Linear*-Time Permutation Check

Benedikt Bünz, Jessica Chen, and Zachary DeStefano[⋆]

New York University

Abstract. Permutation and lookup arguments are at the core of most deployed SNARK protocols today. Most modern techniques for performing them require a grand product check. This requires either committing to large field elements (E.g. in Plonk) or using GKR (E.g. in Spartan) which has worse verifier cost and proof size. Sadly, both have a soundness error that grows linearly with the input size.
We present two permutation arguments that have $\mathrm{polylog}(n)/|\mathbb{F}|$ soundness error—for reasonable input size $n = 2^{32}$ and field $|\mathbb{F}| = 2^{128}$, the soundness error improves significantly from 2^{-96} to 2^{-120}. Moreover, the arguments achieve $\log(n)$ verification cost and proof size without ever needing to commit to anything beyond the witness. BiPerm only requires the prover to perform $O(n)$ field operations on top of committing to the witness, but at the cost of limiting the choices of PCS. We show a stronger construction, MulPerm, which has no restriction on the PCS choice and its prover performs essentially linear field operations, $n \cdot \tilde{O}(\sqrt{\log(n)})$.
Our permutation arguments generalize to lookups. We demonstrate how our arguments can be used to improve SNARK systems such as HyperPlonk and Spartan, and build a GKR-based protocol for proving non-uniform circuits.

Keywords: Proof system · Permutation check · Sumcheck

⋆ bb, jessicachen, zd@nyu.edu

Table of Contents

1 Introduction

Proof systems enable a prover to convince a verifier that it executed some computation correctly. Modern proof systems formulate the computation as an arithmetic circuit over some finite field $\mathbb{F}$. These proof systems are constructed from an underlying information-theoretic protocol, called a *polynomial interactive oracle proof* or PIOP. In a PIOP, the prover sends oracles to polynomials (univariate or multilinear ones), and the verifier sends random challenges and later accepts or rejects based on queries to the prover-sent oracles. Additionally, an oracle that only depends on the relation is preprocessed and available to the verifier. A PIOP can be compiled to a succinct non-interactive argument of knowledge (SNARK), given a suitable polynomial commitment scheme (PCS) and a hash-function modeled as a random oracle. Key goals in designing PIOPs are to improve the prover time, the number and length of oracles that are sent as well as the query complexity, verifier time, and soundness error. An optimal PIOP would only require sending a single oracle to the witness, have linear prover time, logarithmic (or constant) verifier time and proof size, and logarithmic soundness error.

To investigate how the PIOPs underlying SNARKs fare, we need to study two key components separately: gate check and wiring check. The gate check proves that for any gate (addition or multiplication), the output is computed correctly, e.g.$\mathsf{input}_{\mathsf{left}} \cdot \mathsf{input}_{\mathsf{right}} = \mathsf{output}$. The gate constraint can be checked for all gates using a single invocation of the sumcheck protocol, which requires a single query to the witness oracle, has linear prover time, logarithmic proof size, and verifier time, as well as logarithmic soundness error [Set20; CBBZ23].

This approach is highly efficient; however, SNARKs require a second component: Proving the correctness of the wiring. That is, proving that the input of a gate corresponds to the relevant output of another gate. Plonk [GWC19] and HyperPlonk [CBBZ23] do this by showing that $w(x) = w(\sigma(x)) \ \forall x \in H$ for some set H. Here w on H maps to the wire values and σ is a permutation on H, such that $\sigma(i) = j$ for $i, j \in H$, if and only if the ith and jth wire are identical. Other proof systems such as Spartan[Set20], STARK[BBHR18b], and Quarks[SL20], handle wiring slightly differently but rely on a common primitive at their core, which is the permutation check.

A permutation check enables a prover to convince a $f(x) = g(\sigma(x)) \ \forall x \in H$ for some functions f and g[1] that map from H to some finite field $\mathbb{F}$ and σ is a permutation on H. Permutation checks are also needed for memory checking in proofs of machine computations to show that all read and write operations were performed consistently with the memory. Beyond forming the core of SNARK protocols, permutation checks have applications to lookup arguments, proofs of shuffling, and commitment schemes for sparse polynomials.

Today, there are two key variants of permutation checks in use, both of which can be viewed as PIOP that are suboptimal. The first one uses the fact that $\prod_{x\in H}(f(x) - Y) = \prod_{x\in H}(g(x) - Y)$ if and only if f and g interpolate the same

[1] In some applications $f = g$

set on H [Lip89; Nef01; BG12][2]. The prover uses a *product check*, to prove the correct evaluation of these products at a random point $r \in \mathbb{F}$. There are some downsides to this approach: First, the prover needs to send an oracle to $n = |H|$ additional elements in order to show that the product was computed correctly. These elements will span the entire field, even if f and g only interpolate small elements and even zeros. When the permutation check PIOP is compiled, this oracle corresponds to an additional commitment. Most polynomial commitments [KZG10; BBBPWM18; KZHB25; DP25] are linear not just in the length of the oracle but also in the bitwidth of the committed elements. In the extreme case, if f and g evaluate to $\{0, 1\}$ on H and $|\mathbb{F}| \geq 2^\lambda$, then the prover needs to commit to λ times the size of f and g. Furthermore, the protocol checks that two degree n polynomials are equal; thus, it has soundness error $n/|\mathbb{F}|$.

The second approach views the computation of this product as a depth $\log(n)$ circuit and uses the GKR [GKR08] protocol to compute it. This removes the need to commit to additional elements and reduces the prover time to $O(n)$ field operations. Unfortunately, this comes at a cost of $O(\log^2 n)$ rounds, proof size, and verifier time. It also retains the poor soundness of the product check. Further, using GKR significantly increases the prover latency and adds to the implementation complexity. Additionally, a recent attack was launched against the Fiat-Shamir security of super-constant round GKR [KRS25]. Although this attack is not known to affect GKR for this particular circuit, removing the need for GKR remains an important goal, nonetheless. Prior work [SL20] has proposed combining the protocols and first running $\log \lambda$ rounds of GKR and then using the commitment method. This does improve the verifier time and proof size to $\log \lambda \cdot \log n$ but does not resolve the soundness issue.

1.1 Our contribution

We propose two new permutation checks, BiPerm and MulPerm, that only run a constant (one or two) number of sumcheck protocols and do not require committing to additional polynomials during the runtime of the protocol. The protocols have logarithmic proof size, verifier time, as well as polylog(n) soundness error, and both obviate the need to commit to additional $\Theta(n)$ sized vectors. BiPerm has strictly linear prover time but requires preprocessing and opening an oracle of size $O(n^{1.5})$ but is sparse and only has $O(n)$ non-zero entries. This can be efficiently compiled with *some* polynomial commitments. MulPerm, on the other hand, can be compiled with *any* PCS but requires $n \cdot \tilde{O}(\sqrt{\log n})$ field operations.

In Table 1 we give an overview of our permutation arguments and compare them to prior work. Note that unlike product-check based solution, our protocols only require committing to sublinear amount of data. Compared to the GKR based solutions [Set20; PH23], our protocol has significantly better soundness and verifier time. We also strictly improve on HyperPlonk's lookup argument in terms of prover time and soundness error. In Table 2, we evaluate

[2] This check proves that there exists a permutation between g and f but can easily be turned into a check that a *specific* permutation of $f(H)$ yields $g(H)$.

the PIOPs when instantiated with different PCS schemes, applied to a preprocessed permutation. While BiPerm and Shout [ST25] have the best performance in terms of prover time and verifier time, they can only be reasonably instantiated with group-based PCS schemes that have sublinear opening proofs, such as Hyrax [WTsTW18], Dory [Lee21], or KZH [KZHB25]. MulPerm, on the other hand, has slightly superlinear prover time, but can be efficiently instantiated with any PCS scheme, including hash-based schemes such as FRI [BBHR18a], STIR [ACFY24],WHIR [ACFY25], or Ligero [AHIV17].

PIOP	$\lvert$**Comm.**$\rvert(\mathbb{F})$	**Rounds**	$\lvert\mathbb{i}\rvert$	$wt(\mathbb{i})$	**PCS**	$\delta_s(/\lvert\mathbb{F}\rvert)$
ProdCheck	$n + \log n$	$\log n$	n	$n \log n$	Any	n
GKR-based	$\log^2 n$	$\log^2 n$	n	$n \log n$	Any	n
GKR-k	$n/2^k$	$k \cdot \log n$	n	$n \log n$	Any	n
HyperPlonk (3.8)	$\log^2 n$	$2 \log n$	$n \log n$	$n \log n$	Any	$\log^2 n$
Naïve Perm (§4.5)	$\log n$	$\log n$	n^2	n	None	$\log n$
BiPerm (§6)/Shout	$\log n$	$\log n$	$n^{1.5}$	$2n$	Sparse	$\log n$
MulPerm (§7)	$\log^{1.5} n$	$2 \log n$	$n \log n$	$n \log n$	Any	$\log^{1.5} n$

Table 1. Properties of various permcheck PIOPs. n is the number of permuted elements. Among the properties: $\lvert$Comm.$\rvert(\mathbb{F})$ is the number of field elements we commit to, $\lvert\mathbb{i}\rvert$ is the length of oracles preprocessed in the index; $wt(\mathbb{i})$ is the weight of index, i.e. the nonzero number of bits in the index; $\lvert$P msgs$\rvert$ refers to the total number of field elements of prover messages; and δ_s is the soundness error. Among the PIOPs: GKR-k is the permcheck PIOP that runs the first k rounds of GKR and then directly runs ProdCheck for the remaining computation; Naïve Perm is the most naïve permcheck PIOP discussed in Section 4.5. Sparse PCS refers to the ones with which committing and opening zeroes are free.

PIOP	KZG (MSM)		Dory (MSM)		FRI (#H)		Ligero (#H)			
	Pre	Prove	Pre	Prove	Pre	Prove	Pre	Prove	#$\mathbb{F}$ ops	V time
ProdCheck	$n\mathbb{W}$	$n\mathbb{F}$	$n\mathbb{W}$	$n\mathbb{F}$	n	n	n	n	n	$\log n$
GKR-based	$n\mathbb{W}$	$n\mathbb{F}$	$n\mathbb{W}$	$n^{0.5}\mathbb{F}$	n	n	n	0	n	$\log^2 n$
GKR-k	$n\mathbb{W}$	$n\mathbb{F}$	$n\mathbb{W}$	$n/2^k + n^{0.5}\mathbb{F}$	n	n	n	$n/2^k$	n	$k \log n$
HyperPlonk	$n\mathbb{W}$	$n\mathbb{F}$	$n\mathbb{W}$	$n^{0.5}\mathbb{F}$	n	n	n	0	$n \cdot \tilde{O}(\log n)$	$2 \log n$
Naïve Perm	nB	$n^2\mathbb{F}$	nB	$n\mathbb{F}$	n^2	n^2	n^2	0	n	$\log n$
BiPerm	nB	$n^{1.5}\mathbb{F}$	nB	$n^{0.75}\mathbb{F}$	$n^{1.5}$	$n^{1.5}$	$n^{1.5}$	0	n	$\log n$
MulPerm	$n\mathbb{W}$	$n\mathbb{F}$	$n\mathbb{W}$	$n^{0.5}\mathbb{F}$	n	n	n	0	$n \cdot \tilde{O}(\sqrt{\log n})$	$2 \log n$

Table 2. Cryptographic operations in preprocessing (Pre) and proving (Prove), field operations (#$\mathbb{F}$ ops), and the resulting verifier time (V time) of various permutation arguments constructed by combining permcheck PIOPs with KZG [KZG10], Dory [Lee21], FRI [BBHR18a], and Ligero [AHIV17]. n is the number of permuted elements. GKR-k is the permcheck PIOP that runs the first k rounds of GKR and then directly runs prodcheck for the remaining computation; Naïve Perm is the most naïve permcheck PIOP discussed in Section 4.5. MSM means the size of multi-scalar multiplications, and #H means the number of hashes. $\mathbb{W} \subseteq \mathbb{F}$ is a subfield containing small elements. Only the dominating factor is listed. The non-optimal costs are red.

Extension to lookup gates and non-preprocessed maps Our idea extends to a family of protocols. We can also work with non-injective mapping ρ that maps from H to some set H', possibly different from H. This can be used to build *lookup gates.* A witness satisfies lookup gates if all its elements can be looked up in some table t. In that application, we prove that $w(\rho(x)) = t(x) \ \ \forall x \in H$. Another set of protocols and related applications arise when the mapping is not preprocessed but rather *prover-provided.* In that case, our ideas still apply but we need to additionally prove that the permutation or map is well-formed. In Table 3 we present all variations of permutation and lookup arguments and their respective applications.

Protocol	Application	Relation $\forall x \in H$	
Fixed Perm (§7,§6)	HyperPlonk (§3)	Fixed $\pi \in \mathsf{PERM}(H)$	$f(\pi(x)) = g(x)$
Fixed Lookup (§9)	SPARK (§3)	Fixed $\sigma \in \mathsf{FUN}(H, H')$	$f(x) = T(\sigma(x))$
Provided Perm (§8)	Memory-Checking	$\exists \pi \in \mathsf{PERM}(H)$	$f(\pi(x)) = g(x)$
Provided Lookup (§9.2)	Lookups/Range Checks	$\exists \sigma \in \mathsf{FUN}(H, H')$	$f(x) = T(\sigma(x))$

Table 3. Permutation and lookup arguments and their applications. Fixed means that the map is preprocessed as part of the index. Provided means that the prover sends it as an oracle.

1.2 Applications

Our arguments can serve as drop-in replacements for the currently used permutation and lookup arguments. We outline a few specific applications where the efficiency properties of our schemes are beneficial.

Single commitment SNARK Using either BiPerm or MulPerm in combination with HyperPlonk, one can build a PIOP for NP that has linear or nearly linear prover time, logarithmic verifier time and polylogarithmic soundness error, all of which combined are better than prior known constructions. Replacing the permutation argument in HyperPlonk with BiPerm or MulPerm would reduce the number of oracles that are sent to a single oracle to the witness. Further, we can batch sum check arguments from the permutation check with the sumchecks of the gate constraint check, which means that this witness oracle is only queried at a single point. This transformation is particularly valuable if the witness is sparse or has low weight and the PIOP is compiled with a polynomial commitment that can take advantage of these properties. We detail these improvements in Section 3.

Better SPARK compiler A similar transformation can be applied to the SPARK compiler which is a key component of the Spartan [Set20] family of protocols. The SPARK compiler enables committing to large but sparse polynomials using their dense representations. In Spartan this is used to commit to the R1CS matrices. SPARK internally uses a permutation argument, either based on GKR [GKR08] or a variation of the Nef product check [SL20]. We can replace this permutation check with our permutation check (or more precisely with the extension to lookup arguments). This improved version of SPARK inherits the properties from BiPerm and MulPerm, i.e. has improved soundness, and lower prover time (vs. Quarks [SL20]) or lower verifier time/proof size (vs. the GKR variant). Combining it with the Spartan protocol or SuperSpartan for CCS [STW23], yields a new SNARK for R1CS and CCS that requires only a single oracle to the witness and only one or two sumcheck protocols. We show the details in Section 3.

Lookup gates Lookup gates show that a witness value is inside a table of values. Lookup arguments are useful for range proofs and for non-arithmetic operations [BCGJM18]. In a range proof, the table consists of the values in the range. They can be used to implement non-arithmetic operations using a table that contains pairs of inputs and outputs [AST24]. We build MulLookup the first lookup argument with poly-logarithmic soundness error. MulLookup has similar prover performance as Lasso [STW24] but better soundness error, verifier time, proof size, and less requirements on the structure of the table. Compared to other lookups, such as LogUp [Hab22] we commit to less additional information, and the prover's only dependence on T is the time to evaluate a mulilinear evaluation of the table. MulLookup argument generalizes our permutation argument. Instead of the prover sending a permutation, the prover sends a map from the witness to the table. We then prove that this map is well-formed, i.e. that the output of the map is binary and that when applied to the witness we only get values in the table. Notably, the argument preserves the efficiency and soundness properties of the permutation argument. The prover only needs to commit to the map, and after that no additional commitment operations are needed. The prover's field operations are $n \cdot \tilde{O}(\sqrt{\log(n)})$ for tables that are smaller than n and $O(n(\log T - \log n))$ for tables of size T that are larger than n. The prover needs to open the witness and the table polynomials at a single point. While generically, this will require $\Theta(T)$ field operations, for many tables of interest the table polynomial can be efficiently evaluated. For example for range proofs the multilinear polynomial describing the table is simply $t(\boldsymbol{X}) = \sum_{i=0}^{\log T-1} X_i 2^i$, and can be evaluated in time $O(\log T)$. Compared to other lookup arguments, ours is the first with only a poly-logarithmic soundness error.

The efficiency properties of our protocol only hold for tables up to $T < n^{\log(n)}$. Additionally, the opening proof might become a bottleneck for tables that are too large. However, in the case that these large tables are structured one can use the techniques from Lasso [STW24]. Lasso at it's core relies on the Spark compiler. Our permutation argument provides an efficient alternative to Spark, as described above. This replacement reduces the verification time, proof size and soundness error of Lasso.

