to build an efficient function g that generates the entire boolean hypercube; moreover, g has to be linear so that the degree of the polynomial does not blow up. However, such a linear function does not exist. Fortunately, we can construct a quadratic function from $\mathbb{F}^{\mu}$ to $\mathbb{F}^{\mu}$ that traverses B_{μ}. We then show how to linearize it by modifying some of the building blocks that Plookup uses. This gives an efficient Plookup protocol over the hypercube.

Definition 3.11 (Lookup relation). *The indexed relation $\mathcal{R}_{LOOKUP}$ is the set of tuples $(\mathbb{i};\mathbb{x};\mathbb{w}) = \big(\mathbf{t};[[f]];(f,\mathsf{addr})\big)$ where $\mathbf{t} \in \mathbb{F}^{2^{\mu}-1}$, $f \in \mathcal{F}_{\mu}^{(\leq d)}$, and $\mathsf{addr} : B_{\mu} \to [1, 2^{\mu})$ is a map such that $f(\mathbf{x}) = \mathbf{t}_{\mathsf{addr}(\mathbf{x})}$ for all $\mathbf{x} \in B_{\mu}$.*

Before presenting the PIOP for $\mathcal{R}_{\text{LOOKUP}}$, we first show how to build a *quadratic* function that generates the entire boolean hypercube.

A Quadratic Generator in $\mathbb{F}_{2^{\mu}}$. For every $\mu \in \mathbb{N}$, we fix a *primitive polynomial* $p_{\mu} \in \mathbb{F}_2[X]$ where $p_{\mu} := X^{\mu} + \sum_{s\in S} X^s + 1$ for some set $S \subseteq [\mu - 1]$, so that $\mathbb{F}_2[X]/(p_{\mu}) \cong \mathbb{F}_2^{\mu}[X] \cong \mathbb{F}_{2^{\mu}}$. By definition of primitive polynomials, $X \in \mathbb{F}_2^{\mu}[X]$ is a generator of $\mathbb{F}_2^{\mu}[X] \setminus \{0\}$. This naturally defines a generator function $g_{\mu} : B_{\mu} \to B_{\mu}$ as $g_{\mu}(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{\mu}) = (\boldsymbol{b}_{\mu}, \boldsymbol{b}'_1, \ldots, \boldsymbol{b}'_{\mu-1})$, where $\boldsymbol{b}'_i = \boldsymbol{b}_i \oplus \boldsymbol{b}_{\mu}$ $(i \leq 1 < \mu)$ if $i \in S$, and $\boldsymbol{b}'_i = \boldsymbol{b}_i$ otherwise. Essentially, for a polynomial $f \in \mathbb{F}_2^{\mu}[X]$ with coefficients $\boldsymbol{b}$, $g_{\mu}(\boldsymbol{b})$ is the coefficient vector of $X \cdot f(X)$. Hence the following lemma is straightforward.

Lemma 3.12. *Let $g_{\mu} : B_{\mu} \to B_{\mu}$ be the generator function defined above. For every $\mathbf{x} \in B_{\mu} \setminus \{0^{\mu}\}$, it holds that $\{g_{\mu}^{(i)}(\mathbf{x})\}_{i\in[2^{\mu}-1]} = B_{\mu} \setminus \{0^{\mu}\}$, where $g_{\mu}^{(i)}(\cdot)$ denotes i repeated application of g_{μ}.*

Directly composing a polynomial f with the generator g will blow up the degree of the resulting polynomial; moreover, the prover needs to send the composed oracle $f(g(\cdot))$. Both of which affect the efficiency of the PIOP. We address the issue by describing a trick that manipulates f in a way that simulates the behavior of $f(g(\cdot))$ on the boolean hypercube, but without blowing up the degree.

Linearizing the Generator. For a multivariate polynomial $f \in \mathcal{F}_{\mu}^{(\leq d)}$, we define $f_{\Delta_{\mu}} \in \mathcal{F}_{\mu}^{(\leq d)}$ as

$$f_{\Delta_{\mu}}(\mathbf{X}_1, \ldots, \mathbf{X}_{\mu}) := \mathbf{X}_{\mu} \cdot f(1, \mathbf{X}'_1, \ldots, \mathbf{X}'_{\mu-1}) + (1 - \mathbf{X}_{\mu}) \cdot f(0, \mathbf{X}_1, \ldots, \mathbf{X}_{\mu-1})$$

where $\mathbf{X}'_i := 1 - \mathbf{X}_i$ $(i \leq 1 < \mu)$ if $i \in S$, and $\mathbf{X}'_i := \mathbf{X}_i$ otherwise.

Lemma 3.13. *For every $\mu \in \mathbb{N}$, let $g_{\mu} : B_{\mu} \to B_{\mu}$ be the generator function defined in Lemma 3.12. For every $d \in \mathbb{N}$ and polynomial $f \in \mathcal{F}_{\mu}^{(\leq d)}$, it holds that $f_{\Delta_{\mu}}(\mathbf{x}) = f(g_{\mu}(\mathbf{x}))$ for every $\mathbf{x} \in B_{\mu}$. Moreover, $f_{\Delta_{\mu}}$ has individual degree d and one can evaluate $f_{\Delta_{\mu}}$ from 2 evaluations of f.*

Proof. By definition, f_{Δ_μ} has individual degree d and an evaluation of f_{Δ_μ} can be derived from 2 evaluations of f. Next, we argue that $f_{\Delta_\mu}(\mathbf{x}) = f(g_\mu(\mathbf{x}))$ for every $\mathbf{x} \in B_\mu$.

First, $f_{\Delta_\mu}(0^\mu) = f(g_\mu(0^\mu))$ because $f_{\Delta_\mu}(0^\mu) = f(0^\mu)$ and $g_\mu(0^\mu) = 0^\mu$ by definition of f_{Δ_μ}, g_μ. Second, for every $\mathbf{x} \in B_\mu \setminus \{0^\mu\}$, by definition of g_μ,

$$f(g_\mu(\mathbf{x}_1, \ldots, \mathbf{x}_\mu)) = f(\mathbf{x}_\mu, \mathbf{x}'_1, \ldots, \mathbf{x}'_{\mu-1}),$$

where $\mathbf{x}'_i = \mathbf{x}_i \oplus \mathbf{x}_\mu$ $(i \leq 1 < \mu)$ for every i in the fixed set S, and $\mathbf{x}'_i = \mathbf{x}_i$ otherwise. We observe that $\mathbf{x}_i \oplus \mathbf{x}_\mu = 1 - \mathbf{x}_i$ when $\mathbf{x}_\mu = 1$ and $\mathbf{x}_i \oplus \mathbf{x}_\mu = \mathbf{x}_i$ when $\mathbf{x}_\mu = 0$, thus we can rewrite

$$\begin{aligned} f(\mathbf{x}_\mu, \mathbf{x}'_1, \ldots, \mathbf{x}'_{\mu-1}) &= \mathbf{x}_\mu \cdot f(1, \mathbf{x}^*_1, \ldots, \mathbf{x}^*_{\mu-1}) + (1 - \mathbf{x}_\mu) \cdot f(0, \mathbf{x}_1, \ldots, \mathbf{x}_{\mu-1}) \\ &= f_{\Delta_\mu}(\mathbf{x}_1, \ldots, \mathbf{x}_\mu) \end{aligned}$$

where $\mathbf{x}^*_i = 1 - \mathbf{x}_i$ $(i \leq 1 < \mu)$ for every i in the fixed set S, and $\mathbf{x}^*_i = \mathbf{x}_i$ otherwise. The last equality holds by definition of f_{Δ_μ}. In summary, $f(g_\mu(\mathbf{x}_1, \ldots, \mathbf{x}_\mu)) = f_{\Delta_\mu}(\mathbf{x}_1, \ldots, \mathbf{x}_\mu)$ for every B_μ and the lemma holds.

Construction. Now we are ready to present the PIOP for $\mathcal{R}_{\text{LOOKUP}}$, which is an adaptation of Plookup [27] in the multivariate setting. The PIOP invokes a protocol for $\mathcal{R}^2_{\text{MSET}}$. We introduce a notation that embeds a vector to the hypercube while still preserving the vector order with respect to the generator function. For a vector $\mathbf{t} \in \mathbb{F}^{2^\mu - 1}$, we denote by $t \leftarrow \mathsf{emb}(\mathbf{t}) \in \mathcal{F}_\mu^{(\leq 1)}$ the multilinear polynomial such that $t(0^\mu) = 0$ and $t\big(g_\mu^{(i)}(1, 0^{\mu-1})\big) = \mathbf{t}_i$ for every $i \in [2^\mu - 1]$. By Lemma 3.12, t is well-defined and embeds the entire vector $\mathbf{t}$ onto $B_\mu \setminus \{0^\mu\}$.

For an index $\mathbf{t} \in \mathbb{F}^{2^\mu - 1}$, the indexer generates an oracle $[[t]]$ where $t \leftarrow \mathsf{emb}(\mathbf{t})$. For a tuple $\big(\mathbf{t}; [[f]]; (f, \mathsf{addr})\big)$ where $f(B_\mu) \subseteq t(B_\mu) \setminus \{0\}$, let $(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{2^\mu - 1})$ be the vector where $\boldsymbol{a}_i \in \mathbb{N}$ is the number of appearance of $\mathbf{t}_i$ in $f(B_\mu)$. Note that $\sum_{i=1}^{2^\mu - 1} \boldsymbol{a}_i = 2^\mu$. Denote by $\mathbf{h} \in \mathbb{F}^{2^{\mu+1} - 1}$ the vector

$$\mathbf{h} := \big(\underbrace{\mathbf{t}_1, \ldots, \mathbf{t}_1}_{1 + \boldsymbol{a}_1}, \mathbf{t}_2, \ldots, \mathbf{t}_{i-1}, \underbrace{\mathbf{t}_i, \ldots, \mathbf{t}_i}_{1 + \boldsymbol{a}_i}, \mathbf{t}_{i+1}, \ldots, \mathbf{t}_{2^\mu - 2}, \underbrace{\mathbf{t}_{2^\mu - 1}, \ldots \mathbf{t}_{2^\mu - 1}}_{1 + \boldsymbol{a}_{2^\mu - 1}}\big).$$

We present the protocol below:

- $\mathcal{P}$ sends $\mathcal{V}$ oracles $[[h]]$, where $h \leftarrow \mathsf{emb}(\mathbf{h}) \in \mathcal{F}_{\mu+1}^{(\leq 1)}$.
- Define $g_1 := \mathsf{merge}(f, t) \in \mathcal{F}_{\mu+1}^{(\leq d)}$ and $g_2 := \mathsf{merge}(f, t_{\Delta_\mu}) \in \mathcal{F}_{\mu+1}^{(\leq d)}$, where merge is defined in Eq. (7). Run a multiset check PIOP (Sect. 3.4) for $\big(([[g_1]], [[g_2]], [[h]], [[h_{\Delta_{\mu+1}}]]) ; (f, t, h)\big) \in \mathcal{R}^2_{\text{MSET}}$.
- $\mathcal{V}$ queries $h(0^{\mu+1})$ and checks that the answer equals 0.

Theorem 3.14. *The PIOP for $\mathcal{R}_{LOOKUP}$ is perfectly complete and has knowledge error $\delta^{d,\mu}_{lkup} := \delta^{d,\mu+1}_{mset,2} = \mathcal{O}\big((2^\mu + d\mu)/|\mathbb{F}|\big)$.*

3.7 Batch Openings

This section describes a batching protocol proving the correctness of multiple multivariate polynomial evaluations. Essentially, the protocol reduces multiple oracle queries to different polynomials into a *single* query to a multivariate oracle. The batching protocol is helpful for HyperPlonk to enable efficient batch evaluation openings. In particular, the SNARK prover only needs to compute a single multilinear PCS evaluation proof, even if there are multiple PCS evaluations.

We note that Thaler [47, §4.5.2] shows how to batch two evaluations of a *single* multilinear polynomial. The algorithm can be generalized for multiple evaluations of *different* multilinear polynomials. However, the prover time complexity is $O(k^2\mu \cdot 2^\mu)$ where k is the number of evaluations, and μ is the number of variables. In comparison, our algorithm achieves complexity $O(k \cdot 2^\mu)$ which is $k\mu$-factor faster. Note that $O(k \cdot 2^\mu)$ is already optimal as the prover needs to take $O(k \cdot 2^\mu)$ time to evaluate $\{f_i(\mathbf{z}_i)\}_{i\in[k]}$ before batching.

Definition 3.15 (BatchEval relation). *The relation $\mathcal{R}^k_{BATCH}$ is the set of all tuples $(\mathbb{x};\mathbb{w}) = \left((\mathbf{z}_i)_{i\in[k]}, (y_i)_{i\in[k]}, \left([[f_i]]\right)_{i\in[k]}; (f_i)_{i\in[k]}\right)$ where $\mathbf{z}_i \in \mathbb{F}^\mu$, $y_i \in \mathbb{F}$, $f_i \in \mathcal{F}^{(\leq d)}_\mu$ and $f_i(\mathbf{z}_i) = y_i$ for all $i \in [k]$.*

Remark 3.16. The polynomials $\{f_i\}_{i\in[k]}$ are all μ-variate. This is without loss of generality. E.g., suppose one of the evaluated polynomial f'_j has only $\mu - 1$ variables, we can define $f_j(Y, \mathbf{X}) = Y \cdot f'_j(\mathbf{X}) + (1-Y) \cdot f'_j(\mathbf{X})$ which is essentially f'_j but with μ variables. The same trick easily extends to f'_j with arbitrary $\mu' < \mu$ variables.

Construction. For ease of exposition, we consider the case where $f_1, \dots, f_k$ are *multilinear*. We emphasize that the same techniques can be extended for multivariate polynomials.

Assume w.l.o.g that $k = 2^\ell$ is a power of 2. We observe that $\mathcal{R}^k_{\text{BATCH}}$ is essentially a ZeroCheck relation over the set $Z := \{\mathbf{z}_i\}_{i\in[k]} \subseteq \mathbb{F}^\mu$, that is, for every $i \in [k]$, $f_i(\mathbf{z}_i) - y_i = 0$. Nonetheless, Z is outside the boolean hypercube, and we cannot directly reuse the ZeroCheck PIOP.