PIOP	P **time**	V **time**	wt(**Comm.**)	**Table Restriction**
Lasso	$O(n)$	$\log^2 n$	$n \log T$ bits	Decomposable
Generalized-Lasso	$O(c \cdot n)$	$\log^2 n$	$c \cdot n\lambda$ bits	MLE-structured
LogUp	$O(n+T)$	$\log n$	$n\lambda$ bits	Any
HyperPlonk	$O(n+T)$	$\log n$	$n\lambda$ bits	Any.
Plookup	$O((n+T)\log n + T)$	$\log n$	$n\lambda$ bits	Any
MulLookup	$n \cdot \tilde{O}(\sqrt{\log T})$	$O(\log n)$	$n \log T$ bits	$T < n$
MulLookup	$O(n \cdot (\log T - \log n))$	$O(\log n)$	$n \log T$ bits	$n < T < n^{\log n}$

Table 4. Properties of various lookup PIOPs. Prover time in terms of field operations, Verifier time, size of additional commitments, and table restrictions. T is the size of the table and n is the size of the witness. All PIOPs evaluate the table polynomial once and the witness polynomial once. The properties of Lasso listed assume Lasso uses GKR-based permutation check for memory-checking; the values will change if it were to use other permutation checks. The last column gives restrictions on the table for the efficiency properties to hold.

1.3 BiPerm and Shout

Shout [ST25] introduces a lookup argument that has the same structure and efficiencies as BiPerm, when parameterized with $d = 2$. Shout can be used as a permutation argument for an indexed/preprocessed table, as the preprocessing guarantees that the commitment mapping is indeed a permutation. Shout, however, does not give a prover-provided permutation argument. Shout employs a specific argument of one-hotness, which suffices for the lookup case; however, this does not prove that a prover-provided map is a permutation. We, instead, provide a generic argument for transforming a lookup into a permutation by checking that the map is invertible. Most importantly, BiPerm and Shout both require a PCS where the preprocessing and opening cost only depend on the sparseness of the committed polynomial. Our main contribution is MulPerm, a lookup and permutation argument that can be used with any PCS.

1.4 R1CS-style GKR

We also present an improvement of the famous GKR scheme [GKR08] for circuits, detailed in Section 3.3. GKR enables proving deterministic circuits without requiring committing to intermediate values. It has recently garnered renewed attention by practitioners [Sou23; LZ25; HLQXYZZ25; Sou25] especially in it's commit-and-prove version [WTsTW18], that enables combining proofs for the non-deterministic portion of a circuit with GKR for the deterministic portion. Downsides of standard GKR include constant fan-in, logarithmic depth, and strong uniformity requirements on the circuit. [LZ19] showed a way to perform GKR for non-standard circuits. However, there is a tradeoff between the fan-in and the prover cost in their protocol. Specifically, the cost and degree of the sumcheck increase as the gate degree of the circuit increases.

We build a Spartan-style GKR proof system for layered circuits, where each layer is defined by preprocessed R1CS-style matrices. This proof system allows for much more flexibility in the circuit structure and handles additions for free. The key difficulty is that we have to outsource the verifier work of evaluating the multilinear evaluations (MLEs) of these matrices at a random point. We can use a variant of our lookup argument to achieve this efficiently. Prior lookup arguments would have either required committing to $\Theta(n)$ data (e.g. [SL20]), which removes the benefit of using GKR in the first place, or used a uniform variant of GKR to prove correctness of the evaluation (e.g. [Set20]) which adds $\log(n)$ sumchecks to the evaluation, even for constant depth circuits.

1.5 Additional related work

A recent line of work [ZBKMNS22; ZGKMR22; GK22; EFG22] built lookup arguments for preprocessed tables, where the prover is entirely independent of the table size. All these works require a trusted setup and rely on pairings for security. In contrast, our lookup argument requires the prover to evaluate the table, represented as an MLE, at a random point, thus, it is generically linear in the table size. However, we formulate our lookup argument as a PIOP that can be compiled with various polynomial commitments from different assumptions.

2 Technical overview

The core idea of both protocols is to formulate the permutation claim $f(\sigma(x)) = g(x)$ as a sumcheck, i.e., showing that

$$\sum_{x \in H} f(x)\mathbb{1}_\sigma(x,y) = g(y) \ \forall y \in H$$

Here

$$\mathbb{1}_\sigma(X \in H, Y \in H) = \begin{cases} 1 & \sigma(X) = Y \\ 0 & \text{otherwise} \end{cases}.$$

In order to prove this statement efficiently, we need to arithmetize $\mathbb{1}_\sigma$. A natural choice would be to arithmetize it as a multilinear extension, and set H to be the boolean hypercube over $\mu = \log_2(n)$ variables. Unfortunately, the multilinear extension of $\mathbb{1}_\sigma(x,y)$, $\tilde{\mathbb{1}}_{[\sigma]}$ has 2μ variables, thus takes up $2^{2\mu} = n^2$ points to represent. Even though, it is sparse with only n non-zero entries, there exists no polynomial commitment today which can open polynomials with n^2 points in sublinear time[3]. HyperPlonk [CBBZ23][4] first proposed arithmetizing $\mathbb{1}_\sigma$ as a product of μ multilinear extensions of $\mathbb{1}_{\sigma_i}, \forall i \in [\mu]$, such that

$$\mathbb{1}_{\sigma_i}(\boldsymbol{X}, Y_i) = \begin{cases} 1 & \sigma_i(X) = Y_i \\ 0 & \text{otherwise} \end{cases}.$$

Here $\boldsymbol{X}, \boldsymbol{Y} \in \{0,1\}^\mu$ and $\sigma_i : \{0,1\}^\mu \to \{0,1\}$ maps to the ith bit of σ. Using this arithmetization, we can write a permutation check as

$$\sum_{x \in \{0,1\}^\mu} f(x) \cdot \prod_{i=1}^{\mu} \tilde{\mathbb{1}}_{\sigma_i}(x, y_i) = g(y) \ \forall y \in \{0,1\}^\mu.$$

Assuming that f and g are multilinear, this can be proven using degree $\mu + 2$ sumchecks in time $O(n \cdot \mu^2)$ and with soundness error $O(\frac{\mu^2}{|\mathbb{F}|})$. We show that we can improve on both the soundness and the prover time of the protocol.

In Section 6, we describe the details of BiPerm, a permcheck PIOP using $\ell = 2$ that yields a linear-time permutation argument if compiled using certain PCSs. In Section 7, we introduce MulPerm, a permcheck PIOP using larger ℓ that can be compiled using any PCS into an almost linear-time permutation argument.

[3] The best polynomial commitments have opening algorithms that take time $\sqrt{|f|}$, where $|f|$ is the size of the committed polynomial.

[4] HyperPlonk presents two permutation checks. One is a multilinear version of the product check, that is similar to Quarks[SL20]. The other one is described above. For distinction we will refer to the first one as the Quarks permutation check and the second one as the HyperPlonk one.

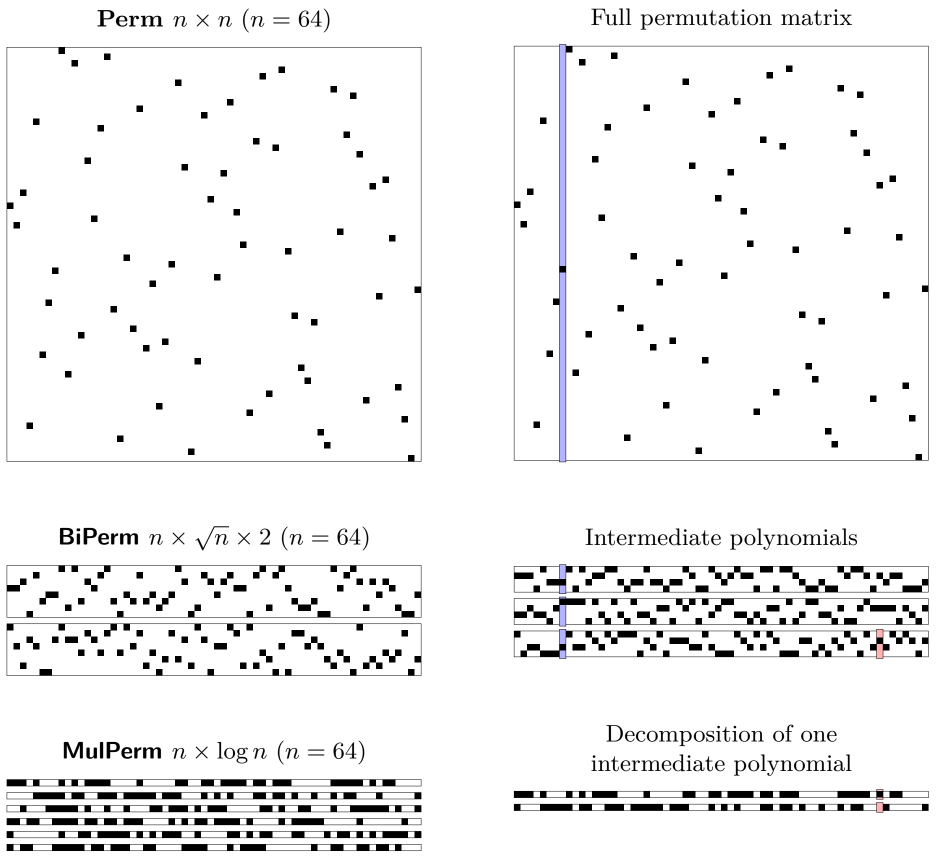
Perm $n \times n$ $(n = 64)$
Full permutation matrix
BiPerm $n \times \sqrt{n} \times 2$ $(n = 64)$
Intermediate polynomials
MulPerm $n \times \log n$ $(n = 64)$
Decomposition of one intermediate polynomial

Fig. 1. High-level visual illustrations of polynomials various permutation arguments in this paper. **Left:** Illustration of the preprocessed eq polynomials in naïve Perm (Section 5), BiPerm (Section 6), and MulPerm (Section 7). Each large rectangle is a 0/1 matrix representing evaluations of a committed polynomial where black cell represents the value 1 at that point (all other points are 0). Within each rectangle, each row represents a $\boldsymbol{X} \in B_\mu$ and each column represents a $\boldsymbol{Y} \in B_\mu$ (potentially encoded across multiple polynomials). In Perm and BiPerm, each preprocessed matrix is n-sparse. In MulPerm, the number of 0 and 1 entries are roughly equal. For this small case, $\log n$ and $\sqrt{n}$ do not differ significantly; however for larger n, the difference is far larger. **Right:** Illustration of polynomials in the two sumchecks in MulPerm ($n = 64$). The full permutation matrix is decomposed into $\lceil\sqrt{\log n}\rceil$ matrices each of which is interpreted as a multilinear polynomial. Each intermediate matrix is further decomposed into $\lfloor\sqrt{\log n}\rfloor$ matrices. The first sum-check verifies consistency between the columns in the full permutation and the intermediate polynomials. The second sum-check batch-verifies consistency between the columns in the intermediate polynomials and their decompositions. This keeps the degree in each variable low while avoiding commitments to large polynomials.

BiPerm, a fully linear permutation check for some PCS BiPerm has fully linear prover time and only runs one degree 3 sumcheck protocol, with $O(\mu)$ verifier time and $O(\frac{\mu}{|\mathbb{F}|})$ soundness error. The core insight is that we can write $\mathbb{1}_\sigma$ as the product of two functions:

$$\mathbb{1}_{\sigma_L}(X, Y_L) = \begin{cases} 1 & \sigma_L(X) = Y_L \\ 0 & \text{otherwise} \end{cases}$$

and

$$\mathbb{1}_{\sigma_R}(X, Y_R) = \begin{cases} 1 & \sigma_R(X) = Y_R \\ 0 & \text{otherwise} \end{cases}.$$

Here Y_L, Y_R are the first $\mu/2$ and second $\mu/2$ half of $Y \in \{0,1\}^\mu$ and $\sigma_L(X), \sigma_R(X)$ map to the first and second half of the bits of $\sigma(X)$ respectively. Using at the multilinear extensions of $\mathbb{1}_{\sigma_L}$ and $\mathbb{1}_{\sigma_R}$, we can write the claim as a degree 3 sumcheck:

$$\sum_{x \in \{0,1\}^\mu} f(x) \cdot \tilde{\mathbb{1}}_{\sigma_L}(x, y_L) \cdot \tilde{\mathbb{1}}_{\sigma_R}(x, y_R) = g(y) \ \forall y \in \{0,1\}^\mu$$

This almost directly gives us our first protocol BiPerm. As a limitation BiPerm, uses $\tilde{\mathbb{1}}_{\sigma_L}(x, y_L)$ and $\tilde{\mathbb{1}}_{\sigma_L}(x, y_L)$, which are multilinear polynomials with 1.5μ variables and of size $n^{1.5}$ with at most n non-zero entries. Importantly, these polynomials can also be evaluated efficiently using $O(n)$ field operations. $\tilde{\mathbb{1}}_{[\sigma_L]}, \tilde{\mathbb{1}}_{[\sigma_R]}$ after PIOP compilation are committed to during preprocessing and opened at a random point during the evaluation protocol. This can be done in linear time for some polynomial commitment schemes, such as Hyrax, KZH, and Dory [WTsTW18; KZHB25; Lee21]. These PCS schemes have commitment time that scales linear in the sparsity and opening time that (beyond evaluation) is only square root in the size of the polynomial. For some other polynomial commitment schemes, such as Ligero [AHIV17], the preprocessing time would be superlinear, but the proving time would be linear. For yet another family, including KZG, PST, FRI, STIR, and WHIR [KZG10; PST13; BBHR18a; ACFY24; ACFY25], the preprocessing and proving times would both be superlinear. We compare the PCS schemes and their compatibility with BiPerm in Table 2.

MulPerm, an almost-linear permutation check for any PCS Given the limitations of BiPerm, we design MulPerm. MulPerm works with any PCS, i.e. only requires committing to and opening linear sized polynomials, but it has slightly worse prover time $n \cdot \tilde{O}(\sqrt{\log n})$. The idea is to arithmetize $\mathbb{1}_\sigma$ into ℓ parts, $\mathsf{p}_1, \ldots, \mathsf{p}_\ell$. The prover can then use a sumcheck to prove that the degree ℓ sumcheck statement

$$\sum_{x \in \{0,1\}^\mu} f(x) \prod_{i=1}^{\ell} \mathsf{p}_i \left(x, y \left[i \cdot \frac{\mu}{\ell} : (i+1) \cdot \frac{\mu}{\ell} \right] \right) = g(y) \ \ \forall y \in \{0,1\}^\mu$$

Proving this using a sumcheck argument reduces to evaluating $\mathsf{p}_1 \ldots, \mathsf{p}_\ell$ at a random point $\alpha \in \mathbb{F}^\mu, r \in \mathbb{F}^{\mu/\ell}$. We then use a second sumcheck to prove the evaluation of each p_i using precommitted $\tilde{\mathbb{1}}_{[\sigma_i]} := \mathsf{eq}(\sigma_i(x), y_i)$, which map from $\mathbb{F}^{\mu+1}$ to $\mathbb{F}$, i.e., are linear in size. Each p_i can, thus, be written as

$$\mathsf{p}_i(x, y_i) = \prod_{j=1}^{\mu/\ell} \mathsf{eq}(\sigma_{i\cdot\ell+j}(x), y_{i\cdot\ell+j})$$

Our key insight is that $\mathsf{eq}(\sigma_i(x), y_i)$ maps to $\{0,1\}$ for any $(x, y_i) \in \{0,1\}^{\mu+1}$. In the sumcheck the prover needs to compute $r_i(X_1) = \mathsf{eq}(\sigma_i(X_1, x'), y_i)$ for a formal variable X_1 and $(x', y_i) \in \{0,1\}^{\mu}$. Importantly, $r_i(X_1)$ can only take up one of 4 possible functions $X_1, (1-X_1), 1$, or 0. Thus, $r(X_1) = \mathsf{p}_i(X_1, x', y_i)$ can only be one of $4^{\mu/\ell}$ possible functions. For $\ell > 2$, $4^{\mu/\ell} = n^{1-\epsilon} = o(n)$. It turns out that in the kth sumcheck round, the number of functions the round polynomial can take up is $2^{2^k \cdot \mu/\ell}$, i.e., doubly exponential in k. On the other hand, the size of the boolean hypercube over which we sum is cut in half in each sumcheck round. We find that by setting $\ell = O(\sqrt{\mu})$, the complexity of the two parts of the protocols is balanced out and the overall prover time is $n \cdot \tilde{O}(\sqrt{\mu})$ field operations (no additional commitments). The verifier time of MulPerm is $O(\mu)$, and the soundness error is $O(\mu^2/\mathbb{F})$. We give an illustration of our Protocols in Figure 1.