The key idea is to interpret each zero constraint as a sumcheck via multilinear extension, so that we can work on the boolean hypercube later. In particular, for every $i \in [k]$, we want to constrain $f_i(\mathbf{z}_i) - y_i = 0$. Since f_i is multilinear, by definition of multilinear extension, this is equivalent to constraining that

$$c_i := \left(\sum_{\boldsymbol{b}\in B_\mu} f_i(\boldsymbol{b}) \cdot eq(\boldsymbol{b}, \mathbf{z}_i)\right) - y_i = 0\,. \tag{8}$$

Note that Eq. (8) holds for every $i \in [k]$ if and only if the polynomial

$\sum_{i\in[k]} eq(\mathbf{Z}, \langle i\rangle) \cdot c_i$ is identically zero, where $\langle i\rangle$ is ℓ-bit representation of $i-1$. By the Schwartz Zippel Lemma, it is sufficient to check that for a random vector $\mathbf{t} \xleftarrow{\$} \mathbb{F}^\ell$, it holds that

$$\sum_{i\in[k]} eq(\mathbf{t}, \langle i\rangle) \cdot c_i = \sum_{i\in[k]} eq(\mathbf{t}, \langle i\rangle) \cdot \left[\left(\sum_{\boldsymbol{b}\in B_\mu} f_i(\boldsymbol{b}) \cdot eq(\boldsymbol{b}, \mathbf{z}_i)\right) - y_i\right] = 0\,. \tag{9}$$

Next, we arithmetize equation (9) and make it an algebraic formula. For every $(i, \boldsymbol{b}) \in [k] \times B_\mu$, we set value $g_{i,\boldsymbol{b}} := eq(\mathbf{t}, \langle i\rangle) \cdot f_i(\boldsymbol{b})$, and define an MLE $\tilde{g}$ for $(g_{i,\boldsymbol{b}})_{i\in[k],\,\boldsymbol{b}\in B_\mu}$ such that $\tilde{g}(\langle i\rangle, \boldsymbol{b}) = g_{i,\boldsymbol{b}} \forall (i, \boldsymbol{b}) \in [k] \times B_\mu$; similarly, we define an MLE $\tilde{eq}$ for $(eq(\boldsymbol{b}, \mathbf{z}_i))_{i\in[k],\,\boldsymbol{b}\in B_\mu}$ where $\tilde{eq}(\langle i\rangle, \boldsymbol{b}) = eq(\boldsymbol{b}, \mathbf{z}_i) \forall (i, \boldsymbol{b}) \in [k] \times B_\mu$. Let $s := \sum_{i\in[k]} eq(\mathbf{t}, \langle i\rangle) \cdot y_i$, then Eq. (9) can be rewritten as

$$\sum_{i\in[k],\boldsymbol{b}\in B_\mu} \tilde{g}(\langle i\rangle, \boldsymbol{b}) \cdot \tilde{eq}(\langle i\rangle, \boldsymbol{b}) = s\,.$$

This is equivalent to prove a sumcheck claim for the degree-2 polynomial $g^* := \tilde{g}(\mathbf{Y}, \mathbf{X}) \cdot \tilde{eq}(\mathbf{Y}, \mathbf{X})$ over set $B_{\ell+\mu}$. Hence we obtain the following PIOP protocol in Algorithm 3. Note that $g^* = \tilde{g} \cdot \tilde{eq}$ is only with degree 2. Thus we can run a classic sumcheck without sending any univariate oracles.

Algorithm 3. Batch evaluation of multi-linear polynomials

```
procedure BatchEval($[f_i \in \mathcal{F}_\mu^{(\le 1)}, \mathbf{z}_i \in \mathbb{F}^\mu, y_i \in \mathbb{F}]_{i=1}^k$)
    $\mathcal{V}$ sends $\mathcal{P}$ a random vector $\mathbf{t} \xleftarrow{\$} \mathbb{F}^\ell$.
    Define sum $s := \sum_{i\in[k]} eq(\mathbf{t}, \langle i\rangle) \cdot y_i$.
    Let $\tilde{g}$ be the MLE for $(g_{i,\boldsymbol{b}})_{i\in[k],\,\boldsymbol{b}\in B_\mu}$ where $g_{i,\boldsymbol{b}} := eq(\mathbf{t}, \langle i\rangle) \cdot f_i(\boldsymbol{b})$.
    Let $\tilde{eq}$ be the MLE for $(eq(\boldsymbol{b}, \mathbf{z}_i))_{i\in[k],\,\boldsymbol{b}\in B_\mu}$ such that $\tilde{eq}(\langle i\rangle, \boldsymbol{b}) = eq(\boldsymbol{b}, \mathbf{z}_i)$.
    $\mathcal{P}$ and $\mathcal{V}$ run a SumCheck PIOP for $\big(s, [[g^*]]; g^*\big) \in \mathcal{R}_{\mathrm{SUM}}$, where $g^* := \tilde{g} \cdot \tilde{eq}$.
    Let $(\boldsymbol{a}_1, \boldsymbol{a}_2) \in \mathbb{F}^{\ell+\mu}$ be the sumcheck challenge vector. $\mathcal{P}$ answers the oracle query $\tilde{g}(\boldsymbol{a}_1, \boldsymbol{a}_2)$.
    $\mathcal{V}$ evaluates $\tilde{eq}(\boldsymbol{a}_1, \boldsymbol{a}_2)$ herself, and checks that $\tilde{g}(\boldsymbol{a}_1, \boldsymbol{a}_2) \cdot \tilde{eq}(\boldsymbol{a}_1, \boldsymbol{a}_2)$ is consistent with the last message of the sumcheck.
end procedure
```

Remark 3.17. If the SNARK is using a homomorphic commitment scheme, to answer query $\tilde{g}(\boldsymbol{a}_1, \boldsymbol{a}_2)$ the prover only needs to provide a single PCS opening proof for a μ-variate polynomial $g'(\mathbf{X}) := \tilde{g}(\boldsymbol{a}_1, \mathbf{X}) = \sum_{i\in[k]} eq(\langle i\rangle, \boldsymbol{a}_1) \cdot eq(\mathbf{t}, \langle i\rangle) \cdot f_i(\mathbf{X})$ on point $\boldsymbol{a}_2$. The verifier can evaluate $\{eq(\langle i\rangle, \boldsymbol{a}_1) \cdot eq(\mathbf{t}, \langle i\rangle)\}_{i\in[k]}$ in time $O(k)$, and homomorphically compute g''s commitment from the commitments to $\{f_i\}_{i\in[k]}$, and checks the opening proof against g''s commitment. Finally, the verifier checks that $g'(\boldsymbol{a}_2)$ matches the claimed evaluation $\tilde{g}(\boldsymbol{a}_1, \boldsymbol{a}_2)$.

Analysis. The PIOP for $\mathcal{R}_{\text{BATCH}}$ is complete and knowledge-sound given the completeness and knowledge-soundness of the sumcheck PIOP.

Next, we analyze the complexity of the protocol: The prover time is $O(k \cdot 2^\mu)$ as it runs a sumcheck PIOP for a polynomial $g^* := \tilde{g} \cdot \tilde{eq}$ of degree 2 and $\mu + \log k$ variables, where $\tilde{g}$ and $\tilde{eq}$ can both be constructed in time $O(k \cdot 2^\mu)$. Note that this is already optimal as the prover anyway needs to take $O(k \cdot 2^\mu)$ time to evaluate $\{f_i(\mathbf{z}_i)\}_{i \in [k]}$ before batching. The verifier takes time $O(\mu + \log k)$ in the sumcheck; the sum s can be computed in time $O(k)$; the evaluation $\tilde{eq}(\boldsymbol{a}_1, \boldsymbol{a}_2) = \sum_{i \in [k]} eq(\boldsymbol{a}_1, \langle i \rangle) \cdot \tilde{eq}(\langle i \rangle, \boldsymbol{a}_2)$ can be derived from $\boldsymbol{a}_1$ and the k evaluations $\{\tilde{eq}(\langle i \rangle, \boldsymbol{a}_2) = eq(\boldsymbol{a}_2, \mathbf{z}_i)\}_{i \in [k]}$ where each evaluation $eq(\boldsymbol{a}_2, \mathbf{z}_i)$ takes time $O(\mu)$. In summary, the verifier time is $O(k\mu)$.