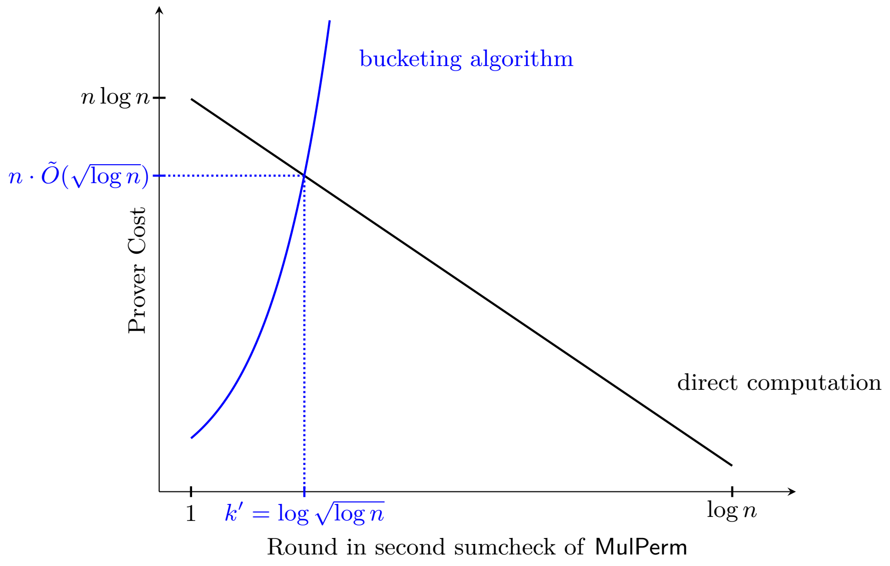

Fig. 2. For each round of the second sumcheck (Algorithm 7) in MulPerm: the black line shows the prover cost if directly computing the prover message, and the blue line shows the prover cost if using the bucketing algorithm (Algorithm 6). The idea is to use the bucketing trick up to round k' and then switch over to direct computation in order to minimize the overall prover cost.

Lookup arguments We also apply our techniques to lookup arguments. In lookup arguments, the permutation is replaced with some arbitrary map σ between g, defined over B_μ and a table t, of size T defined over B_κ, such that

$$g(\boldsymbol{x}) = \sum_{\boldsymbol{y} \in B_\kappa} t(\boldsymbol{y}) \mathbb{1}_\sigma(\boldsymbol{x}, \boldsymbol{y}) \forall \boldsymbol{x} \in B_\mu. \tag{1}$$

Here $\mathbb{1}_\sigma(\boldsymbol{x}, \boldsymbol{y})$ is 1 if $\sigma(\boldsymbol{x}) = \boldsymbol{y}$, i.e. $g(\boldsymbol{x}) = t(\boldsymbol{y})$ for $\boldsymbol{x} \in B_\mu$ and $\boldsymbol{y} \in B_\kappa$. While this has a very similar structure to the permutation check, there are a number of challenges. Firstly, (1) is neither multilinear in $\boldsymbol{x}$ nor in $\boldsymbol{y}$. This means we first need to run a sumcheck over x and y to show that

$$\sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) g(\boldsymbol{x}) = \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \sum_{\boldsymbol{y} \in B_\kappa} t(\boldsymbol{y}) \mathbb{1}_\sigma(\boldsymbol{x}, \boldsymbol{y})$$

for a random vector $\boldsymbol{s}$.

Secondly, for large tables, i.e. for $\kappa > \mu$, a naive implementation of the sumcheck takes $2^\kappa = T$ time. However, we note that $\mathbb{1}_\sigma(y, x)$ is incredibly sparse and has at most n non-zero entries. We use this to reduce the prover time of the outer sumcheck to $n \cdot (\kappa - \mu)$. Finally, $\mathbb{1}_\sigma(y, x)$ can now be arithmetized as a product of κ eqs. We reparameterize our protocol to achieve a prover time of $n\tilde{O}(\sqrt{\kappa})$ for the inner sumcheck. Our protocol also requires one evaluation of t. For some tables, e.g. range proofs and all the RISC-V instructions described in Jolt [AST24], this can be highly efficient.

Prover provided-permutation and lookup checks Up till now, we assumed that the permutation or lookup maps are fixed and preprocessed. In many application, this is not the case but rather the prover computes and commits to the map during runtime. This presents a new challenge: What if the map is not well-formed, e.g. in the lookup case what if it maps from B_μ to some arbitrary image in $\mathbb{F}^\kappa$. To prevent this, we have the prover prove that σ is well-formed. For this, the prover proves that each output of σ is either 0 or 1. For a lookup that maps from B_μ to B_μ, this can be done in $O(n \log \log n)$ time, which is less than the cost in the rest of the argument. In the permutation case, this check is insufficient. We need to prove that σ is indeed a valid permutation. To do this we have the prover commit to the inverse of σ, τ and prove that $\sigma(\tau(x)) = x \;\; \forall x \in B_\mu$. This shows that σ has an inverse on B_μ, i.e. is bijective.

3 Applications to SNARKs

The main application of our permutation and lookup arguments is reducing the prover and verifier costs in popular SNARK systems. In particular, we show two instances of SNARK systems that, when using our compiler, only require the prover to commit to the witness and no additional $\Theta(n)$-sized messages. This can significantly improve the prover's efficiency if the witness is sparse or of low weight (compared to the size of the field). Additionally, our solutions do not require the use of GKR and can be instantiated with any polynomial commitment scheme.

3.1 Improving HyperPlonk

By replacing the permutation check and the lookup check in HyperPlonk with our algorithms, we achieve a SNARK where the prover cost is dominated by a single commitment to the witness and runs two sumcheck arguments. The proof consists of two sumcheck arguments that can be run in linear or nearly-linear time, depending on whether BiPerm or $\mathsf{MulPerm}$ is used. Previously, HyperPlonk's permutation argument required committing to a polynomial of size n with full field elements (of λ-bit width), even if the witness is sparse or small. Similarly, our lookup argument can be used to replace the lookup argument in HyperPlonk+. Instead of having to commit to $n+T$ full field elements, for looking up n values in a table of size T, our protocol requires committing to only n elements of with $\log T$ bit width.

	Commitment ops	Field Ops	Verifier time	PCS
Quarks-style (HP 3.5)	$2n$ $\mathbb{F}$ elements	n	$\log(n)$	Any
HyperPlonk sumcheck (HP 3.6)	$\lvert w\rvert$	$n \cdot \tilde{O}(\log(n))$	$\log(n)$	Any
With $\mathsf{MulPerm}$	$\lvert w\rvert$	$n \cdot \tilde{O}(\sqrt{\log(n)})$	$\log(n)$	Any
With BiPerm	$\lvert w\rvert$	n	$2\log(n)$	Sparse

Table 5. HyperPlonk with different permutation arguments. $|w|$ refers to the total weight of the witness and n to its length.

3.2 Improving Spartan

In Spartan [Set20], $\tilde{A}, \tilde{B}, \tilde{C}$ are the MLEs of matrices $A, B, C \in \mathbb{F}^{m\times m}$ in an R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$. Let $s = \log m, \mu = \log n$ and $Z = (io, 1, w)$. Note that n is the number of nonzero entries in a matrix.[5] The verifier needs evaluations of $\tilde{A}, \tilde{B}, \tilde{C}$ at random points r_x, r_y. However, evaluating these polynomials is too expensive due to its quadratic input size. To get around this, Spartan uses SPARK which uses memory-checking and GKR to delegate the evaluations to the prover, who only needs to commit to the witness and performs linear number of field operations. Here, we describe a solution that improves SPARK, obviating the need for memory-checking, which induces a large soundness error and larger verifier cost.

We adopt the way of encoding sparse polynomial $\tilde{M} \in \{\tilde{A}, \tilde{B}, \tilde{C}\}$ in SPARK. $\tilde{M}$ is represented by three preprocessed polynomials $val(), row(), col() : B_\mu \mapsto B_s$, such that when given input $\boldsymbol{j} \in B_\mu$, outputs the value, row index, and column index of the jth nonzero element in the matrix, respectively. Then, for

[5] Note that the matrices don't necessarily need to be square. For simplicity, we will only deal with square R1CS languages, which can always be achieved through padding.

every $\boldsymbol{x}, \boldsymbol{y} \in B_s$,

$$\tilde{M}(\boldsymbol{x}, \boldsymbol{y}) := \sum_{\boldsymbol{j} \in B_\mu} \tilde{val}(\boldsymbol{j}) \cdot \tilde{\mathbb{1}}_{row}(\boldsymbol{j}, \boldsymbol{x}) \cdot \tilde{\mathbb{1}}_{col}(\boldsymbol{j}, \boldsymbol{y})$$

When the verifier asks for the evaluation of $\tilde{M}(\boldsymbol{r}_x, \boldsymbol{r}_y)$, the prover simply proves the following via a sumcheck

$$\tilde{M}(\boldsymbol{r}_x, \boldsymbol{r}_y) := \sum_{\boldsymbol{j} \in B_\mu} \tilde{val}(\boldsymbol{j}) \cdot \tilde{\mathbb{1}}_{row}(\boldsymbol{j}, \boldsymbol{r}_x) \cdot \tilde{\mathbb{1}}_{col}(\boldsymbol{j}, \boldsymbol{r}_y)$$

We observe that this sumcheck has the structure of a preprocessed lookup argument, as row, col are mappings between two boolean hypercubes of size n and m respectively. Thus, MulPerm for preprocessed lookup can be used to prove this statement. The prover only needs to commit to the witness, and run two sumcheck arugments which requires $n \cdot \tilde{O}(\sqrt{\log m})$ field operation. There is no need to use GKR-based memory checking, which significantly reduces the soundness error, proof size, and verifier time to only $O(\log(n))$.

In the final step of this sumcheck, verifier needs evaluate $\tilde{val}, \tilde{row}, \tilde{col}$ at a random point, which will be provided by the prover as PCS openings.

3.3 R1CS-style GKR

Definition 1. *A L-layered R1CS GKR circuit is a deterministic circuit with L layers. Layer* 1 *holds the inputs to the circuit and Layer L holds the outputs of the circuit. In Layer i for every* $i \in [L-1]$, $\boldsymbol{z}_i$ *are the inputs to Layer i and* $\boldsymbol{z}_{i+1}$ *are the outputs of Layer i, such that* $A^{(i)}\boldsymbol{z}_i \circ B^{(i)}\boldsymbol{z}_i = \boldsymbol{z}_{i+1}$ *where matrices* $A^{(i)}, B^{(i)} \in \mathbb{F}^{|\boldsymbol{z}_{i+1}| \times |\boldsymbol{z}_i|}$. *Let* $\mu_i = \log |\boldsymbol{z}_i|$.

The indexed relation $\mathcal{R}_{\mathsf{R1CSGKR}}$ *is the set of tuples*

$$(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\tilde{A}^{(i)}, \tilde{B}^{(i)}, [[\tilde{A}^{(i)}]], [[\tilde{B}^{(i)}]]; [[\tilde{z}_i]]; \tilde{z}_i \,|i \in [L])$$

where $\tilde{A}^{(i)}, \tilde{B}^{(i)}, \tilde{z}_i$ *are the MLEs of the polynomials* $A^{(i)}, B^{(i)} : B_{\mu_{i+1}} \times B_{\mu_i} \mapsto \{0,1\}$ *and* $z_i : B_{\mu_i} \mapsto \{0,1\}$ *that interpret the matrices and vectors in every layer* $i \in [L]$*, and*

$$\Big(\sum_{\boldsymbol{y} \in B_{\mu_i}} \tilde{A}^{(i)}(\boldsymbol{x}, \boldsymbol{y}) \cdot \tilde{z}_i(\boldsymbol{y})\Big)\Big(\sum_{\boldsymbol{y} \in B_{\mu_i}} \tilde{B}^{(i)}(\boldsymbol{x}, \boldsymbol{y}) \cdot \tilde{z}_i(\boldsymbol{y})\Big) = \tilde{z}_{i+1}(\boldsymbol{x}) \;\; \forall \boldsymbol{x} \in B_{\mu_{i+1}}$$

The task of proving the equation above for every $x \in B_{\mu_{i+1}}$ can be reduced to a single sumcheck using random challenge $\boldsymbol{r} \in \mathbb{F}^{\mu_i}$

$$\sum_{\boldsymbol{x} \in B_{\mu_{i+1}}} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{r}) \cdot \Big(\sum_{\boldsymbol{y} \in B_{\mu_i}} \tilde{A}^{(i)}(\boldsymbol{x}, \boldsymbol{y}) \cdot \tilde{z}_i(\boldsymbol{y})\Big)\Big(\sum_{\boldsymbol{y} \in B_{\mu_i}} \tilde{B}^{(i)}(\boldsymbol{x}, \boldsymbol{y}) \cdot \tilde{z}_i(\boldsymbol{y})\Big) = \tilde{z}_{i+1}(\boldsymbol{r}) \tag{2}$$

Again, we adopt the way of encoding sparse polynomial $\tilde{M} \in \{\tilde{A}^{(i)}, \tilde{B}^{(i)} \,|i \in [L]\}$ in SPARK. Let the number of nonzero entries in $\tilde{M}$ be $q = 2^s$. $\tilde{M}$ is represented by three preprocessed polynomials $val(), row(), col()$ that maps from B_s

to their corresponding boolean hypercube; when given input $\boldsymbol{j} \in B_s$, $val(), row(), col()$ output the value, row index, and column index of the jth nonzero element in the matrix, respectively. Then, for every $\boldsymbol{x} \in B_{\mu_{i+1}}, \boldsymbol{y} \in B_{\mu_i}$,

$$\tilde{M}(\boldsymbol{x}, \boldsymbol{y}) := \sum_{\boldsymbol{j} \in B_s} \tilde{val}(\boldsymbol{j}) \cdot \tilde{\mathbb{1}}_{row}(\boldsymbol{j}, \boldsymbol{x}) \cdot \tilde{\mathbb{1}}_{col}(\boldsymbol{j}, \boldsymbol{y})$$

Given preprocessed $\tilde{A}^{(i)}, \tilde{B}^{(i)}$ and polynomials $\tilde{z}_i$ for every $i \in [L]$, we now describe a Spartan-like GKR algorithm that uses our permutation argument to prove $(\tilde{A}^{(i)}, \tilde{B}^{(i)}, [[\tilde{A}^{(i)}]], [[\tilde{B}^{(i)}]]; [[\tilde{z}_i]]; \tilde{z}_i \mid i \in [L]) \in \mathcal{R}_{\mathsf{R1CSGKR}}$. Run sumcheck for Equation 2 for every layer. After each sumcheck of Layer $i \in [L-1]$, verifier is left with prover claimed evaluations $A^{(i)}(\boldsymbol{r}_x, \boldsymbol{r}_y), B^{(i)}(\boldsymbol{r}_x, \boldsymbol{r}_y)$, where $\boldsymbol{r}_x \in B_{\mu_{i+1}}, \boldsymbol{r}_y \in B_{\mu_i}$ are challenges. The verifier can check these evaluations via the following sumcheck,

$$\tilde{M}(\boldsymbol{r}_x, \boldsymbol{r}_y) := \sum_{\boldsymbol{j} \in B_s} \tilde{val}(\boldsymbol{j}) \cdot \tilde{\mathbb{1}}_{row}(\boldsymbol{j}, \boldsymbol{r}_x) \cdot \tilde{\mathbb{1}}_{col}(\boldsymbol{j}, \boldsymbol{r}_y)$$

for every $i \in [L-1]$, $M \in \{A^{(i)}, B^{(i)}\}$ and its corresponding $val(), row(), col()$ Similar to our observation in Spartan, we observe that this sumcheck has the structure of a preprocessed lookup argument, since val, row, col all map between boolean hypercubes. Let $n = \max(2^{\mu_i}, 2^{\mu_{i+1}})$.

Using our algorithm for preprocessed lookup achieves almost linear prover cost of $q \cdot \tilde{O}(\sqrt{\log n})$ field operations. It eliminates the need to use GKR or to commit to anything besides the witness. When the verifier needs to evaluate $\tilde{val}, \tilde{row}, \tilde{col}$ at a random point in the end of the sumcheck, these evaluations will be provided by the prover as PCS openings. We note that all $2L$ claims of $\tilde{M}^{(i)}(\boldsymbol{r}_x, \boldsymbol{r}_y)$ can be batched into one sumcheck using random linear combination instead of being proved individually.

Example: Inner product of $\boldsymbol{w}$ with constants $\boldsymbol{c}$. Suppose $\boldsymbol{w} \in H^n$ are the inputs, and $\boldsymbol{c} \in H^n$ is a vector of constants. We want to prove their inner product $\langle \boldsymbol{w}, \boldsymbol{c} \rangle = \sum_{i=1}^{n} \boldsymbol{c}_i \boldsymbol{w}_i$ is computed correctly. Let $\mu := \log n$.

If we were to use the standard GKR, the circuit will have depth $\log n$ since each addition gate only has two incoming-wires. However, if we use the R1CS-style GKR, then this computation can be modeled using a single layer, where $z_{in} = [1, \boldsymbol{w}]$, $A = [0, \boldsymbol{c}]^T$ and $B = [1, 0^n]^T$ The prover does not need to commit to anything other than the inputs $\boldsymbol{w}$.

4 Preliminaries

Notation: We use λ to denote the security parameter.
When dealing with permutations, we use n to denote the number of elements in the permutation and define $\mu := \log n$. We use $B_\mu := \{0,1\}^\mu \subseteq \mathbb{F}^\mu$ to denote the μ-dimensional boolean hypercube.
We use $\mathcal{F}_\mu^{(\leq d)}$ to mean the set of μ-variate, at most degree d polynomial. We use $\mathsf{PERM}(H)$ to denote the set of permutation functions that map between H, and

$\mathsf{FUN}(H, H')$ to denote the set of functions that map from H to H'.
All indices in this paper start from 1. For $s \in \mathbb{N}$, $[s]$ denotes the set $\{1, 2, \ldots, s\}$.
For $a, b \in \mathbb{N}, a \leq b$ and vector $\boldsymbol{x} \in \mathbb{F}^{(\geq b)}$, $\boldsymbol{x}_{[a:b]}$ denotes the vector slice $[\boldsymbol{x}_a, \boldsymbol{x}_{a+1}, \ldots, \boldsymbol{x}_b]$; $\boldsymbol{x}_{[:a]} := \boldsymbol{x}_{[1:a]}$, and $\boldsymbol{x}_{[a:]} = \boldsymbol{x}_{[a:|\boldsymbol{x}|]}$.
We use $\tilde{O}$ notation to hide polylogarithmic factors in asympototic complexity. $a(n) = \tilde{O}(b(n))$ is equivalent to $a(n) = O(b(n) \log^k b(n))$ for any constant k.
In our protocols, we assume the mapping σ is preprocessed and included in the index. See Section 8 for discussion on the case when σ is not preprocessed.