A More Efficient Batching Scheme in a Special Setting. Sometimes one only needs to open a *single* multilinear polynomial at multiple points, where each point is *in the boolean hypercube.* In this setting, we provide a more efficient algorithm with complexity $O(2^\mu)$ which is k times faster than Algorithm 3. We also note that the technique can be used to construct an efficient Commit-and-Prove SNARK scheme from multilinear commitments.

4 HyperPlonk: Plonk on the Boolean Hypercube

Equipped with the building blocks in Sect. 3, we now describe the Polynomial IOP for HyperPlonk. In Sect. 4.1, we introduce $\mathcal{R}_{\text{PLONK}}$ — an indexed relation on the boolean hypercube that generalizes the vanilla Plonk constraint system [29]. We present a Polynomial IOP protocol for $\mathcal{R}_{\text{PLONK}}$ and analyze its security and efficiency in Sect. 4.2.

4.1 Constraint Systems

Notation. For any $m \in \mathbb{Z}$ and $i \in [0, 2^m)$, we use $\langle i \rangle_m = \mathbf{v} \in B_m$ to denote the m-bit binary representation of i, that is, $i = \sum_{j=1}^{m} \mathbf{v}_j \cdot 2^{j-1}$.

Definition 4.1 (HyperPlonk indexed relation). *Fix public parameters* $\mathsf{gp} := \left(\mathbb{F}, \ell, n, \ell_w, \ell_q, f\right)$ *where* $\mathbb{F}$ *is the field,* $\ell = 2^\nu$ *is the public input length,* $n = 2^\mu$ *is the number of constraints,* $\ell_w = 2^{\nu_w}, \ell_q = 2^{\nu_q}$ *are the number of witnesses and selectors per constraint*[6], *and* $f : \mathbb{F}^{\ell_q + \ell_w} \to \mathbb{F}$ *is an algebraic map with degree d. The indexed relation* $\mathcal{R}_{PLONK}$ *is the set of all tuples*

$$(\mathbb{i}; \mathbb{x}; \mathbb{w}) = ((q, \sigma); (p, [[w]]); w),$$

where $\sigma : B_{\mu+\nu_w} \to B_{\mu+\nu_w}$ *is a permutation,* $q \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_q}$, $p \in \mathcal{F}^{(\leq 1)}_{\mu+\nu}$, $w \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_w}$, *such that*

[6] We can pad zeroes if the actual number is not a power of two.

- *the wiring identity is satisfied, that is,* $\big(\sigma; ([[w]], [[w]]); w\big) \in \mathcal{R}_{PERM}$ *(Definition 3.9);*
- *the gate identity is satisfied, that, is,* $\big(([[\tilde{f}]]); \tilde{f}\big) \in \mathcal{R}_{ZERO}$ *(Definition 3.3), where the virtual polynomial* $\tilde{f} \in \mathcal{F}_{\mu}^{(\le d)}$ *is defined as*

$$\tilde{f}(\mathbf{X}) := f(q(\langle 0\rangle_{\nu_q}, \mathbf{X}), \ldots, q(\langle \ell_q - 1\rangle_{\nu_q}, \mathbf{X}), w(\langle 0\rangle_{\nu_w}, \mathbf{X}), \ldots, w(\langle \ell_w - 1\rangle_{\nu_w}, \mathbf{X})); \quad (10)$$

- *the public input is consistent with the witness, that is, the public input polynomial* $p \in \mathcal{F}_{\nu}^{(\le 1)}$ *is identical to* $w(0^{\mu+\nu_w-\nu}, \mathbf{X}) \in \mathcal{F}_{\nu}^{(\le 1)}$*.*

$\mathcal{R}_{\text{PLONK}}$ is general enough to capture many computational models. In the introduction, we reviewed how $\mathcal{R}_{\text{PLONK}}$ captures simple arithmetic circuits. $\mathcal{R}_{\text{PLONK}}$ can be used to capture higher degree circuits with higher arity and more complex gates, including state machine computations.

State Machines. $\mathcal{R}_{\text{PLONK}}$ can model state machine computations, as shown by Gabizon and Williamson [28]. A state machine execution with $n-1$ steps starts with an initial state $\mathsf{state}_0 \in \mathbb{F}^k$ where k is the width of the state vector. In each step $i \in [0, n-1)$, given input the previous state state_i and an online input $\mathsf{inp}_i \in \mathbb{F}$, the state machine executes a transition function f and outputs $\mathsf{state}_{i+1} \in \mathbb{F}^w$. Let $\mathcal{T} := (\mathsf{state}_0, \ldots, \mathsf{state}_{n-1})$ be the execution trace and define $\mathsf{inp}_{n-1} := \bot$, we say that $\mathcal{T}$ is valid for input $(\mathsf{inp}_0, \ldots, \mathsf{inp}_{n-1})$ if and only if (i) $\mathsf{state}_{n-1}[0] = 0^k$, and (ii) $\mathsf{state}_{i+1} = f(\mathsf{state}_i, \mathsf{inp}_i)$ for all $i \in [0, n-1)$.

We build a HyperPlonk indexed relation that captures the state machine computation. W.l.o.g we assume that $n = 2^{\mu}$ for some $\mu \in \mathbb{N}$.[7] Let ν_w be the minimal integer such that $2^{\nu_w} > 2k$. We also assume that there is a low-depth algebraic predicate f_* that captures the transition function f, that is, $f_*(\mathsf{state}', \mathsf{state}, \mathsf{inp}) = 0$ if and only if $\mathsf{state}' = f(\mathsf{state}, \mathsf{inp})$. For each $i \in [0, n)$:

- the online input at the i-th step is $\mathsf{inp}_i := w(\langle 0\rangle_{\nu_w}, \langle i\rangle_{\mu})$;
- the input state of step i is $\mathsf{state}_{\text{in},i} := \big[w\left(\langle 1\rangle_{\nu_w}, \langle i\rangle_{\mu}\right), \ldots, w\left(\langle k\rangle_{\nu_w}, \langle i\rangle_{\mu}\right)\big]$;
- the output state of step i is $\mathsf{state}_{\text{out},i} := \big[w\left(\langle k+1\rangle_{\nu_w}, \langle i\rangle_{\mu}\right), \ldots, w\left(\langle 2k\rangle_{\nu_w}, \langle i\rangle_{\mu}\right)\big]$;
- the selector for step i is $\mathbf{q}_i := q\big(\langle i\rangle_{\mu}\big)$;
- the transition and output correctness are jointly captured by a high-degree algebraic map f',

  $$f'(\mathsf{inp}_i, \mathsf{state}_{\text{in},i}, \mathsf{state}_{\text{out},i}; \mathbf{q}_i) := (1-\mathbf{q}_i)\cdot f_*(\mathsf{state}_{\text{out},i}, \mathsf{state}_{\text{in},i}, \mathsf{inp}_i) + \mathbf{q}_i \cdot \mathsf{state}_{\text{in},i}[0]\,.$$

  For all $i \in [0, n-1)$, we set $\mathbf{q}_i = 0$ so that $\mathsf{state}_{i+1} = f_i(\mathsf{state}_i, \mathsf{inp}_i)$ if and only if

  $$f'(\mathsf{inp}_i, \mathsf{state}_{\text{in},i}, \mathsf{state}_{\text{out},i}; \mathbf{q}_i) = f_*(\mathsf{state}_{\text{out},i}, \mathsf{state}_{\text{in},i}, \mathsf{inp}_i) = 0\,;$$

  we set $\mathbf{q}_{n-1} = 1$ so that $\mathsf{state}_{\text{in},n-1}[0] = 0$ if and only if

  $$f'(\mathsf{inp}_{n-1}, \mathsf{state}_{\text{in},n-1}, \mathsf{state}_{\text{out},n-1}; \mathbf{q}_{n-1}) = \mathsf{state}_{\text{in},n-1}[0] = 0\,.$$

[7] We can pad with dummy states if the number of steps is not a power of two.