The following facts are used throughout the paper.

Definition 2 (Equality polynomial). *We define equality polynomial* $\mathsf{eq} : \mathbb{F}^\mu \times \mathbb{F}^\mu \mapsto \mathbb{F}$ *for arbitrary* $\mu \in \mathbb{N}$ *as*

$$\mathsf{eq}(\boldsymbol{X}, \boldsymbol{Y}) := \prod_{i=1}^{\mu} \big(\boldsymbol{X}_i \boldsymbol{Y}_i + (1 - \boldsymbol{X}_i)(1 - \boldsymbol{Y}_i)\big)$$

Note that for any $\boldsymbol{x}, \boldsymbol{y} \in B_\mu$, $\mathsf{eq}(\boldsymbol{x}, \boldsymbol{y}) \in \{0, 1\}$. *Specifically,* $\mathsf{eq}(\boldsymbol{x}, \boldsymbol{y}) = 1$ *if* $\boldsymbol{x}_i = \boldsymbol{y}_i$ *for every* $i \in [\mu]$, *and* $\mathsf{eq}(\boldsymbol{x}, \boldsymbol{y}) = 0$ *otherwise.*

Lemma 1 (Multilinear extensions). *For every function* $f : B_\mu \mapsto \mathbb{F}$, *there is a unique multilinear polynomial* $\tilde{f} \in \mathbb{F}[X_1, \ldots, X_\mu]$ *such that* $\tilde{f}(\boldsymbol{b}) = f(\boldsymbol{b})$ *for all* $\boldsymbol{b} \in B_\mu$. *We call* $\tilde{f}$ *the multilinear extension of* f, *and* $\tilde{f}$ *can be expressed as*

$$\tilde{f}(\boldsymbol{X}) = \sum_{\boldsymbol{b} \in B_\mu} f(\boldsymbol{b}) \cdot \mathsf{eq}(\boldsymbol{b}, \boldsymbol{X})$$

Lemma 2 (Schwartz-Zippel Lemma). *Let* $f : \mathbb{F}^\mu \mapsto \mathbb{F}$ *be a non-zero polynomial of total degree* d. *Let* S *be any finite subset of* $\mathbb{F}$, *and let* $r_1, \ldots, r_\mu$ *be* μ *field elements selected independently and uniformly from set* S. *Then*

$$\Pr[f(r_1, \ldots, r_\mu) = 0] \leq \frac{d}{|S|}$$

4.1 Interactive proofs and arguments of knowledge

We adapt the definitions for interactive proofs of knowledge from HyperPlonk [CBBZ23].

Definition 3 (Interactive Proof and Argument of Knowledge). *An interactive protocol* $\Pi = (\mathsf{Setup}, \mathcal{I}, \mathsf{P}, \mathsf{V})$ *between a prover* P *and verifier* V *is an argument of knowledge for an indexed relation* $\mathcal{R}$ *with knowledge error* $\delta : \mathbb{N} \to [0, 1]$ *if the following properties hold, where given an index* $\mathbb{i}$, *common input* $\mathbb{x}$ *and prover witness* $\mathbb{w}$, *the deterministic indexer outputs* $(\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathbb{i})$ *and the output of the verifier is denoted by the random variable* $\langle \mathsf{P}(\mathsf{pp}, \mathbb{x}, \mathbb{w}), \mathsf{V}(\mathsf{vp}, \mathbb{x}) \rangle$*:*

– *Perfect Completeness: for all* $(\mathbb{i}; \mathbb{x}; \mathbb{w}) \in \mathcal{R}$

$$\Pr\left[\langle \mathsf{P}(\mathsf{pp}, \mathbb{x}, \mathbb{w}), \mathsf{V}(\mathsf{vp}, \mathbb{x}) \rangle = 1 \,\middle|\, \begin{array}{c} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathsf{gp}, \mathbb{i}) \end{array} \right] = 1$$

– *δ-Soundness (adaptive): Let $\mathcal{L}(\mathcal{R})$ be the language corresponding to the indexed relation $\mathcal{R}$ such that $(\mathbb{i}, \mathbb{x}) \in \mathcal{L}(\mathcal{R})$ if and only if there exists $\mathbb{w}$ such that $(\mathbb{i}; \mathbb{x}; \mathbb{w}) \in \mathcal{R}$. Π is δ-sound if for every pair of probabilistic polynomial time adversarial prover algorithm $(\mathcal{A}_1, \mathcal{A}_2)$ the following holds:*

$$\Pr\left[\langle \mathcal{A}_2(\mathbb{i}, \mathbb{x}, \mathsf{st}), \mathsf{V}(\mathsf{vp}, \mathbb{x})\rangle = 1 \wedge (\mathbb{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R}) \left| \begin{array}{c} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathbb{i}, \mathbb{x}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{gp}) \\ (\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathsf{gp}, \mathbb{i}) \end{array}\right.\right] \leq \delta(|\mathbb{i}|+|\mathbb{x}|)\,.$$

We say a protocol is computationally sound if δ is negligible. If $\mathcal{A}_1, \mathcal{A}_2$ are unbounded and δ is negligible, then the protocol is statistically sound. If $A = (\mathcal{A}_1, \mathcal{A}_2)$ is unbounded, the soundness definition becomes for all $(\mathbb{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R})$

$$\Pr\left[\langle \mathcal{A}_2(\mathbb{i}, \mathbb{x}, \mathsf{gp}), \mathsf{V}(\mathsf{vp}, \mathbb{x})\rangle = 1 \left| \begin{array}{c} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathsf{gp}, \mathbb{i}) \end{array}\right.\right] \leq \delta(|\mathbb{i}| + |\mathbb{x}|)$$

– *δ-Knowledge Soundness: There exists a polynomial $\mathsf{poly}(\cdot)$ and a probabilistic polynomial-time oracle machine $\mathcal{E}$ called the* extractor *such that given oracle access to any pair of probabilistic polynomial time adversarial prover algorithm $(\mathcal{A}_1, \mathcal{A}_2)$ the following holds:*

$$\Pr\left[\begin{array}{c} \langle \mathcal{A}_2(\mathbb{i}, \mathbb{x}, \mathsf{st}), \mathsf{V}(\mathsf{vp}, \mathbb{x})\rangle = 1 \\ \wedge \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} \end{array} \left| \begin{array}{c} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathbb{i}, \mathbb{x}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{gp}) \\ (\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathsf{gp}, \mathbb{i}) \\ \mathbb{w} \leftarrow \mathsf{Ext}^{\mathcal{A}_1, \mathcal{A}_2}(\mathsf{gp}, \mathbb{i}, \mathbb{x}) \end{array}\right.\right] \leq \delta(|\mathbb{i}| + |\mathbb{x}|)$$

An interactive protocol is "knowledge sound", or simply an "argument of knowledge", if the knowledge error δ is negligible in λ. If the adversary is unbounded, then the argument is called an interactive proof of knowledge.

– *Public coin An interactive protocol is considered to be public coin if all of the verifier messages (including the final output) can be computed as a deterministic function given a random public input.*

Non-interactive arguments. Interactive public-coin arguments can be made non-interactive using the Fiat-Shamir transform. The Fiat-Shamir transform replaces the verifier challenges with hashes of the transcript up to that point. The works by [AFK21; Wik21] show that this is secure for multi-round special-sound protocols and multi-round oracle proofs. We adapt the definitions for PIOP from HyperPlonk [CBBZ23]. See the definition for interactive proofs of knowledge in Definition 3.

Definition 4. *A polynomial interactive oracle proof (PIOP) is a public-coin interactive proof for a polynomial oracle relation $\mathcal{R} = \{(\mathbb{i}; \mathbb{x}; \mathbb{w})\}$. The relation is an oracle relation in that the index $\mathbb{i}$ and the instance $\mathbb{x}$ can contain oracles to multi-variate polynomials over some field $\mathbb{F}$. The oracles specify μ and the degree in each variable. These oracles can be queried at arbitrary points in $\mathbb{F}^\mu$ to evaluate the polynomial at these points. The actual polynomials corresponding to* *the oracles are contained in the* pp *and the* $\mathbb{w}$*, respectively. We denote an oracle to a polynomial* f *by* $[[f]]$*. In every protocol message, the* P *sends multi-variate polynomial oracles. The verifier in every round sends a random challenge.*

We measure the following parameters for the complexity of a PIOP:

– *The size of the proof oracles is the length of the transmitted polynomials.*
– *The round complexity measures the number of rounds. In our protocols, it is always equivalent to the number of oracles sent.*
– *The size of the witness is the length of the witness polynomial.*
– *The prover time measures the runtime of the prover. In this work, we only care about the number of field operations performed by the prover, specifically the number of field additions and multiplications.*
– *The verifier time measures the runtime of the verifier.*
– *The query complexity is the number of queries the verifier performs to the oracles.*
– *The sparsity of the oracles is the number of non-zero entries in the evaluation tables of the committed polynomials.*

Proof of Knowledge. As a proof system, the PIOP satisfies perfect completeness and unbounded knowledge-soundness with knowledge-error δ. Note that the extractor can query the oracle at arbitrary points to efficiently recover the entire polynomial.

Soundness and knowledge soundness. HyperPlonk proved that soundness directly implies knowledge soundness for certain oracle relations and oracle arguments [CBBZ23].

Lemma 3 (Lemma 2.3 of [CBBZ23]). *: Sound PIOPs are knowledge sound] Consider a* δ*-sound PIOP for oracle relations* $\mathcal{R}$ *such that for all* $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$*,* $\mathbb{w}$ *consists only of polynomials such that the instance contains oracles to these polynomials. The PIOP has* δ *knowledge-soundness, and the extractor runs in time* $O(|\mathbb{w}|)$*.*

4.2 PIOP compilation

PIOP compilation transforms PIOP into an interactive argument of knowledge Π by replacing the oracles with polynomial commitments. Every query by the verifier is replaced with an invocation of the Eval protocol at the query point. The compiled verifier accepts if the PIOP verifier accepts and if the output of all Eval invocations is 1. If Π is public-coin, then it can be further compiled to a Non-interactive ARgument of Knowledge (NARK) through Fiat-Shamir transform.

Theorem 1 (PIOP Compilation). *[BFS20; CHMMVW20] 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 sum of (i) prover time of the PIOP, (ii) the oracle length times the commitment time, and (iii) the query complexity times the prover time of Γ.*
- *Verifier time The verifier time is equal to the sum of (i) the verifier time of the PIOP and (ii) the verifier time for Γ times the query complexity of the PIOP.*
- *Proof size The proof size is equal to sum of (i) the message complexity of the PIOP times the commitment size and (ii) the query complexity times the proof size of Γ. If the proof size is $O(\log^c(|\mathbb{w}|))$, then we say the proof is succinct.*

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

4.3 Sumcheck protocol

Definition 5 (SumCheck relation). *The relation $\mathcal{R}_{\mathsf{SUM}}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = \big((v, [[f]]); f\big)$ where $f \in \mathcal{F}_{\mu}^{(\leq d)}$ and $\sum_{\boldsymbol{b} \in \{0,1\}^{\mu}} f(\boldsymbol{b}) = v$.*

Theorem 2. *The PIOP for $\mathcal{R}_{\mathsf{SUM}}$ is perfectly complete and has knowledge error $\delta_{\mathsf{SUM}}^{d,\mu} := d\mu/|\mathbb{F}|$.*

The classic SumCheck protocol [LFKN90] is a PIOP for $\mathcal{R}_{\mathsf{SUM}}$. Given a tuple $(\mathbb{x}; \mathbb{w}) = (v, [[f]]; f)$ for μ-variate degree d polynomial f such that $\sum_{\boldsymbol{x} \in \{0,1\}^{\mu}} f(\boldsymbol{b}) = S$:

- For $k = 1, \ldots, \mu$:
  - The prover computes $u_k(X) := \sum_{\boldsymbol{x} \in \{0,1\}^{\mu-k}} f(\boldsymbol{\alpha}, X, \boldsymbol{x})$, where $\boldsymbol{\alpha} := [\alpha_i]_{i=1}^{k-1}$ are challenges. Prover sends the oracle $[[u_k]]$ to the verifier. u_k is univariate and of degree at most d.
  - The verifier checks that $S = u_k(0) + u_k(1)$, samples $\alpha_k \leftarrow \mathbb{F}$, sends α_k to the prover, and sets $S \leftarrow u_k(\alpha_k)$.
- Finally, the verifier accepts if $f(\boldsymbol{\alpha}) = S$.

We refer to [Tha20] for the proof of Theorem 2.

Prover messages sent as oracles. Like in HyperPlonk [CBBZ23], our prover will send an oracle of u_k in each round, instead of the actual polynomial. This reduces the communication and verifier complexity without changing the soundness analysis. This is useful especially if the degree of u is large, as in our PIOPs. Moreover, the verifier has to evaluate every u_k at three points: 0, 1, and α_k. Section 3.1 of [CBBZ23] introduces a useful optimization where the prover instead

sends an oracle for a degree $d-2$ polynomial u'_k and an extra field element $u_k(0)$; the verifier can then obtain the three evaluations through only one query to the oracle of u'_k.
In the complexity analyses throughout this paper, we assume the above algorithms are applied explicitly.

4.4 Multilinear polynomial commitments.

Definition 6 (Commitment scheme). *A commitment scheme Γ is a tuple $\Gamma = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ of PPT algorithms where:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{gp}$ *generates public parameters* gp*;*
- $\mathsf{Commit}(\mathsf{gp}; x) \to (C; r)$ *takes a secret message x and outputs a public commitment C and (optionally) a secret opening hint r (which might or might not be the randomness used in the computation).*
- $\mathsf{Open}(\mathsf{gp}, C, x, r) \to b \in \{0,1\}$ *verifies the opening of commitment C to the message x provided with the opening hint r.*
  A commitment scheme Γ is binding *if for all PPT adversaries $\mathcal{A}$:*

$$\Pr\left[b_0 = b_1 \neq 0 \wedge x_0 \neq x_1 \;:\; \begin{array}{l} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (C, x_0, x_1, r_0, r_1) \leftarrow \mathcal{A}(\mathsf{gp}) \\ b_0 \leftarrow \mathsf{Open}(\mathsf{gp}, C, x_0, r_0) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{gp}, C, x_1, r_1) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

A commitment scheme Γ is hiding *if for any polynomial-time adversary $\mathcal{A}$:*

$$\left| \Pr\left[b = b' \;:\; \begin{array}{l} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (x_0, x_1, st) \leftarrow \mathcal{A}(\mathsf{gp}) \\ b \leftarrow\!\!\$\, \{0,1\} \\ (C_b; r_b) \leftarrow \mathsf{Commit}(\mathsf{gp}; x_b) \\ b' \leftarrow \mathcal{A}(\mathsf{gp}, st, C_b) \end{array} \right] - 1/2 \right| = \mathsf{negl}(\lambda)\,.$$

If the adversary is unbounded, then we say the commitment is statistically hiding. We additionally define polynomial commitment schemes for multi-variate polynomials.

Definition 7. *(Polynomial commitment) A polynomial commitment scheme is a tuple of protocols $\Gamma = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ where $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ is a binding commitment scheme for a message space $R[X]$ of polynomials over some ring R, and*

- $\mathsf{Eval}((\mathsf{vp}, \mathsf{pp}), C, \boldsymbol{z}, y, d, \mu; f) \to b \in \{0,1\}$ *is an interactive public-coin protocol between a PPT prover* P *and verifier* V*. Both* P *and* V *have as input a commitment C, points $\boldsymbol{z} \in \mathbb{F}^\mu$ and $y \in \mathbb{F}$, and a degree d. The prover has prover parameters* pp*, and the verifier has verifier parameters* vp*. The prover additionally knows the opening of C to a secret polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$. The protocol convinces the verifier that $f(\boldsymbol{z}) = y$.*

A polynomial commitment scheme is correct *if an honest committer can successfully convince the verifier of any evaluation. Specifically, if the prover is honest, then for all polynomials $f \in \mathcal{F}_\mu^{(\leq d)}$ and all points $z \in \mathbb{F}^\mu$,*

$$\Pr\left[b = 1 \ : \ \begin{array}{l} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (C; r) \leftarrow \mathsf{Commit}(\mathsf{gp}, f) \\ y \leftarrow f(\boldsymbol{z}) \\ b \leftarrow \mathsf{Eval}(\mathsf{gp}, c, \boldsymbol{z}, y, d, \mu; f, r) \end{array}\right] = 1 \ .$$

We require that Eval *is an interactive argument of knowledge and has* knowledge soundness*, which ensures that we can extract the committed polynomial from any evaluation.*

Batch evaluation of polynomial commitments While our PIOPs can be compiled with arbitrary PCS schemes, in the concrete efficiency analysis, we assume that the PCS has efficient batch evaluation. That is we can efficiently evaluate t polynomials that are committed together at the same point $x^* \in \mathbb{F}^\mu$. Let $y_1, \ldots, y_t$ be the claimed evaluation.For homomorphic commitments, we show that the random linear combination $\sum_{i=1}^t \alpha^i f_i(X)$, at x^* is equal to $\sum_{i=1}^t \alpha^i \cdot y_i$, for a challenge $\alpha \in \mathbb{F}$. The prover's cryptographic operations for size n polynomials are proportional to $n + t$, not $n \cdot t$. For hash-based proof systems, we can commit to all t polynomials using an interleaved code and give an evaluation protocol for the interleaved code, such that the cryptographic operations are independent of t.

4.5 Permutation and its inverse

Definition 8 (Multilinear versions of permutation and its inverse). *Given a permutation $\sigma : B_\mu \mapsto B_\mu$, we can write it as a multilinear map [CBBZ23]:*

$$\tilde{\sigma}(\boldsymbol{X}) = (\tilde{\sigma}_1(\boldsymbol{X}), \ldots, \tilde{\sigma}_\mu(\boldsymbol{X})) : \mathbb{F}^\mu \mapsto \mathbb{F}^\mu.$$

For all $i \in [\mu]$, $\tilde{\sigma}_i(\cdot)$ is the multilinear extension of the ith bit of $\sigma(\cdot)$. Note that, by definition, $\tilde{\sigma}_i(\boldsymbol{y}) \in \{0, 1\}$ for all $\boldsymbol{y} \in B_\mu$ and $i \in [\mu]$. Similarly, the multilinear map representing the inverse permutation $\sigma^{-1} : B_\mu \mapsto B_\mu$ is defined as

$$\tilde{\sigma}^{-1}(\boldsymbol{X}) = (\tilde{\sigma}_1^{-1}(\boldsymbol{X}), \ldots, \tilde{\sigma}_\mu^{-1}(\boldsymbol{X})) : \mathbb{F}^\mu \mapsto \mathbb{F}^\mu.$$

such that for all $i \in [\mu]$, $\tilde{\sigma}_i^{-1}(\cdot)$ is the multilinear extension of the ith bit of $\sigma^{-1}(\cdot)$. These multilinear maps can be interpolated, treating the subscripts *as explicit binary* inputs*, producing multilinear functions $\tilde{\sigma}_{[\mu]} : B_{\mu+\log\mu} \mapsto B$ and $\tilde{\sigma}_{[\mu]}^{-1} : B_{\mu+\log\mu} \mapsto B$, such that*

$$\tilde{\sigma}_{[\mu]}(\boldsymbol{I}, \boldsymbol{X}) : \mathbb{F}^{\mu+\log\mu} \mapsto \mathbb{F} \quad \textit{where} \quad \tilde{\sigma}_{[\mu]}(\boldsymbol{i}, \boldsymbol{X}) := \tilde{\sigma}_i(\boldsymbol{X}) \ \forall i \in [\mu].$$

$$\tilde{\sigma}_{[\mu]}^{-1}(\boldsymbol{I}, \boldsymbol{X}) : \mathbb{F}^{\mu+\log\mu} \mapsto \mathbb{F} \quad \textit{where} \quad \tilde{\sigma}_{[\mu]}^{-1}(\boldsymbol{i}, \boldsymbol{X}) := \tilde{\sigma}_i^{-1}(\boldsymbol{X}) \ \forall i \in [\mu].$$

Note that here, $\boldsymbol{i}$ is a $\log\mu$ length binary string corresponding to the value i.

It is convenient to consider the permutation and inverse permutations as relations

$$\mathcal{R}_\sigma = \{(\boldsymbol{x}, \boldsymbol{y}) : \sigma(\boldsymbol{x}) = \boldsymbol{y}\} \quad \text{and} \quad \mathcal{R}_{\sigma^{-1}} = \{(\boldsymbol{x}, \boldsymbol{y}) : \boldsymbol{x} = \sigma^{-1}(\boldsymbol{y})\}.$$

Specifically, we consider their indicator functions, $B_\mu \times B_\mu \mapsto B$, of these relations.

$$\mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) = \mathbb{1}_{\sigma^{-1}}(\boldsymbol{Y}, \boldsymbol{X}) = \begin{cases} 1 & (\boldsymbol{X}, \boldsymbol{Y}) \in \mathcal{R}_\sigma \\ 0 & (\boldsymbol{X}, \boldsymbol{Y}) \notin \mathcal{R}_\sigma \end{cases}.$$

This can trivially be extended into multilinear extension using Lemma 1 as

$$\sum_{\boldsymbol{x}, \boldsymbol{y} \in B_\mu} \mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) \cdot \mathsf{eq}(\boldsymbol{x}||\boldsymbol{y}, \boldsymbol{X}||\boldsymbol{Y}) = \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}||\sigma(\boldsymbol{x}), \boldsymbol{X}||\boldsymbol{Y})$$

however, this degree 1 formulation has 2μ variables and thus a sum-check protocol will involve a sum over $2^{2\mu} = n^2$ evaluation points.

The multilinearity of σ and σ^{-1} allows for a more succinct arithmetization. Because σ is a multilinear map, that is it is linear separately in each element of $\boldsymbol{X}$, $\mathcal{R}_\sigma$ can be expressed as the intersection of μ relations. If we define

$$\mathcal{R}_{\sigma_i} = \{(\boldsymbol{x}, \boldsymbol{y}) : \sigma_i(\boldsymbol{x}) = \boldsymbol{y}_i\} \quad \text{then} \quad \mathcal{R}_\sigma = \bigcap_{i=1}^{\mu} \mathcal{R}_{\sigma_i}.$$

This implies that

$$\mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) = \mathbb{1}_{\cap\sigma_i}(\boldsymbol{X}, \boldsymbol{Y}).$$

Given the identity $\mathbb{1}_{A \cap B} = \mathbb{1}_A \cdot \mathbb{1}_B$, we have

$$\mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) = \prod_i^\mu \mathbb{1}_{\sigma_i}(\boldsymbol{X}, \boldsymbol{Y}).$$

With this decomposition, the indicator function for $\mathcal{R}_\sigma$ can be arithmetized as a degree μ multivariate polynomial as

$$\mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) = \prod_i^\mu \mathbb{1}_{\sigma_i}(\boldsymbol{X}, \boldsymbol{Y}) = \prod_i^\mu \mathsf{eq}(\tilde{\sigma}_i(\boldsymbol{X}), \boldsymbol{Y}_i) = \mathsf{eq}(\tilde{\sigma}(\boldsymbol{X}), \boldsymbol{Y}).$$

The degree 1 arithmetization and this degree μ arithmetization constitute two extremes. Our results follow from considering various arithmetizations with respective degrees between 1 and μ.

4.6 Permutation check reduced to sumcheck

We adapt the definition for permutation relation from [CBBZ23].

Definition 9 (Permutation relation, indexed). *The indexed relation $\mathcal{R}_{\mathsf{PERM}}$ is the set of tuples*

$$(\mathbb{i}; \mathbb{x}; \mathbb{w}) = \big(\tilde{\sigma}, [[\tilde{\sigma}]]; ([[f]], [[g]]); (f, g)\big)\,,$$

where $\tilde{\sigma} \in \mathcal{F}_{\mu}^{(\leq 1)}$ is the multilinear extension of permutation $\sigma \in \mathsf{PERM}(B_\mu)$, $f, g \in \mathcal{F}_{\mu}^{(\leq 1)}$, and $f(\boldsymbol{x}) = g(\sigma(\boldsymbol{x}))$ for all $\boldsymbol{x} \in B_\mu$.
Note that instead of $\tilde{\sigma}$, the index may contain polynomials that make up $\tilde{\sigma}$ or different forms of this permutation.

Reduction to Sumcheck Previously, in the pursuit of a permutation PIOP with soundness error smaller than $n/|\mathbb{F}|$ to enable usage with polynomial-sized fields, HyperPlonk introduced a sumcheck formulation for permutation check [CBBZ23]. However, their formulation includes unnecessary terms and their protocol requires $O(n \log^2 n)$ field operations. We derive a very similar but more concise sumcheck formulation for permutation check; because our sumcheck formulation is derived in a similar way as in Hyperplonk, both the statement and proof of Lemma 4 are very similar to those of Theorem 3.7 in [CBBZ23]. The details and proof of the sumcheck reduction is in Appendix A.

Lemma 4. *Let $n = 2^\mu$. Given $f, g \in \mathcal{F}_{\mu}^{(\leq 1)}$, and preprocessed $\tilde{\sigma} \in \mathsf{PERM}(B_\mu)$, sumcheck PIOP for*

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{\alpha}) = g(\boldsymbol{\alpha}) \tag{3}$$

proves $\big(\tilde{\sigma}, [[\tilde{\sigma}]]; ([[f]], [[g]]); (f, g)\big) \in \mathcal{R}_{\mathsf{PERM}}$ with perfectly completeness.

5 Motivating different arithmetization of $\tilde{\mathbb{1}}_{\boldsymbol{\sigma}}$

We explore a naïve arithmetization of $\tilde{\mathbb{1}}_\sigma$ and motivate different arithmetizations described in later sections.

5.1 Degree-$\boldsymbol{\mu}$ arithmetization of $\tilde{\mathbb{1}}_{\boldsymbol{\sigma}}$ (HyperPlonk)

Theorem 3. *Let $n = 2^\mu$. Given $f, g \in \mathcal{F}_{\mu}^{(\leq 1)}$, and preprocessed $\tilde{\sigma} = (\tilde{\sigma}_1, \ldots, \tilde{\sigma}_\mu)$ where $\sigma_i : B_\mu \mapsto \{0, 1\}$ for every $i \in [\mu]$, using an arithmetization of $\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$ that is degree μ in $\boldsymbol{X}_i$ for every $i \in [\mu]$, the sumcheck PIOP for Equation 3 described by the prover and verifier in Algorithm 1 for proving $(\tilde{\sigma}, [[\tilde{\sigma}]]; [[f]], [[g]]; f, g) \in \mathcal{R}_{\mathsf{PERM}}$ has the following properties:*
- *Oracles: μ oracles $[[\tilde{\sigma}_1]], \ldots, [[\tilde{\sigma}_\mu]]$ in the instance, each of which is size n, maps to $\{0, 1\}$ over B_μ and half-sparse. Two oracles $[[f]], [[g]]$ in the index, each of size n. $O(\log n)$ oracles of prover messages sent during runtime, each of constant size.*
- *Knowledge soundness error $O(\log^2 n/|\mathbb{F}|)$.*
- *Prover Complexity $O(n \log n)$ field operations.*
- *Verifier Complexity $O(\log n)$.*

– *Verifier queries* $[[f]], [[g]]$, *and batch-queries* $[[\tilde{\sigma}_1]], \dots, [[\tilde{\sigma}_\mu]]$.

Here, we define a different naïve arithmetization of $\tilde{\mathbb{1}}_\sigma$ by viewing $\mathcal{R}_\sigma$ as $\bigcap_{i=1}^{\mu} \mathcal{R}_{\sigma_i}$. The polynomial we obtain is degree μ in $\boldsymbol{X}$,

$$\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) = \mathsf{eq}(\tilde{\sigma}(\boldsymbol{X}), \boldsymbol{Y}) = \prod_{i=1}^{\mu} \mathsf{eq}(\tilde{\sigma}_i(\boldsymbol{X}), \boldsymbol{Y}_i)$$

Substituting this arithmetization into Equation 3, we get

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \prod_{i=1}^{\mu} \mathsf{eq}(\tilde{\sigma}_i(\boldsymbol{x}), \boldsymbol{\alpha}_i) = g(\boldsymbol{\alpha}) \tag{4}$$

We now prove Lemma 3.

Proof.

Knowledge Soundness If there $\exists \boldsymbol{y} \in B_\mu$ such that

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{y}) \neq g(\boldsymbol{y})$$

then the left-hand side and right-hand side are not equal as formal polynomials, so by the Schwartz-Zippel Lemma, it is the case that

$$\Pr_{\alpha \in \mathbb{F}^\mu}\left[\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{\alpha}) = g(\boldsymbol{\alpha})\right] \leq \frac{\mu}{|\mathbb{F}|}.$$

Since any reasonable arithmetizaton of $\tilde{\mathbb{1}}_\sigma$ is at most degree μ in $\boldsymbol{x}$, the sumcheck will be over a virtual polynomial that has μ variables and individual degree at most $\mu + 1$, and its soundness error is upper bounded by $\mu(\mu+1)/|\mathbb{F}| = O(\log^2 n/|\mathbb{F}|)$. Thus, the total soundness error is $O(\log^2 n/|\mathbb{F}|)$.

Prover Complexity The total cost of collapsing evaluation tables is $n(\mu+1) = n \cdot O(\log n)$ field operations.

The prover message in each round of the sumcheck PIOP is a univariate degree-$\mu+1$ polynomial computed as $\mu+2$ evaluation points. Specifically, in the kth round of the sumcheck PIOP for $k \in [\mu]$, computing the polynomial $u_k(Y)$ requires summing up the polynomial $f(\boldsymbol{\beta}, X, \boldsymbol{x}) \cdot \mathsf{eq}(\boldsymbol{\alpha}, \tilde{\sigma}(\boldsymbol{\beta}, X, \boldsymbol{x}))$ over $\boldsymbol{x} \in B_{\mu-k}$. The polynomial inside the sum is a product of $\mu+1$ univariate polynomials, namely $f(\boldsymbol{\beta}, X, \boldsymbol{x})$ and $\mathsf{eq}\big(\boldsymbol{\alpha}_i, \tilde{\sigma}_i(\boldsymbol{\beta}, X, \boldsymbol{x})\big)$ for $i \in [\mu]$. Therefore, computing $u_k(Y)$ requires multiplying $\mu+1$ lists of $\mu+2$ evaluation points for $2^{\mu-k}$ times. Summing together, the total number of field additions is $\sum_{k=1}^{\mu} 2^{\mu-k} = 2^\mu - 1 < n$, and the total number of field multiplications is $\sum_{k=1}^{\mu} 2^{\mu-k} \cdot (\mu+1)(\mu+2) = (n-1)(\mu+1)(\mu+2) = n \cdot O(\log^2 n)$. By using FFT, the total number of field multiplications can be reduced to $n \cdot \tilde{O}(\log n)$.

Query Complexity The verifier needs to query the oracle of g once, the oracle of f once, and batch-query the oracle of $\tilde{\sigma}$ once.

Algorithm 1 Naïve Sumcheck PIOP for Equation 3

```
procedure SUMCHECK VERIFIER($([[\tilde{\sigma}_1]], \ldots, [[\tilde{\sigma}_\mu]]), [[f]], [[g]]$)
    Randomly select $\boldsymbol{\alpha} \leftarrow\!\!\$\ \mathbb{F}^\mu$. Send $\boldsymbol{\alpha}$ to $\mathsf{P}$
    Query $[[g]]$ to get $S \leftarrow g(\boldsymbol{\alpha})$
    for $k \leftarrow 1 \ldots \mu$ do
        Receive $[[u_k]]$ from $\mathsf{P}$
        if $S \neq u_k(0) + u_k(1)$ then
            reject
        end if
        Randomly select $\beta_k \leftarrow\!\!\$\ \mathbb{F}$. Send $\beta_k$ to $\mathsf{P}$
        $S \leftarrow u_k(\beta_k)$
    end for
    Query $[[f]]$ to get $V_f \leftarrow f(\boldsymbol{\beta})$
    Batch-query the oracle of $([[\tilde{\sigma}_1]], \ldots, [[\tilde{\sigma}_\mu]])$ to get $V_i \leftarrow \tilde{\sigma}_i(\boldsymbol{\beta})$ for $i \in [\mu]$.
    if $S \neq V_f \cdot \mathsf{eq}(\boldsymbol{\alpha}, [V_i]_{i=1}^{\mu})$ then
        reject
    end if
    accept
end procedure

procedure SUMCHECK PROVER($(\tilde{\sigma}_1, \ldots, \tilde{\sigma}_\mu), f$)
    Preprocessed tables containing evaluations of $f$ and every $\tilde{\sigma}_i$ over $B_\mu$
    Receive $\boldsymbol{\alpha} \in \mathbb{F}^\mu$ from $\mathsf{V}$
    $\boldsymbol{\beta} \leftarrow []$
    for $k \leftarrow 1 \ldots \mu$ do
        $u_k(X) \leftarrow 0$
        for $\boldsymbol{x} \in B_{\mu-k}$ do
            $u_{\boldsymbol{x}}(X) \leftarrow f(\boldsymbol{\beta}, X, x) \cdot \mathsf{eq}(\boldsymbol{\alpha}, \tilde{\sigma}(\boldsymbol{\beta}, X, \boldsymbol{x}))$        ▷ Using FFT
            $u_k(X) \leftarrow u_k(X) + u_{\boldsymbol{x}}(X)$
        end for
        Send $[[u_k]]$ to $\mathsf{V}$
        Receive $\beta_k$ from $\mathsf{V}$.
        Collapse the tables of $\tilde{\sigma}_i$ and $f$
    end for
end procedure
```

5.2 Low-degree arithmetization for improved prover complexity

We observe that in the prover algorithm for sumcheck PIOP described in Algorithm 1, the bottleneck in prover complexity is computing the univariate degree-μ eq polynomial in every prover message. In fact, among the total $(n-1)(\mu+1)(\mu+2) = O(n\log^2 n)$ field operations the prover needs to perform, $(n-1)\mu(\mu+2)$ of them are solely for computing the equality polynomials, which implies the prover complexity for the rest of the computation is only $O(n\log n)$.