Note that we also need to enforce equality between the i-th input state and the $(i-1)$-th output state for all $i \in [n-1]$. We achieve it by fixing a permutation σ and constraining that the witness assignment is invariant after applying the permutation.

Remark 4.2. We can halve the size of the witness and remove the permutation check by using the polynomial shifting technique in Sect. 3.6. Specifically, we can remove output state columns $\mathsf{state}_{\mathrm{out},i}$ and replace it with $\mathsf{state}_{\mathrm{in},i+1}$ for every $i \in [0, n)$.

4.2 The PolyIOP Protocol

In this Section, we present a *multivariate* PIOP for $\mathcal{R}_{\mathrm{PLONK}}$ that removes expensive FFTs.

Construction. Intuitively, the PIOP for $\mathcal{R}_{\mathrm{PLONK}}$ builds on a zero-check PIOP (Sect. 3.2) for custom algebraic gates and a permutation-check PIOP (Sect. 3.5) for copy constraints; consistency between the public input and the online witness is achieved via a random evaluation check between the public input polynomial and the witness polynomial.

Let $\mathsf{gp} := \big(\mathbb{F}, \ell, n, \ell_w, \ell_q, f\big)$ be the public parameters and let $d := \deg(f)$. For a tuple $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = \big((q, \sigma); (p, [[w]]); w\big)$, we describe the protocol in Fig. 1.

Indexer. $\mathcal{I}(q, \sigma)$ calls the permutation PIOP indexer $([[s_{\mathrm{id}}]], [[s_\sigma]]) \leftarrow \mathcal{I}_{\mathrm{perm}}(\sigma)$. The oracle output is $([[q]], [[s_{\mathrm{id}}]], [[s_\sigma]])$, where $q \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_q}$, $s_{\mathrm{id}}, s_\sigma \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_w}$.

The protocol. $\mathcal{P}(\mathsf{gp}, \mathbb{i}, p, w)$ and $\mathcal{V}(\mathsf{gp}, p, [[q]], [[s_{\mathrm{id}}]], [[s_\sigma]])$ run the following protocol.

1. $\mathcal{P}$ sends $\mathcal{V}$ the witness oracle $[[w]]$ where $w \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_w}$.
2. $\mathcal{P}$ and $\mathcal{V}$ run a PIOP for the gate identity, which is a zero-check PIOP (Sect. 3.2) for $\big([[\tilde{f}]]; \tilde{f}\big) \in \mathcal{R}_{\mathrm{ZERO}}$ where $\tilde{f} \in \mathcal{F}^{(\leq d)}_{\mu}$ is as defined in Eq. 10.
3. $\mathcal{P}$ and $\mathcal{V}$ run a PIOP for the wiring identity, which is a permutation PIOP (Sect. 3.5) for $\big(\sigma; ([[w]], [[w]]); (w, w)\big) \in \mathcal{R}_{\mathrm{PERM}}$.
4. $\mathcal{V}$ checks the consistency between witness and public input. It samples $\mathbf{r} \xleftarrow{\$} \mathbb{F}^{\nu}$, queries $[[w]]$ on input $(\langle 0 \rangle_{\mu+\nu_w-\nu}, \mathbf{r})$, and checks $p(\mathbf{r}) \overset{?}{=} w(\langle 0 \rangle_{\mu+\nu_w-\nu}, \mathbf{r})$.

Fig. 1. PIOP for $\mathcal{R}_{\mathrm{PLONK}}$.

Theorem 4.3. *Let* $\mathsf{gp} := \big(\mathbb{F}, \ell, n, \ell_w, \ell_q, f\big)$ *be the public parameters where* $\ell_w, \ell_q = O(1)$ *are some constants. Let* $d := \deg(f)$*. The construction in Fig. 1 is a multivariate PolyIOP for relation* $\mathcal{R}_{PLONK}$ *(Definition 4.1) with soundness error* $\mathcal{O}\big(\frac{2^\mu + d\mu}{|\mathbb{F}|}\big)$ *and the following complexity:*

- *the prover time is* $\mathsf{tp}^{\mathsf{gp}}_{plonk} = \mathcal{O}\big(nd \log^2 d\big)$*;*

- *the verifier time is* $\mathsf{tv}^{\mathsf{gp}}_{plonk} = \mathcal{O}(\mu + \ell)$*;*
- *the query complexity is* $\mathsf{q}^{\mathsf{gp}}_{plonk} = 2\mu + 4 + \log \ell_w$*, that is,* $2\mu + \log \ell_w$ *univariate oracle queries,* 3 *multilinear oracle queries, and* 1 *query to the virtual polynomial* $\tilde{f}$*.*
- *the round complexity and the number of proof oracles is* $\mathsf{rc}^{\mathsf{gp}}_{plonk} = 2\mu + 1 + \nu_w$*;*
- *the number of field elements sent by the prover is* $\mathsf{nf}^{\mathsf{gp}}_{plonk} = 2\mu$*;*
- *the size of the proof oracles is* $\mathsf{pl}^{\mathsf{gp}}_{plonk} = \mathcal{O}\big(n\big)$*; the size of the witness is* $n\ell_w$*.*

Remark 4.4. Two separate sumcheck PIOPs are underlying the HyperPlonk PIOP. We can batch the two sumchecks into one by random linear combination. The optimized protocol has round complexity $\mu + 1 + \log \ell_w$, and the number of field elements sent by the prover is μ. The query complexity $\mu + 3 + \log \ell_w$, that is, $\mu + \log \ell_w$ univariate queries, 2 multilinear queries, and 1 queries to the virtual polynomial $\tilde{f}$.

5 Orion+: A Linear-Time Multilinear PCS with Constant Proof Size

Recently, Xie et al. [50] introduced a highly efficient multilinear polynomial commitment scheme called Orion. The prover time is strictly linear, that is, $O(2^\mu)$ field operations and hashes where μ is the number of variables. For $\mu = 27$, it takes only 115 s to commit to a polynomial and compute an evaluation proof using a single thread on a consumer-grade desktop. The verifier time and proof size is $O_\lambda(\mu^2)$, which also improves the state-of-the-art [16,32]. However, the concrete proof size is still unsatisfactory, e.g., for $\mu = 27$, the proof size is 6 MBs. In this section, we describe a variant of Orion PCS that enjoys similar proving complexity but has $O(\mu)$ proof size and verifier time, with good constants. In particular, for security parameter $\lambda = 128$ and $\mu = 27$, the proof size is less than 10 KBs, which is 600× smaller than Orion for $\mu = 27$.

In this section, we first review the linear-code-based PCS that Orion builds upon. Then we show how Orion+ shrinks the proof size and verifier time. For more details see the full version.

Linear-Time PCS from Tensor-Product Argument [16,32]. Bootle, Chiesa, and Groth [16] propose an elegant scheme for building PCS with strictly linear-time provers. Golovnev et al. [32] later further simplify the scheme. Let $f \in \mathcal{F}^{(\leq 1)}_{\mu}$ be a multilinear polynomial where $f_{\boldsymbol{b}} \in \mathbb{F}$ is the coefficient of $\mathbf{X}_{\boldsymbol{b}} := \mathbf{X}_1^{\boldsymbol{b}_1} \cdots \mathbf{X}_\mu^{\boldsymbol{b}_\mu}$ for every $\boldsymbol{b} \in B_\mu$. Denote by $n = 2^\mu$, $k = 2^\nu < 2^\mu$ and $m = n/k$, one can view the evaluation of f as a tensor product, that is,

$$f(\mathbf{X}) = \langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle \tag{11}$$

where $\mathbf{w} = (f_{\langle 0 \rangle}, \ldots, f_{\langle n-1 \rangle})$, $\mathbf{t}_0 = \big(\mathbf{X}_{\langle 0 \rangle}, \mathbf{X}_{\langle 1 \rangle}, \ldots, \mathbf{X}_{\langle k-1 \rangle}\big)$ and $\mathbf{t}_1 = \big(\mathbf{X}_{\langle 0 \rangle}, \mathbf{X}_{\langle k \rangle}, \ldots, \mathbf{X}_{\langle (m-1)\cdot k \rangle}\big)$. Here $\langle i \rangle$ denotes the μ-bit binary representation of i. Let $E : \mathbb{F}^m \to \mathbb{F}^M$ be a linear encoding scheme, that is, a linear function whose image is a linear code. Golovnev et al. [32, §4.2] construct a PCS scheme as follows:

- **Commitment:** To commit a multilinear polynomial f with coefficients $\mathbf{w} \in \mathbb{F}^n$, the prover $\mathcal{P}$ interprets $\mathbf{w}$ as a $k \times m$ matrix, namely $\mathbf{w} \in \mathbb{F}^{k\times m}$, encodes $\mathbf{w}$'s rows, and obtains matrix $W \in \mathbb{F}^{k\times M}$ such that $W[i,:] = E(\mathbf{w}[i,:])$ for every $i \in [k]$. Then $\mathcal{P}$ computes a Merkle tree commitment for each column of W and builds another Merkle tree T on top of the column commitments. The polynomial commitment C_f is the Merkle root of T.
- **Evaluation proof:** To prove that $f(\mathbf{z}) = y$ for some point $\mathbf{z} \in \mathbb{F}^\mu$ and value $y \in \mathbb{F}$, the prover $\mathcal{P}$ translates $\mathbf{z}$ to vectors $\mathbf{t}_0 \in \mathbb{F}^k$ and $\mathbf{t}_1 \in \mathbb{F}^m$ as above and proves that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ (where $\mathbf{w} \in \mathbb{F}^{k\times m}$ is the message encoded and committed in C_f). To do so, $\mathcal{P}$ does two things:
  - **Proximity check:** The prover shows that the matrix $W \in \mathbb{F}^{k\times M}$ committed by C_f is close to k codewords. Specifically, the verifier sends a random vector $\mathbf{r} \in \mathbb{F}^k$, the prover replies with a vector $\mathbf{y_r} := \mathbf{r} \cdot \mathbf{w} \in \mathbb{F}^m$ which is the linear combination of $\mathbf{w}$'s rows according to $\mathbf{r}$. The verifier checks that the encoding of $\mathbf{y_r}$, namely $E(\mathbf{y_r}) \in \mathbb{F}^M$, is close to $\mathbf{r} \cdot W$, the linear combination of W's rows. This implies that the k rows of W are all close to codewords [32, §4.2].
  - **Consistency check:** The prover shows that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ where $\mathbf{w} \in \mathbb{F}^{k\times m}$ is the k error-decoded messages from $W \in \mathbb{F}$ committed in C_f. The scheme is similar to the proximity check except that we replace the random vector $\mathbf{r}$ with $\mathbf{t}_0$. After receiving the linearly combined vector $\mathbf{y}_0 \in \mathbb{F}^m$, the verifier further checks that $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.

We describe the concrete PCS evaluation protocol below.

Protocol 1 (PCS evaluation [32]): The goal is to check that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ (where $\mathbf{w} \in \mathbb{F}^{k\times m}$ is the message encoded and committed in C_f).

1. $\mathcal{V}$ sends a random vector $\mathbf{r} \in \mathbb{F}^k$.
2. $\mathcal{P}$ sends vector $\mathbf{y_r}, \mathbf{y}_0 \in \mathbb{F}^m$ where $\mathbf{y_r} = \sum_{i=1}^k \mathbf{r}_i \cdot \mathbf{w}[i,:]$, and $\mathbf{y}_0 = \sum_{i=1}^k \mathbf{t}_{0,i} \cdot \mathbf{w}[i,:]$, where $\mathbf{w} \in \mathbb{F}^{k\times m}$ is the message matrix being encoded and committed.
3. $\mathcal{V}$ sends $\mathcal{P}$ a random subset $I \subseteq [M]$ with size $|I| = \Theta(\lambda)$.
4. $\mathcal{P}$ opens the entire columns $\{W[:,j]\}_{j\in I}$ using Merkle proofs, where $W \in \mathbb{F}^{k\times M}$ is the row-wise encoded matrix. That is, $\mathcal{P}$ outputs the column commitment h_j for every column $j \in I$, and provide the Merkle proof for h_j w.r.t. to Merkle root C_f.
5. $\mathcal{V}$ checks that (i) the Merkle openings are correct w.r.t. C_f, and (ii) for all $j \in I$, it holds that $E(\mathbf{y_r})_j = \langle \mathbf{r}, W[:,j] \rangle$ and $E(\mathbf{y}_0)_j = \langle \mathbf{t}_0, W[:,j] \rangle$.
6. $\mathcal{V}$ checks that $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.

Note that by sampling a subset I with size $\Theta(\lambda)$ and checking that $\mathbf{r}\cdot W$, $\mathbf{t}_0 \cdot W$ are consistent with the encodings $E(\mathbf{y_r})$, $E(\mathbf{y}_0)$ on set I, the verifier is confident that $\mathbf{r}\cdot W$, $\mathbf{t}_0 \cdot W$ are indeed close to the encodings $E(\mathbf{y_r})$, $E(\mathbf{y}_0)$ with high probability. By setting $k = \sqrt{n}$, the prover takes $O(n)$ $\mathbb{F}$-ops and hashes; the verifier time and proof size are both $O_\lambda(\sqrt{n})$. Orion describes an elegant code-switching scheme that reduces the proof size and verifier time down to $O_\lambda(\log^2(n))$. However, the concrete proof size is still large. Next, we describe a scheme that has much smaller proof.

Linear-Time PCS with Small Proofs. Similar to Orion (and more generally, the proof composition technique [15,16,32]), instead of letting the verifier check the correctness of $\mathbf{y}_{\mathbf{r}}$, $\mathbf{y}_0$ and the openings of the columns $W[:,j] \forall j \in I$, the prover can compute another (succinct) outer proof validating the correctness of $\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0, W[:,j]$. However, we need to minimize the outer proof's circuit complexity, which is non-trivial. Orion builds an efficient SNARK circuit that removes all of the hashing gadgets, with the tradeoff of larger proof size. We describe a variant of their scheme that minimizes the proof size without significantly increasing the circuit complexity.