Let us examine the details of this eq polynomial. in the kth round of sumcheck for $k \in [\mu]$, prover needs to compute the following univariate degree-μ eq polynomial for every $\boldsymbol{x} \in B_{\mu-k}$:

$$\begin{aligned}&\mathsf{eq}(\boldsymbol{\alpha}, \tilde{\sigma}(\boldsymbol{\beta}, X, \boldsymbol{x}))\\ &= \prod_{i=1}^{\mu} \mathsf{eq}\Big(\boldsymbol{\alpha}_i, \tilde{\sigma}_i(\boldsymbol{\beta}, X, \boldsymbol{x})\Big)\\ &= \prod_{i=1}^{\mu} \Big((1-\boldsymbol{\alpha}_i)\cdot\big(1-\tilde{\sigma}_i(\boldsymbol{\beta}, X, \boldsymbol{x})\big) + \boldsymbol{\alpha}_i \cdot \tilde{\sigma}_i(\boldsymbol{\beta}, X, \boldsymbol{x})\Big)\end{aligned}$$

where $\boldsymbol{\beta} \in \mathbb{F}^{k-1}$ are challenges from the previous rounds. This polynomial is degree μ in X. It is expensive to compute because it is a product of μ univariate polynomials, which means computing it requires multiplying together μ lists of evaluation points, each of which containing $\mu+2$ points.

This problem can be resolved by reducing the degree of $\boldsymbol{X}$ in the arithmetization of $\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$. However, as pointed out previously in Section 4.5, the trivial multilinear extension of $\mathbb{1}_\sigma$

$$\sum_{\boldsymbol{x},\boldsymbol{y}\in B_\mu} \mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) \cdot \mathsf{eq}(\boldsymbol{x}||\boldsymbol{y}, \boldsymbol{X}||\boldsymbol{Y}) = \sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x}||\sigma(\boldsymbol{x}), \boldsymbol{X}||\boldsymbol{Y})$$

is quadratic, leading to over linear opening cost ($> n$) during compilation of the PIOP even with the best known PCS available.

Idea We seek to find alternative arithmetization of $\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$ that is degree $< \mu$ in every $\boldsymbol{X}_i$ variate.

Our main idea is the following. Let $\ell < \mu$ be a fixed "group parameter." Instead of viewing $\mathcal{R}_\sigma$ as the intersection of μ bit-wise relation ($\mathcal{R}_\sigma = \bigcap_{i=1}^{\mu} \mathcal{R}_{\sigma_i}$, we view it as the intersection of ℓ group relations ($\mathcal{R}_\sigma = \bigcap_{j=1}^{\ell} \mathcal{R}_j$), where $\mathcal{R}_j$ is the intersection of the μ/ℓ bit-wise relations in the jth group for every $j \in [\ell]$ ($\mathcal{R}_j = \bigcap_{i=(j-1)\mu/\ell+1}^{j\mu/\ell} \mathcal{R}_{\sigma_i}$). The μ-bit permutation relation is now broken into ℓ groups of μ/ℓ bits.

Let $\boldsymbol{v}^{(j)}$ denote the jth μ/ℓ-bit group of the any μ-sized vector $\boldsymbol{v}$. Let $\mathbb{1}_j(\boldsymbol{X}, \boldsymbol{Y}^{(j)})$ be the indicator function of $\mathcal{R}_j$ for every $j \in [\ell]$

$$\mathbb{1}_j(\boldsymbol{X}, \boldsymbol{Y}^{(j)}) = \begin{cases} 1 & (\boldsymbol{X}, \boldsymbol{Y}^{(j)}) \in \mathcal{R}_j \\ 0 & (\boldsymbol{X}, \boldsymbol{Y}^{(j)}) \notin \mathcal{R}_j \end{cases}.$$

and let $\tilde{\mathbb{1}}_j$ be its multilinear extension. Then according to this perspective, $\tilde{\mathbb{1}}_\sigma$ is arithmetized as

$$\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y}) = \prod_{j=1}^{\ell} \tilde{\mathbb{1}}_j(\boldsymbol{X}, \boldsymbol{Y}^{(j)})$$

which is degree ℓ in every $\boldsymbol{X}_i$ and multilinear in $\boldsymbol{Y}$.

A natural arithmetization of $\mathbb{1}_j(\boldsymbol{X}, \boldsymbol{Y}^{(j)})$ is

$$\mathsf{p}(\boldsymbol{X}, \boldsymbol{Y}^{(j)}, j) = \mathsf{eq}(\tilde{\sigma}^{(j)}(\boldsymbol{X}), \boldsymbol{Y}^{(j)}, j) = \prod_{i=1}^{\mu/\ell} \mathsf{eq}(\tilde{\sigma}((j-1)\mu/\ell + i, \boldsymbol{X}), \boldsymbol{Y}_i^{(j)})$$

and the arithmetization of $\tilde{\in}_j$ is $\tilde{\mathsf{p}}$.

Substituting this arithmetization into Equation 3, we get the following sumcheck formulation

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \prod_{j=1}^{\ell} \tilde{\mathsf{p}}(\boldsymbol{x}, \boldsymbol{\alpha}^{(j)}, j) = g(\boldsymbol{\alpha})$$

The prover can either include $\tilde{\mathsf{p}}$ in the index (i.e. commit to it during preprocessing), or additionally prove the correctness of $\tilde{\mathsf{p}}$.

6 BiPerm: Linear-time permutation check for sparse PCS

Theorem 4 (BiPerm). *Let $n = 2^\mu$. Given $f, g \in \mathcal{F}_\mu^{(\le 1)}$, and preprocessed $[[\tilde{\mathbb{1}}_{\sigma_L}]], [[\tilde{\mathbb{1}}_{\sigma_R}]] : B_{1.5\mu} \mapsto \{0,1\}$, using an arithmetization of $\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$ that is degree 2 in $\boldsymbol{X}$, the sumcheck PIOP for Equation 3 described by the prover and verifier in Algorithm 2 for proving $(\tilde{\mathbb{1}}_{\sigma_L}, \tilde{\mathbb{1}}_{\sigma_R}, [[\tilde{\mathbb{1}}_{\sigma_L}]], [[\tilde{\mathbb{1}}_{\sigma_R}]]; [[f]], [[g]]; f, g) \in \mathcal{R}_{\mathsf{PERM}}$ has the following properties:*
- *Oracles: two oracles $[[\tilde{\mathbb{1}}_{\sigma_L}]], [[\tilde{\mathbb{1}}_{\sigma_R}]]$ in the index, which are size $n^{1.5}$. Two oracles $[[f]], [[g]]$ in the instance, each of size n. $\log n$ oracles of prover messages sent during runtime, each of size $\log n$.*
- *Knowledge soundness error $O(\log n/|\mathbb{F}|)$.*
- *Prover Complexity $O(n)$ field operations.*
- *Verifier Complexity $O(\log n)$.*
- *Verifier queries $[[f]], [[g]], [[\tilde{\mathbb{1}}_{\sigma_L}]], [[\tilde{\mathbb{1}}_{\sigma_R}]]$.*

In the previous section we arithmetized $\tilde{\mathbb{1}}_\sigma$ as the product of μ multilinear functions, resulting in a polynomial that is degree μ in every $\boldsymbol{X}_i$ and multilinear in $\boldsymbol{Y}$.

Here instead, we arithmetize $\mathbb{1}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$ as the product of just 2 multilinear polynomials, each of which depends on half of $\boldsymbol{Y}$.

Let L and R be the intervals $[1, \mu/2]$ and $[\mu/2 + 1, \mu]$. Consider the pair of relations

$$\mathcal{R}_{\sigma_L} = \bigcap_{i \in L} \mathcal{R}_{\sigma_i} \quad \text{and} \quad \mathcal{R}_{\sigma_R} = \bigcap_{i \in R} \mathcal{R}_{\sigma_i}.$$

Let $\mathbb{1}_{\sigma_L}$ and $\mathbb{1}_{\sigma_R}$ be the indicator functions for $\mathcal{R}_{\sigma_L}$ and $\mathcal{R}_{\sigma_R}$, respectively; let $\tilde{\mathbb{1}}_{\sigma_L}$ and $\tilde{\mathbb{1}}_{\sigma_R}$ be their multilinear extensions. By definition, $\mathcal{R}_\sigma = \mathcal{R}_{\sigma_L} \cap \mathcal{R}_{\sigma_L}$, and by arithmetizing $\tilde{\mathbb{1}}_\sigma$ as $\tilde{\mathbb{1}}_{\sigma_L} \cdot \tilde{\mathbb{1}}_{\sigma_R}$, Equation 3 can be written as

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_{\sigma_L}(\boldsymbol{x}, \boldsymbol{\alpha}_L) \cdot \tilde{\mathbb{1}}_{\sigma_R}(\boldsymbol{x}, \boldsymbol{\alpha}_R) = g(\boldsymbol{\alpha}). \tag{5}$$

The sumcheck PIOP for the evaluation of the left-hand-side of Equation 5 is described in Algorithm 2. The claims in Theorem 4 that the verifier to performs $O(\log n)$ work and the protocol has soundness error $O(\log n/|\mathbb{F}|)$ follow immediately from the description of the PIOP. A full proof follows.

Proof.

Knowledge Soundness The soundness error of Schwartz-Zippel Lemma is $\mu/|\mathbb{F}|$. The sumcheck is over a virtual polynomial that has μ variables and degree at must 3 in each variable, so its soundness error is at most $3\mu/|\mathbb{F}|$. Thus, the total soundness error for BiPerm is $O(\log n/|\mathbb{F}|)$.

Prover Complexity Provided the evaluations of $\tilde{\mathbb{1}}_{\sigma_L}(\boldsymbol{X}, \boldsymbol{\alpha}_L)$ and $\tilde{\mathbb{1}}_{\sigma_R}(\boldsymbol{X}, \boldsymbol{\alpha}_R)$ over B_μ, the sum-check protocol in Figure 2 can be executed with $O(n)$ prover, $O(\mu)$ verifier work, and has soundness error $O(\mu/|\mathbb{F}|)$. The only thing left to establish is that the prover can compute the evaluation tables of $\tilde{\mathbb{1}}_{\sigma_L}(\boldsymbol{X}, \boldsymbol{\alpha}_L)$ and $\tilde{\mathbb{1}}_{\sigma_R}(\boldsymbol{X}, \boldsymbol{\alpha}_R)$ with $O(n)$ work.

WLOG, we will consider the definition of $\tilde{\mathbb{1}}_{\sigma_L}$.

$$\begin{aligned}
\tilde{\mathbb{1}}_{\sigma_L}(\boldsymbol{X}, \boldsymbol{Y}_L) &= \sum_{\boldsymbol{x} \in B_\mu, \boldsymbol{y}_L \in B_{\mu/2}} \mathsf{eq}(\boldsymbol{x}||\boldsymbol{y}_L, \boldsymbol{X}||\boldsymbol{Y}_L)\mathbb{1}_{\sigma_L}(\boldsymbol{x}, \boldsymbol{y}_L) \\
&= \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}||\sigma_L(\boldsymbol{x}), \boldsymbol{X}||\boldsymbol{Y}_L) \\
&= \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{X}) \cdot \mathsf{eq}(\sigma_L(\boldsymbol{x}), \boldsymbol{Y}_L).
\end{aligned}$$

When $\boldsymbol{X} \in B_\mu$, the only non-zero term in the sum is the case where $\boldsymbol{x} = \boldsymbol{X}$, so the expression simplifies as $\mathsf{eq}(\sigma_L(\boldsymbol{X}), \boldsymbol{Y}_L)$.

It is well known that $\mathsf{eq}(\boldsymbol{y}_L, \boldsymbol{\alpha}_L)$ can be evaluated at all points $\boldsymbol{y}_L \in B_{\mu/2}$ in time $O(2^{\mu/2})$. As $\sigma_L(\boldsymbol{X})$ exclusively takes on values in $B_{\mu/2}$, each evaluation of $\tilde{\mathbb{1}}_{\sigma_L}(\boldsymbol{x}, \boldsymbol{\alpha}_L)$ $\forall \boldsymbol{x} \in B_{\mu/2}$ is merely a lookup into a table of $\mathsf{eq}(\boldsymbol{y}_L, \boldsymbol{\alpha}_L)$'s evaluations.

Computing $\mathsf{eq}(\boldsymbol{y}_L, \boldsymbol{\alpha}_L)$ $\forall \boldsymbol{y}_L \in B_{\mu/2}$ requires $O(2^{\mu/2}) = O(\sqrt{n})$ work.

The remainder of the protocol is a constant-degree sum-check over μ variables. The standard linear time sumcheck algorithm can be applied for an overall $O(n)$ prover work.

Algorithm 2 Sumcheck PIOP in BiPerm

```
procedure BiPerm Verifier($[[\tilde{\mathbb{1}}_{\sigma_L}]], [[\tilde{\mathbb{1}}_{\sigma_R}]]), [[f]], [[g]])$
    Randomly select $\boldsymbol{\alpha} \leftarrow\$ \mathbb{F}^{\mu}$. Send $\boldsymbol{\alpha}$ to $\mathsf{P}$
    Query $[[g]]$ to get $S \leftarrow g(\boldsymbol{\alpha})$
    $\boldsymbol{\beta} \leftarrow []$
    for $k \leftarrow 1 \ldots \mu$ do
        Receive $[[u_k]]$ from $\mathsf{P}$
        if $S \neq u_k(0) + u_k(1)$ then
            reject
        end if
        Randomly select $\beta_k \leftarrow\$ \mathbb{F}$. Append to $\boldsymbol{\beta}$. Send $\beta_k$ to $\mathsf{P}$
        $S \leftarrow u_k(\beta_k)$
    end for
    Query $[[f]]$ to get $V_f \leftarrow f(\boldsymbol{\beta})$
    Query $[[\tilde{\mathbb{1}}_{\sigma_L}]]$ to get $V_{\sigma_L} \leftarrow \tilde{\mathbb{1}}_{\sigma_L}(\boldsymbol{\beta}, \boldsymbol{\alpha}_{[:\mu/2]})$
    Query $[[\tilde{\mathbb{1}}_{\sigma_R}]]$ to get $V_{\sigma_R} \leftarrow \tilde{\mathbb{1}}_{\sigma_R}(\boldsymbol{\beta}, \boldsymbol{\alpha}_{[\mu/2+1:]})$
    if $S \neq V_f \cdot V_{\sigma_L} \cdot V_{\sigma_R}$ then
        reject
    end if
    accept
end procedure

procedure BiPerm Prover($\tilde{\mathbb{1}}_{\sigma_L}, \tilde{\mathbb{1}}_{\sigma_R}, f$)
    Preprocessed tables with evaluations of $f$, $\tilde{\mathbb{1}}_{\sigma_L}(\cdot, \boldsymbol{\alpha}_L)$ and $\tilde{\mathbb{1}}_{\sigma_R}(\cdot, \boldsymbol{\alpha}_R)$ over $B_\mu$
    Receive $\boldsymbol{\alpha} \in \mathbb{F}^{\mu}$ from $\mathsf{V}$
    $\boldsymbol{\beta} \leftarrow []$
    for $k \leftarrow 1 \ldots \mu$ do
        $u_k(X) \leftarrow 0$
        for $\boldsymbol{x} \in B_{\mu-k}$ do
            $u_{\boldsymbol{x}}(X) \leftarrow f(\boldsymbol{\beta}, X, x) \cdot \tilde{\mathbb{1}}_{\sigma_L}\big((\boldsymbol{\beta}, X, \boldsymbol{x}), \boldsymbol{\alpha}_L\big) \cdot \tilde{\mathbb{1}}_{\sigma_R}\big((\boldsymbol{\beta}, X, \boldsymbol{x}), \boldsymbol{\alpha}_R\big)$    ▷ FFT
            $u_k(X) \leftarrow u_k(X) + u_{\boldsymbol{x}}(X)$
        end for
        Send $[[u_k(X)]]$ to $\mathsf{V}$
        Receive $\beta_k$ from $\mathsf{V}$. Append to $\boldsymbol{\beta}$
        Collapse the evaluation tables
    end for
end procedure
```

6.1 Compiling BiPerm using PCS

To keep the prover work linear in the compiled permutation argument, the opening cost of the committed polynomials must be $O(n)$ field operations. This is not immediately obvious, as $\tilde{\mathbb{1}}_{\sigma_L}$ and $\tilde{\mathbb{1}}_{\sigma_R}$ are polynomials in 1.5μ variables and thus of size $n^{1.5}$. However, we are saved by the n-sparsity of these polynomials, namely that they merely have n non-zero entries. By using PCS that has cost only dependent on the sparsity of the polynomial, such as Dory [Lee21] or KZH [KZHB25], we can achieve linear prover and preprocessing cost in the resulting permutation argument. For any hash and code-based PCS, e.g. [AHIV17; BBHR18a] , unfortunately, the preprocessing cost will be $n^{1.5}$. The reason is that even for sparse polynomials, the error-corrected encoding must be non-sparse. Thus, committing to the polynomial, which is done in preprocessing, is linear in the size of the polynomial.

7 MulPerm: Almost linear-time permutation check for any PCS

Notation In this section, we frequently work with field elements $j \in [\ell]$, where ℓ is a fixed parameter. For any such j, we use $\langle j \rangle$ to denote the $\log \ell$-bit representation of $j-1$ and j' to denote $(j-1)\mu/\ell$.

The BiPerm PIOP can only be compiled using sparse PCS because the total weight the oracles in the index is larger than n. Without such PCS, committing to and opening the two 1.5μ-variate sub-indicator functions $(\tilde{\mathbb{1}}_{\sigma_L}, \tilde{\mathbb{1}}_{\sigma_R})$ is too costly.

MulPerm aims to avoid preprocessing the multilinear extensions of indicator functions so that the PIOP can be compiled with any PCS. The correctness of the multilinear extensions will instead be batch-proved using an additional sumcheck protocol, as described in Section 8 of HyperPlonk [CBBZ23].

Theorem 5 (MulPerm). *Let $n = 2^\mu$. Given $f, g \in \mathcal{F}_\mu^{(\leq 1)}$, and preprocessed $\tilde{\sigma}_{[\mu]} : \mathbb{F}^{\log \mu + \mu} \mapsto \mathbb{F}$, using an arithmetization of $\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$ that is degree $\sqrt{\log n}$ in $\boldsymbol{X}$, the double-sumcheck PIOP for Equation 3 described by the prover and verifier in Algorithm 3 proves $(\tilde{\sigma}_{[\mu]}, [[\tilde{\sigma}_{[\mu]}]]; [[f]], [[g]]; f, g) \in \mathcal{R}_{\mathsf{PERM}}$ with the following properties:*

- *Oracles: one oracle $[[\tilde{\sigma}_{[\mu]}]]$ in the index, which is size $n \log n$ and maps to $\{0,1\}$ over $B_{\log \mu + \mu}$. Two oracles $[[f]], [[g]]$ in the instance, each of size n. $O(\log n)$ oracles of prover messages sent during runtime, each of size $O(\sqrt{\log n})$.*
- *Soundness error $\frac{\text{polylog}(n)}{|\mathbb{F}|}$,*
- *Proof size $O(\log n)$,*
- *Prover performs $n \cdot \tilde{O}(\sqrt{\log n})$ field operations*
- *Verifier queries $[[f]], [[g]]$, batch-queries $[[\tilde{\sigma}_{[\mu]}]]$ at $\sqrt{\log n}$ positions, and each of the oracles of the prover messages.*

Suppose we arithmetize $\tilde{\mathbb{1}}_\sigma(\boldsymbol{X}, \boldsymbol{Y})$ as the product of ℓ sub-indicator functions, each of which depend on μ/ℓ bits of $\boldsymbol{Y}$, where $\ell \leq \mu$ is some fixed parameter. Looking ahead, we eventually set $\ell = \sqrt{\mu}$ for $\mathsf{MulPerm}$ to minimize the prover cost.

Let interval $\mathcal{I}_j$ be $[j' + 1, j' + \mu/\ell]$ for every $j \in [\ell]$. Consider the relations:

$$\mathcal{R}_j = \bigcap_{i \in \mathcal{I}_j} \mathcal{R}_{\sigma_i} \ \ \forall j \in [\ell]$$

Let $\mathbb{1}_j$ be the indicator function for $\mathcal{R}_j$, and let $\tilde{\mathbb{1}}_j$ be its multilinear extension. By definition, $\mathcal{R}_\sigma = \bigcup_{j \in [\ell]} \mathcal{R}_j$, and by arithmetizing $\tilde{\mathbb{1}}_\sigma$ as $\prod_{j \in [\ell]} \tilde{\mathbb{1}}_j$, Equation 3 can be written as

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \prod_{j \in [\ell]} \tilde{\mathbb{1}}_j(\boldsymbol{x}, \boldsymbol{\alpha}^{(j)}) = g(\boldsymbol{\alpha}) \ \ \forall j \in [\ell]$$

Notice that since we are no longer preprocessing $\tilde{\mathbb{1}}_j$, we can hardcode the slice of $\boldsymbol{\alpha}$ into the arithemetization. Let the arithmetization of $\mathbb{1}_j(\boldsymbol{X}, \boldsymbol{Y}^{(j)})$ be $\mathsf{p}(\boldsymbol{X}, \langle j \rangle)$ for every $j \in [\ell]$.

Particularly, interpret the random challenge $\boldsymbol{\alpha} \in \mathbb{F}^\mu$ as a polynomial $\boldsymbol{\alpha} : \mathbb{F}^{\log \mu} \mapsto \mathbb{F}$, such that $\boldsymbol{\alpha}(\langle i \rangle) = \boldsymbol{\alpha}_i$ for $i \in [\mu]$. The partial product polynomial $\mathsf{p} : B_{\mu + \log \ell} \mapsto \mathbb{F}$ is defined as

$$\mathsf{p}(\boldsymbol{x}, \langle j \rangle) := \prod_{i=1}^{\mu/\ell} \mathsf{eq}\big(\boldsymbol{\alpha}(\langle j' + i \rangle), \tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, \boldsymbol{x})\big) \tag{6}$$

where $\langle j' + i \rangle$ denotes the $\log \mu$-bit representation of $j' + i - 1$. The arithmetization of $\tilde{\mathbb{1}}_j$ is simply:

$$\tilde{\mathsf{p}}(\boldsymbol{x}^*, \boldsymbol{j}^*) = \sum_{\boldsymbol{x} \in B_\mu, j \in [\ell]} \mathsf{eq}\big((\boldsymbol{x}, \langle j \rangle), (\boldsymbol{x}^*, \boldsymbol{j}^*)\big) \cdot \mathsf{p}(\boldsymbol{x}, \langle j \rangle) \tag{7}$$

As $[[\tilde{\mathsf{p}}]]$ is not included in the indexer, the prover will need to prove that $\tilde{\mathsf{p}}$ is indeed the multilinear extension of p, which can be accomplished by another sumcheck. $\mathsf{MulPerm}$ therefore is a double-sumcheck PIOP.

Substituting $\tilde{\mathsf{p}}$ into Equation 3, we obtain the following sumcheck formulation, the left-hand side of which is degree $\ell + 1$ in $\boldsymbol{x}$:

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \prod_{j \in [\ell]} \tilde{\mathsf{p}}(\boldsymbol{x}, \langle j \rangle) = g(\boldsymbol{\alpha})$$

Remark 1. This double-sumcheck protocol can be thought of as a non-generic 2-layered GKR circuit, in which the top layer computes the left-hand side of Equation 8 and the lower layer computes $\tilde{\mathsf{p}}(\boldsymbol{x}')$ for every $\boldsymbol{x}' \in B_{\mu + \log \ell}$. The two ideas are equivalent.

Algorithm 3 MulPerm PIOP. $\tilde{\mathsf{p}}$ is defined as in Equation 6 and 7.

```
 1: procedure MulPerm Verifier($[[\tilde{\sigma}_{[\mu]}]], [[f]], [[g]]$)
 2:     Randomly select $\boldsymbol{\alpha} \leftarrow\$ \mathbb{F}^{\mu}$. Send $\boldsymbol{\alpha}$ to $\mathsf{P}$
 3:     Query $[[g]]$ to get $S \leftarrow g(\boldsymbol{\alpha})$
 4:     Sumcheck1 Verifier($[[f]], S$). If procedure rejects, output reject. Otherwise,
    gets returned values $[P_j]_{j=1}^{\ell}$ and $\boldsymbol{\beta} \in \mathbb{F}^{\mu}$.
 5:     Randomly select $\boldsymbol{t} \in \mathbb{F}^{\log \ell}$. Send $\boldsymbol{t}$ to $\mathsf{P}$
 6:     Compute $S_{\tilde{\mathsf{p}}} \leftarrow \sum_{j\in[\ell]} \mathsf{eq}(\boldsymbol{t}, \langle j \rangle) \cdot P_j$
 7:     Sumcheck2 Verifier(($[[\tilde{\sigma}_{[\mu]}]], \boldsymbol{\beta}||\boldsymbol{t}, S_{\tilde{\mathsf{p}}}$)).
 8: end procedure
 9:
10: procedure MulPerm Prover($\tilde{\sigma}_{[\mu]}, f$)
11:     Preprocessed tables containing evaluations of $f$ over $B_\mu$ and $\tilde{\sigma}_{[\mu]}$ over $B_{\mu+\log \mu}$
12:     Receive $\boldsymbol{\alpha} \in \mathbb{F}^{\mu}$ from $\mathsf{V}$
13:     Evaluation table of $\tilde{\mathsf{p}}$ over $B_{\mu+\log \ell} \leftarrow$ ComputePartialProducts($\tilde{\sigma}_{[\mu]}, \boldsymbol{\alpha}$)
14:     $\boldsymbol{\beta} \in \mathbb{F}^{\mu} \leftarrow$ Sumcheck1 Prover($\tilde{\mathsf{p}}, f$)
15:     Receive $\boldsymbol{t} \in \mathbb{F}^{\log \ell}$ from $\mathsf{V}$
16:     Sumcheck2 Prover($\tilde{\sigma}_{[\mu]}, \boldsymbol{\beta}||\boldsymbol{t}$)
17: end procedure
```

Our double-sumcheck PIOP algorithm is composed of three parts.

1. Prover computes $\tilde{\mathsf{p}}(\boldsymbol{x}')$ for every $\boldsymbol{x}' \in B_{\mu+\log \ell}$. Section 7.1 shows an efficient algorithm for doing so.
2. Verifier sends random challenges $\boldsymbol{\alpha} \in \mathbb{F}^{\mu}$ to prover and queries the oracle of g to get $S \leftarrow g(\boldsymbol{\alpha})$. Prover and verifier then engage in a sumcheck PIOP to reduce
$$\sum_{\boldsymbol{x}\in B_\mu} f(\boldsymbol{x}) \prod_{j\in[\ell]} \tilde{\mathsf{p}}(\boldsymbol{x}, \langle j \rangle) = S \tag{8}$$
to ℓ claims $\tilde{\mathsf{p}}(\boldsymbol{\beta}, \langle j \rangle) = P_j$ for every $j \in [\ell]$. The details of this first sumcheck is described in Section 7.2.
3. Verifier sends random challenges $\boldsymbol{t} \in \mathbb{F}^{\log \ell}$ to prover and computes $S_{\tilde{\mathsf{p}}} \leftarrow \sum_{j\in[\ell]} \mathsf{eq}(\boldsymbol{t}, \langle j \rangle) \cdot P_j$. Let $\boldsymbol{\beta}' := \boldsymbol{\beta}||\boldsymbol{t}$. Prover and verifier engage in another sumcheck to prove $\tilde{\mathsf{p}}(\boldsymbol{\beta}') = S_{\tilde{\mathsf{p}}}$, or equivalently,
$$\sum_{\boldsymbol{x}\in B_\mu, j\in[\ell]} \mathsf{eq}(\boldsymbol{\beta}', \boldsymbol{x}||\langle j \rangle) \underbrace{\prod_{i=1}^{\mu/\ell} \mathsf{eq}\big(\boldsymbol{\alpha}(\langle j' + i \rangle), \tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, \boldsymbol{x})\big)}_{\mathsf{p}(\boldsymbol{x}, \langle j \rangle)} = S_{\tilde{\mathsf{p}}} \tag{9}$$
This second sumcheck is described in Section 7.3.

In the following subsections, we will describe the details of each component, and discuss how to set the parameter ℓ in the end, which intuitively represents the number of partial products.

7.1 Evaluating $\tilde{\mathsf{p}}$ over the boolean hypercube

The prover needs to compute $\tilde{\mathsf{p}}(\boldsymbol{x}')$ for every $\boldsymbol{x}' \in B_{\mu+\log\ell}$ before engaging in the sumcheck protocols. Notice that because $\tilde{\mathsf{p}}(\boldsymbol{x}') = \mathsf{p}(\boldsymbol{x}')$ over the Boolean hypercube, the prover can simply compute the partial product polynomials p.

Naïvely, the prover needs to evaluate the formulation at all $n\ell$ boolean vectors, and each evaluation requires μ/ℓ field multiplications, which sums to $n\mu$ total field multiplications. This is obviously too inefficient for our goal of achieving an almost linear permcheck protocol.

We make an important observation that every bit-wise eq polynomial only has image space of size 2. Thus, the image space of p, which is defined as a product of μ/ℓ bit-wise eq polynomials, has size at most $2^{\mu/\ell}$. Algorithm 6 is a "bucketing" algorithm that takes advantage of this observation when computing the evaluations of $\tilde{p}$.

Lemma 5. *Algorithm 4 computes $\tilde{\mathsf{p}}(\boldsymbol{x}')$ for all $\boldsymbol{x}' \in B_{\mu+\log\ell}$ using $o(n)$ field operations.*

Proof. For any fixed $j \in [\ell]$, there are at most $2^{\mu/\ell}$ possible evaluations across all different $\boldsymbol{x} \in B_{\mu/\ell}$. We can compute these distinct $\ell\cdot 2^{\mu/\ell}$ evaluations, each taking μ/ℓ field multiplications, and then look up the evaluation for the corresponding p. The total number of field multiplications is therefore $(\mu/\ell)\cdot\ell\cdot 2^{\mu/\ell} = \mu\cdot 2^{\mu/\ell} = \mu\cdot n^{1/\ell} = o(n)$ for any $\ell > 1$.

Algorithm 4 Algorithm for efficiently computing $\tilde{\mathsf{p}}(\boldsymbol{x}')$ for every $\boldsymbol{x}' \in B_{\mu+\log\ell}$

```
1: procedure ComputePartialProducts($\tilde{\sigma}_{[\mu]}, \boldsymbol{\alpha} \in \mathbb{F}^{\mu}$)
2:     Preprocessed evaluation tables of $\tilde{\sigma}_{[\mu]}$ over $B_{\mu+\log\ell}$
3:     Let $\{\boldsymbol{s}_1, \boldsymbol{s}_2, \cdots, \boldsymbol{s}_{2^{\mu/\ell}}\}$ be the set of all $\mu/\ell$-bit strings
4:     Initialize empty evaluation table of $\tilde{\mathsf{p}}$
5:     Initialize table $\boldsymbol{S}$ of size $2^{\mu/\ell} \times \ell$
6:     for $i \leftarrow 1 \dots 2^{\mu/\ell}$ do
7:         Compute $\boldsymbol{S}_{ij} \leftarrow \mathsf{eq}(\boldsymbol{\alpha}[j'+1 : j'+\mu/\ell], \boldsymbol{s}_i)$
8:     end for
9:     for $\boldsymbol{x} \in B_{\mu}, j \in [\ell]$ do
10:        Look up $i$ such that $\boldsymbol{s}_i = [\tilde{\sigma}_{[\mu]}(\langle j'+1\rangle, \boldsymbol{x}), \dots, \tilde{\sigma}_{[\mu]}(\langle j'+\mu/\ell\rangle, \boldsymbol{x})]$
11:        Look up $\boldsymbol{S}_{ij}$, and use it to fill in the evaluation of $\tilde{\mathsf{p}}(\boldsymbol{x}, \langle j\rangle)$
12:    end for
13:    Return the evaluation table of $\tilde{\mathsf{p}}$
14: end procedure
```

7.2 First sumcheck

$$\sum_{\boldsymbol{x}\in B_{\mu}} f(\boldsymbol{x}) \prod_{j\in[\ell]} \tilde{\mathsf{p}}(\boldsymbol{x}, \langle j\rangle) = S \tag{8}$$

Lemma 6. *The sumcheck PIOP described by the corresponding prover in Algorithm 5 reduces Equation 8 to ℓ claims $\tilde{\mathsf{p}}(\boldsymbol{\beta}, \langle j \rangle) = P_j$ for every $j \in [\ell]$, where $\boldsymbol{\beta} \in \mathbb{F}^{\mu+\log \ell}$ are challenges sent by the verifier.*

- *Prover sends μ prover messages as oracles, each of size $\ell + 2$.*
- *Soundness error $\frac{(\ell+1)(\mu+\log \ell)}{|\mathbb{F}|}$*
- *Prover performs $n \cdot \tilde{O}(\ell)$ field operations.*
- *Proof size $(\mu) \cdot (\ell + 2) + \ell$.*
- *Verifier queries the oracle of f once*

Proof. The number and size of prover messages, the proof size, and the number of verifier queries can be found trivially in the algorithm.

Soundness The formulation for this sumcheck has $\mu + \log \ell$ variates and at most degree $\ell + 1$ in each variable. The sumcheck therefore has soundness error $\frac{(\ell+1)(\mu+\log \ell)}{|\mathbb{F}|}$.

Prover Complexity In the kth round for $1 \le k \le \mu$, there are $2^{\mu-k} \cdot (\ell+1)^2$ field multiplications and $2^{\mu-k}$ additions. In the $(\mu+k)$th round for $1 \le k \le \log \ell$, there are $(\ell/2^k + 1)^2$ field multiplications. Summing together, the total number of field multiplications is

$$\begin{aligned}
&\sum_{k=1}^{\mu} 2^{\mu-k} \cdot (\ell+1)^2 + \sum_{k=1}^{\log \ell} (\ell/2^k + 1)^2 \\
&< \sum_{k=1}^{\mu} 2^{\mu-k} \cdot (\ell+1)^2 + \sum_{k=1}^{\log \ell} (2\ell/2^k)^2 \\
&= n(\ell+1)^2 \cdot \sum_{k=1}^{\mu} 2^{-k} + 4\ell^2 \cdot \sum_{k=1}^{\log \ell} 2^{-2k} \\
&= n(\ell+1)^2(1 - 1/n) + 4\ell^2(1 - \ell^{-2})/3 \\
&= n \cdot O(\ell^2)
\end{aligned}$$

and the total number of field additions is $\sum_{k=1}^{\mu} 2^{\mu-k} = n - 1$.
The algorithm can be further optimized by employing FFT for polynomial multiplication [CBBZ23], which brings the number of field multiplications down to $n \cdot \tilde{O}(\ell)$.