Specifically, after receiving challenge vector $\mathbf{r} \in \mathbb{F}^k$, $\mathcal{P}$ instead sends $\mathcal{V}$ commitments $C_{\mathbf{r}}, C_0$ to the messages $\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0$; after receiving $\mathcal{V}$'s random subset $I \subset [M]$, $\mathcal{P}$ computes a SNARK proof for the following statement:

Statement 1 (PCS Eval verification):

- Witness: $\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0 \in \mathbb{F}^m$, $\{W[:,j]\}_{j \in I}$.
- Circuit statements:
  - $C_{\mathbf{r}}$, C_0 are the commitments to $\mathbf{y}_{\mathbf{r}}$, $\mathbf{y}_0$ respectively.
  - For all $j \in I$, it holds that
    - $h_j = H(W[:,j])$ where H is a fast hashing scheme;
    - $E(\mathbf{y}_{\mathbf{r}})_j = \langle \mathbf{r}, W[:,j] \rangle$ and $E(\mathbf{y}_0)_j = \langle \mathbf{t}_0, W[:,j] \rangle$.
  - $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.
- Public output: $\{h_j\}_{j \in I}$, and $C_{\mathbf{r}}, C_0$.

Besides the SNARK proof, the prover also provides the openings of $\{h_j\}_{j \in I}$ with respect to the commitments C_f. Intuitively, the new protocol is "equivalent" to Protocol 1, because the SNARK witness $\{W[:,j]\}_{j \in I}$ and $\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0$ are identical to those committed in $C_f, C_{\mathbf{r}}, C_0$ by the binding property of the commitments; and the SNARK does all of the verifier checks. Unfortunately, the scheme has the following drawbacks:

- Instantiating the commitments with Merkle trees leads to a large overhead on the proof size. In particular, the proof contains $|I|$ Merkle proofs, each with length $O(\log n)$. For 128-bit security, we need to set $|I| = 1568$, and the proof size is at least 1 MBs for $\mu = 20$.
- The random subset I varies for different evaluation instances. It is non-trivial to efficiently lookup the witness $\{E(\mathbf{y}_{\mathbf{r}})_j, E(\mathbf{y}_0)_j\}_{j \in I}$ in the circuit if the set I is dynamic (i.e. we need an efficient random access gadget).
- The circuit complexity is huge. In particular, the circuit is dominated by the commitments to $\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0$ and the hash commitments to $\{W[:,j]\}_{j \in I}$. This leads to $2m + k|I|$ hash gadgets in the circuit. Note that we can't use algebraic hash functions like Rescue [1] or Poseidon [33], which are circuit-friendly, but have slow running times. For $\mu = 26$, $k = m = \sqrt{n}$ and 128-bit security (where $|I| = 1568$), this leads to 13 million hash gadgets where each hash takes hundreds to thousands of constraints, which is unaffordable.

We resolve the above issues via the following observations.

First, a large portion of the multilinear PCS evaluation proof is Merkle opening paths. We can shrink the proof size by replacing Merkle trees with multilinear

PCS that enable efficient batch openings (Sect. 3.7). Specifically, in the committing phase, after computing the hashes of W's columns, instead of building another Merkle tree T of size $M = O(n/k)$ and set the Merkle root as the commitment, the prover can commit to the column hashes using a multilinear PCS (e.g. KZG). Though the KZG committing is more expensive, *the problem size has been reduced to* $O(n/k)$*, thus for sufficiently large* k*, the committing complexity is still approximately* $O(n)$ $\mathbb{F}$*-ops.* A great advantage is that the batch opening proof for $\{h_j\}_{j\in I}$ consists of only $O(\log n)$ group/field elements, with good constant. Even better, when instantiating the outer proof with HyperPlonk(+), the openings can be batched with those in the outer SNARK and thus incur almost no extra cost in proof size.

Second, with Plookup, we can efficiently simulate random access in arrays in the SNARK circuit. For example, to extract witness $\{\mathbf{Y}_{\mathbf{r},j} = E(\mathbf{y}_{\mathbf{r}})_j\}_{j\in I}$, we can build an (online) table T where each element of the table is a pair $(i, E(\mathbf{y}_{\mathbf{r}})_i)$ $(1 \leq i \leq M)$. Then for every $j \in I$, we build a lookup gate checking that $(j, \mathbf{Y}_{\mathbf{r},j})$ is in the table T, thus guarantee that $\mathbf{Y}_{\mathbf{r},j}$ is identical to $E(\mathbf{y}_{\mathbf{r}})_j$. The circuit description is now independent of the random set I and we only need to preprocess the circuit once in the setup phase.

Third, with the help of Commit-and-Prove-SNARKs (CP-SNARK) [2,20,21], there is no need to check the consistency between commitments $C_{\mathbf{r}}, C_0$ and $\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0$ in the circuit. Instead, we can commit $(\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0)$ to a multilinear commitment C, and build a CP-SNARK proof showing that the vector underlying C is identical to the witness vector $(\mathbf{y}_{\mathbf{r}}, \mathbf{y}_0)$ in the circuit. We further observe that C can be a part of the witness polynomials, which further removes the need of an additional CP-SNARK proof.

After applying previous optimizations, the proof size is dominated by the $|I|$ field elements $\{h_j\}_{j\in I}$. We can altogether remove them by applying the CP-SNARK trick again. In particular, since $\{h_j\}_{j\in I}$ are both committed in the polynomial commitment C_f and the SNARK witness commitment, it is sufficient to construct a CP-SNARK proving that they are consistent in the two commitments with respect to set I. We refer to the full version for constructing CP-SNARK proofs from multilinear commitments.

Since the bulk of verification work is delegated to the prover, there is no need to set $k = \sqrt{n}$. Instead, we can set an appropriate $k = \Theta(\lambda/\log n)$ to minimize the outer circuit size. In particular, the circuit is dominated by 2 linear encodings (of length n/k) and $|I|$ hashes (of length k). If we use vanilla HyperPlonk+ as the outer SNARK scheme and use Reinforced Concrete [5] as the hashing scheme that has a similar running time to SHA-256, for $\mu = 30$, $k = 64$ and 128-bit security (where $|I| = 1568$), the circuit complexity is only $\approx 2^{26}$ constraints. And we can expect the running time of the outer proof to be $O_\lambda(n)$.

References

1. Aly, A., Ashur, T., Ben-Sasson, E., Dhooghe, S., Szepieniec, A.: Design of symmetric-key primitives for advanced cryptographic protocols. IACR Trans. Symm. Cryptol. **2020**(3), 1–45 (2020). https://doi.org/10.13154/tosc.v2020.i3.1-45
2. Aranha, D.F., Bennedsen, E.M., Campanelli, M., Ganesh, C., Orlandi, C., Takahashi, A.: ECLIPSE: enhanced compiling method for pedersen-committed zkSNARK engines. Cryptology ePrint Archive, Report 2021/934 (2021). https://eprint.iacr.org/2021/934
3. Arun, A., Ganesh, C., Lokam, S., Mopuri, T., Sridhar, S.: Dew: transparent constant-sized zkSNARKs. Cryptology ePrint Archive, Report 2022/419 (2022). https://eprint.iacr.org/2022/419
4. Babai, L., Moran, S.: Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. J. Comput. Syst. Sci. **36**(2), 254–276 (1988)
5. Barbara, M., et al.: Reinforced concrete: fast hash function for zero knowledge proofs and verifiable computation. Cryptology ePrint Archive, Report 2021/1038 (2021). https://eprint.iacr.org/2021/1038
6. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 263–280. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_17
7. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Fast reed-solomon interactive oracle proofs of proximity. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) ICALP 2018. LIPIcs, vol. 107, pp. 14:1–14:17. Schloss Dagstuhl, July 2018. https://doi.org/10.4230/LIPIcs.ICALP.2018.14
8. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 701–732. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_23
9. Ben-Sasson, E., Carmon, D., Kopparty, S., Levit, D.: Elliptic curve fast fourier transform (ECFFT) part ii: scalable and transparent proofs over all large fields (2022)
10. Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: transparent succinct arguments for R1CS. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 103–128. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_4
11. Ben-Sasson, E., Chiesa, A., Spooner, N.: Interactive oracle proofs. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9986, pp. 31–60. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53644-5_2
12. Ben-Sasson, E., Sudan, M.: Short pcps with polylog query complexity. SIAM J. Comput. **38**(2), 551–607 (2008)
13. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Goldwasser, S. (ed.) ITCS 2012, pp. 326–349. ACM, January 2012. https://doi.org/10.1145/2090236.2090263
14. Bitansky, N., Chiesa, A., Ishai, Y., Paneth, O., Ostrovsky, R.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 315–333. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_18

15. Bootle, J., Cerulli, A., Ghadafi, E., Groth, J., Hajiabadi, M., Jakobsen, S.K.: Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10626, pp. 336–365. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70700-6_12
16. Bootle, J., Chiesa, A., Groth, J.: Linear-time arguments with sublinear verification from tensor codes. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12551, pp. 19–46. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64378-2_2
17. Bootle, J., Chiesa, A., Hu, Y., Orrù, M.: Gemini: elastic SNARKs for diverse environments. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 427–457. Springer, Heidelberg, May/June 2022. https://doi.org/10.1007/978-3-031-07085-3_15
18. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy, pp. 315–334. IEEE Computer Society Press, May 2018. https://doi.org/10.1109/SP.2018.00020
19. Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK compilers. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 677–706. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_24
20. Campanelli, M., Faonio, A., Fiore, D., Querol, A., Rodríguez, H.: Lunar: a toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021. LNCS, vol. 13092, pp. 3–33. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92078-4_1
21. Campanelli, M., Fiore, D., Querol, A.: LegoSNARK: modular design and composition of succinct zero-knowledge proofs. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019, pp. 2075–2092. ACM Press, November 2019. https://doi.org/10.1145/3319535.3339820
22. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: plonk with linear-time prover and high-degree custom gates. Cryptology ePrint Archive, Report 2022/1355 (2022). https://eprint.iacr.org/2022/1355
23. Chiesa, A., Forbes, M.A., Spooner, N.: A zero knowledge sumcheck and its applications. Cryptology ePrint Archive, Report 2017/305 (2017). https://eprint.iacr.org/2017/305
24. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 738–768. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_26
25. Drake, J.: Plonk-style SNARKs without FFTs (2019). https://notes.ethereum.org/DLRqK9V7RIOsTZkab8HmQ?view
26. Gabizon, A.: Multiset checks in plonk and plookup. https://hackmd.io/@arielg/ByFgSDA7D
27. Gabizon, A., Williamson, Z.J.: plookup: a simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315 (2020). https://eprint.iacr.org/2020/315
28. Gabizon, A., Williamson, Z.J.: Proposal: the turbo-plonk program syntax for specifying snark programs (2020). https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf
29. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953 (2019). https://eprint.iacr.org/2019/953

30. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_37
31. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. **18**(1), 186–208 (1989)
32. Golovnev, A., Lee, J., Setty, S., Thaler, J., Wahby, R.S.: Brakedown: linear-time and post-quantum SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/1043 (2021). https://eprint.iacr.org/2021/1043
33. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: a new hash function for zero-knowledge proof systems. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021, pp. 519–535. USENIX Association, August 2021
34. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11
35. Harvey, D., Van Der Hoeven, J.: Polynomial multiplication over finite fields in time. J. ACM (JACM) **69**(2), 1–40 (2022)
36. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_11
37. Kattis, A.A., Panarin, K., Vlasov, A.: RedShift: transparent SNARKs from list polynomial commitments. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022, pp. 1725–1737. ACM Press, November 2022. https://doi.org/10.1145/3548606.3560657
38. Lee, J.: Dory: efficient, transparent arguments for generalised inner products and polynomial commitments. In: Nissim, K., Waters, B. (eds.) TCC 2021. LNCS, vol. 13043, pp. 1–34. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90453-1_1
39. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. J. ACM (JACM) **39**(4), 859–868 (1992)
40. Papamanthou, C., Shi, E., Tamassia, R.: Signatures of correct computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 222–242. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_13
41. Pearson, L., Fitzgerald, J., Masip, H., Bellés-Muñoz, M., Muñoz-Tapia, J.L.: PlonKup: reconciling PlonK with plookup. Cryptology ePrint Archive, Report 2022/086 (2022). https://eprint.iacr.org/2022/086
42. Posen, J., Kattis, A.A.: Caulk+: table-independent lookup arguments. Cryptology ePrint Archive, Report 2022/957 (2022). https://eprint.iacr.org/2022/957
43. Setty, S.: Spartan: efficient and general-purpose zkSNARKs without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 704–737. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_25
44. Setty, S., Lee, J.: Quarks: quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020). https://eprint.iacr.org/2020/1275
45. System, E.: `Jellyfish` jellyfish cryptographic library (2022). https://github.com/EspressoSystems/jellyfish
46. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 71–89. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_5
47. Thaler, J.: Proofs, arguments, and zero-knowledge (2020)

48. Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. In: 2018 IEEE Symposium on Security and Privacy, pp. 926–943. IEEE Computer Society Press, May 2018. https://doi.org/10.1109/SP.2018.00060
49. Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., Song, D.: Libra: succinct zero-knowledge proofs with optimal prover computation. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 733–764. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_24
50. Xie, T., Zhang, Y., Song, D.: Orion: zero knowledge proof with linear prover time. Cryptology ePrint Archive, Report 2022/1010 (2022). https://eprint.iacr.org/2022/1010
51. Xie, T., Zhang, Y., Song, D.: Orion: zero knowledge proof with linear prover time. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 299–328. Springer, Heidelberg, August 2022. https://doi.org/10.1007/978-3-031-15985-5_11
52. Xiong, A.L., et al.: VERI-ZEXE: decentralized private computation with universal setup. Cryptology ePrint Archive, Report 2022/802 (2022). https://eprint.iacr.org/2022/802
53. Zapico, A., Buterin, V., Khovratovich, D., Maller, M., Nitulescu, A., Simkin, M.: Caulk: lookup arguments in sublinear time. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022, pp. 3121–3134. ACM Press, November 2022. https://doi.org/10.1145/3548606.3560646
54. Zcash: PLONKish arithmetization. https://zcash.github.io/halo2/concepts/arithmetization.html (2022)
55. Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: 2020 IEEE Symposium on Security and Privacy, pp. 859–876. IEEE Computer Society Press, May 2020. https://doi.org/10.1109/SP40000.2020.00052