Algorithm 5 First Sumcheck PIOP in $\mathsf{MulPerm}$

```
1: procedure SUMCHECK1 VERIFIER($[[f]], S$)
2:     $\boldsymbol{\beta} \leftarrow []$
3:     for $k \leftarrow 1 \ldots \mu$ do
4:         Receive $[[u_k]]$ from $\mathsf{P}$
5:         if $S \neq u_k(0) + u_k(1)$ then
6:             reject
7:         end if
8:         Randomly select $\beta_k \leftarrow\!\$\, \mathbb{F}$. Append to $\boldsymbol{\beta}$. Send $\beta_k$ to $\mathsf{P}$
9:         $S \leftarrow u_k(\beta_k)$
10:    end for
11:    Query $[[f]]$ to get $V_f \leftarrow f(\boldsymbol{\beta})$
12:    Receive $[P_j]_{j=1}^{\ell}$ from $\mathsf{P}$
13:    if $S \neq V_f \cdot \prod_{j \in [\ell]} P_j$ then
14:        reject
15:    end if
16:    return $[P_j]_{j=1}^{\ell}, \boldsymbol{\beta}$
17: end procedure
18:
19: procedure SUMCHECK1 PROVER($\tilde{\mathsf{p}}, f$)
20:    Preprocessed tables containing evaluations of $f$ over $B_\mu$ and $\tilde{\mathsf{p}}$ over $B_{\mu+\log \ell}$
21:    $\boldsymbol{\beta} \leftarrow []$
22:    for $k \leftarrow 1 \ldots \mu$ do
23:        $u_k(X) \leftarrow 0$
24:        for $\boldsymbol{x} \in B_{\mu-k}$ do
25:            $u_{\boldsymbol{x}}(X) \leftarrow f(\boldsymbol{\beta}, X, \boldsymbol{x}) \cdot \prod_{j \in [\ell]} \tilde{\mathsf{p}}\big((\boldsymbol{\beta}, X, \boldsymbol{x}), \langle j \rangle\big)$        ▷ Using FFT
26:            $u_k(X) \leftarrow u_k(X) + u_{\boldsymbol{x}}(X)$
27:        end for
28:        Send $[[u_k]]$ to $\mathsf{V}$
29:        Receive $\beta_k$ from $\mathsf{V}$ and append it to $\boldsymbol{\beta}$
30:        Collapse the evaluation tables
31:    end for
32:    Send $P_j := \tilde{\mathsf{p}}(\boldsymbol{\beta}, \langle j \rangle)$ for every $j \in [\ell]$ to $\mathsf{V}$
33:    return $\boldsymbol{\beta}$
34: end procedure
```

7.3 Second sumcheck

$$\sum_{\boldsymbol{x}\in B_\mu, j\in[\ell]} \mathsf{eq}(\boldsymbol{\beta}', \boldsymbol{x}||\langle j\rangle) \underbrace{\prod_{i=1}^{\mu/\ell} \mathsf{eq}\big(\boldsymbol{\alpha}(\langle j'+i\rangle), \tilde{\sigma}_{[\mu]}(\langle j'+i\rangle, \boldsymbol{x})\big)}_{\mathsf{p}(\boldsymbol{x},\langle j\rangle)} = S_{\tilde{\mathsf{p}}} \tag{9}$$

The second sumcheck has degree $\mu/\ell + 1$, which implies direct computation is expensive $(n \cdot \tilde{O}(\log n/\ell))$. We describe a *bucketing algorithm* below to reduce the number of field operations.

Observe that, as noted previously in Section 7.1, every bit-wise $\mathsf{eq}(\alpha(i), \sigma(i, \cdot))$ polynomial only has image space of size 2, which implies the total number of possible polynomial identities for bit-wise eq is limited and that the total number of polynomial identities for partial product is also limited. If we can efficiently compute all the possible identities/buckets for the univariate $\tilde{\mathsf{p}}$ polynomial, then instead of computing them individually for every $\boldsymbol{x}'$ in every round of sumcheck, we can permute the buckets and then sum up the product of each identity with the polynomials $\mathsf{eq}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}'), \boldsymbol{\beta}'\big)$ for $\boldsymbol{x}' \in B_{\mu+\log\ell-k}$ such that the corresponding partial product falls in the current bucket.

Lemma 7. *The sumcheck PIOP for proving Equation 9 described in Algorithm 7 has the following properties:*

- *Prover sends $\mu + \log\ell$ prover messages as oracles, each of size $\mu/\ell + 2$.*
- *Soundness error $\frac{(\mu/\ell+1)(\mu+\log\ell)}{|\mathbb{F}|}$.*
- *Prover performs $n \cdot \tilde{O}(\mu/\ell) + \ell 2^\ell$ field operations.*
- *Proof size $(\mu + \log\ell) \cdot (\mu/\ell + 2)$.*
- *Verifier queries $[[\tilde{\sigma}_{[\mu]}]]$ once.*

Bucketing Start by considering the first round of sumcheck, where the prover needs to compute $\sum_{\boldsymbol{x}'\in B_{\mu+\log\ell-1}} \mathsf{eq}\big((X, \boldsymbol{x}'), \boldsymbol{\beta}'\big)\tilde{\mathsf{p}}(X, \boldsymbol{x}')$ or equivalently

$$\sum_{\boldsymbol{x}\in B_{\mu-1}, j\in[\ell]} \mathsf{eq}\big((X, \boldsymbol{x}, \langle j\rangle), \boldsymbol{\beta}'\big)\tilde{\mathsf{p}}\big((X, \boldsymbol{x}), \langle j\rangle\big)$$

Recall that $\tilde{\mathsf{p}}$ is the multilinear extension of the product of μ/ℓ bit-wise eq between $\boldsymbol{\alpha}$ and $\tilde{\sigma}_{[\mu]}$. Observe that since each univariate $\tilde{\sigma}_{[\mu]}$ maps to $0, 1$, its polynomial identity must be among $0, 1, X, 1-X$. This implies that, for every $j \in [\ell]$, there are at most $2^{2\mu/\ell} < 2^\mu$ possible polynomial identities for $\tilde{\mathsf{p}}\big((X, \boldsymbol{x}), \langle j\rangle\big)$, regardless of the value of $\boldsymbol{x}$. Let these identities be $\{\mathsf{id}_1, \ldots, \mathsf{id}_{2^{2\mu/\ell}\cdot\ell}\}$. We can transform the equation to the following:

$$\sum_{i=1}^{2^{2\mu/\ell}\cdot\ell} \mathsf{id}_i \sum_{\substack{\boldsymbol{x}\in B_{\mu-1}, j\in[\ell] \\ \text{s.t.}\, \tilde{\mathsf{p}}\left((X,\boldsymbol{x}),\langle j\rangle\right)=\mathsf{id}_i}} \mathsf{eq}\big((X, \boldsymbol{x}, \langle j\rangle), \boldsymbol{\beta}'\big)$$

In the kth round of sumcheck, the prover needs to compute the univariate polynomial

$$\sum_{\boldsymbol{x}\in B_{\mu-k}, j\in[\ell]} \mathsf{eq}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}, \langle j\rangle), \boldsymbol{\beta}'\big)\tilde{\mathsf{p}}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}), \langle j\rangle)$$

where $\boldsymbol{\gamma} \in \mathbb{F}^{k-1}$ are the challenges from the previous rounds. The number of polynomial identities of each univariate $\tilde{\sigma}_{[\mu]}$ now grows to 2^{2^k}, which implies the number of distinct identities for $\tilde{\mathsf{p}}$ grows to $2^{2^k\mu/\ell}\cdot\ell$. Let these identities be $\{\mathsf{id}_1, \ldots, \mathsf{id}_{2^{2^k\mu/\ell}\cdot\ell}\}$. Now we transform the equation to:

$$\sum_{i=1}^{2^{2^k\mu/\ell}\cdot\ell} \mathsf{id}_i \sum_{\substack{\boldsymbol{x}\in B_{\mu-k}, j\in[\ell] \\ \text{s.t.} \tilde{\mathsf{p}}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}), \langle j\rangle\big)=\mathsf{id}_i}} \mathsf{eq}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}, \langle j\rangle), \boldsymbol{\beta}\big)$$

Using the bucketing algorithm (Algorithm 6), the cost of computing the prover message in the kth round is $(\mu/\ell + 1)(\mu/\ell + 2)2^{2^k\mu/\ell}\cdot\ell = O(\mu^2/\ell)\cdot 2^{2^k\mu/\ell}$ multiplications and $\ell \cdot 2^{\mu+\log\ell-k}$ additions.

Only bucketing for the first $k < \log\ell$ rounds. The exponential growth of the number of possible polynomial identities motivates switching from the bucketing algorithm back to direct computation in round $k' = \log\ell$, at which point the number of field multiplications using the bucketing algorithm exceeds the one using direct computation.

Cost of switching algorithm At the start of round $\log\ell$, the prover needs to compute the evaluation tables of $\tilde{\sigma}\big(\langle i\rangle, (\boldsymbol{\gamma}, \boldsymbol{x})\big)$ for every $i \in [\mu]$ in order to continue the protocol. Using a similar bucketing algorithm, this can be accomplished using $2^{2^{k'}}\sum_{k=1}^{k'} 2^k < \ell\cdot 2^\ell$ field operations. The details of this algorithm can be found in Appendix B, Algorithm 11.

Proof. The number and size of prover messages, the proof size, and the number of verifier queries can be found trivially in the algorithm.

Soundness The formulation for this sumcheck has $\mu + \log\ell$ variates and at most degree $\mu/\ell + 1$ in each variable. The sumcheck therefore has soundness error $\frac{(\mu/\ell+1)(\mu+\log\ell)}{|\mathbb{F}|}$.

Prover Complexity *Field additions* Whether we apply the bucketing algorithm or not, the number of field additions required in the kth round of the second sumcheck is always $\ell\cdot 2^{\mu+\log\ell-k}$. Therefore the total number of field additions for computing the prover messages is

$$\sum_{k=1}^{\mu+\log\ell} \ell\cdot 2^{\mu-k} = n\ell\cdot\sum_{k=1}^{\mu+\log\ell} 2^{-k} = n\ell(1-\frac{1}{n\ell}) = n\ell - 1 < n\ell$$

Algorithm 6 Bucketing Algorithm for Efficiently Computing the Univariate Prover Message in the kth round of the Second Sumcheck in $\mathsf{MulPerm}$

```
1: procedure BUCKET($\tilde{\sigma}_{[\mu]}, \boldsymbol{\beta}' \in \mathbb{F}^{\mu+\log \ell}, \boldsymbol{\gamma} \in \mathbb{F}^{k-1}, k$)
2:     Let $S$ be the set of all $2^k\mu/\ell$-bit strings.
3:     Initialize $\ell$ tables $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_\ell$, each of two columns and $2^{2^k\mu/\ell}$ rows. Note that $S$
   naturally corresponds to the row indices of the tables.
4:     for $j \in [\ell]$, $\boldsymbol{s} \in S$ do
5:         for $i \in [\mu/\ell - 1]$ do
6:             $\boldsymbol{b} := (b_0, b_1, \ldots, b_{2^k}) \leftarrow \boldsymbol{s}[2^k(i-1)+1 : 2^k i]$.
7:             View $\boldsymbol{b}$ as the evaluation table of a $k$-variate polynomial. Fold $\boldsymbol{b}$ such that
   the first $k-1$ variates are replaced by $\boldsymbol{\gamma}$. $\mathsf{id}_\sigma$ is the final univariate polynomial.
8:             Univariate polynomial $\mathsf{id} \leftarrow \mathsf{id} \cdot \mathsf{eq}(\boldsymbol{\alpha}(\langle j' + i \rangle), \mathsf{id}_\sigma)$
9:         end for
10:        Fill the cell $\boldsymbol{t}_j[\boldsymbol{s}, 1]$ with $\mathsf{id}$
11:    end for
12:    for $\boldsymbol{x} \in B_{\mu-k}, j \in [\ell]$ do
13:        Initialize $\boldsymbol{s}$ as an empty string
14:        for $i \in [\mu/\ell], \boldsymbol{x}_L \in B_k$ do
15:            Lookup $\tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, \boldsymbol{x}_L || \boldsymbol{x})$ and append to $\boldsymbol{s}$
16:        end for
17:        Add $\boldsymbol{x}$ to the cell $\boldsymbol{t}_j[\boldsymbol{s}, 2]$
18:    end for
19:    $u_k(X) \leftarrow 0$
20:    for $\boldsymbol{s} \in S, j \in [\ell]$ do
21:        $\mathsf{id} \leftarrow T_j[\boldsymbol{s}, 1]$
22:        $u_{\mathsf{id}}(X) \leftarrow 0$
23:        for $\boldsymbol{x} \in \boldsymbol{t}_j[\boldsymbol{s}, 2]$ do
24:            $u_{\mathsf{id}}(X) \leftarrow u_{\mathsf{id}}(X) + \mathsf{eq}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}, \langle j \rangle), \boldsymbol{\beta}'\big)$
25:        end for
26:        $u_k(X) \leftarrow u_k(X) + \mathsf{id} \cdot u_{\mathsf{id}}(X)$
27:    end for
28:    return $u_k(X)$
29: end procedure
```

Algorithm 7 Second Sumcheck PIOP in $\mathsf{MulPerm}$

```
procedure SUMCHECK2 VERIFIER($[[\tilde{\sigma}_{[\mu]}]], \boldsymbol{\beta}', S$)
    $\boldsymbol{\gamma} \leftarrow []$
    for $k \leftarrow 1 \dots \mu + \log \ell$ do
        Receive $[[u_k]]$ from $\mathsf{P}$
        if $S \neq u_k(0) + u_k(1)$ then
            reject
        end if
        Randomly select $\gamma_k \leftarrow\!\!\$\; \mathbb{F}$. Append to $\boldsymbol{\gamma}$. Send $\gamma_k$ to $\mathsf{P}$
        $S \leftarrow u_k(\gamma_k)$
    end for
    $\boldsymbol{x}^* \leftarrow \gamma[: \mu]$,  $\boldsymbol{j}^* \leftarrow \gamma[\mu + 1 :]$
    Batch-query $[[\tilde{\sigma}_{[\mu]}]]$ to get $V_i := \tilde{\sigma}\big(\boldsymbol{j}^* \cdot \langle \mu/\ell \rangle + \langle i \rangle, \boldsymbol{x}^*\big)$ for every $i \in [\mu/\ell]$
    if $S \neq \prod_{i \in [\mu/\ell]} \mathsf{eq}\big(\boldsymbol{\alpha}(\boldsymbol{j}^* \cdot \langle \mu/\ell \rangle + \langle i \rangle), V_i\big)$ then
        reject
    end if
    accept
end procedure

procedure SUMCHECK2 PROVER($\tilde{\sigma}_{[\mu]}, \boldsymbol{\beta}'$)
    Preprocessed evaluation tables of $\tilde{\sigma}_{[\mu]}(\langle i \rangle, \boldsymbol{x})$ over $B_\mu$ for every $i \in [\mu]$
    $\boldsymbol{\gamma} \leftarrow []$
    for $k \leftarrow 1 \dots \log \ell - 1$ do
        $u_k(X) \leftarrow$ BUCKET($\tilde{\sigma}_{[\mu]}, \boldsymbol{\beta}', \boldsymbol{\gamma}, k$)          ▷ Using Bucketing Algorithm
        Send $[[u_k]]$ to $\mathsf{V}$
        Receive $\gamma_k$ from $\mathsf{V}$ and append it to $\boldsymbol{\gamma}$
    end for
    Evaluation tables of $\tilde{\sigma}_{[\mu]}\big(\langle i \rangle, (\boldsymbol{\gamma}, \boldsymbol{x})\big)$ $\forall i \in [\mu] \leftarrow$ COLLAPSE($\tilde{\sigma}_{[\mu]}, \boldsymbol{\gamma}, \log \ell - 1$)
    for $k \leftarrow \log \ell + 1, \dots, \mu + \log \ell$ do
        $u_k(X) \leftarrow 0$
        for $\boldsymbol{x}' \in B_{\mu + \log \ell - k}$ do
            $u_{\boldsymbol{x}}(X) \leftarrow \mathsf{eq}\big((\boldsymbol{\gamma}, X, \boldsymbol{x}'), \boldsymbol{\beta}'\big)\mathsf{p}(\boldsymbol{\gamma}, X, \boldsymbol{x}')$          ▷ Using FFT
            $u_k(X) \leftarrow u_k(X) + u_{\boldsymbol{x}}(X)$
        end for
        Send $[[u_k]]$ to $\mathsf{V}$
        Receive $\gamma_k$ from $\mathsf{V}$ and append it to $\boldsymbol{\gamma}$
        Fold the evaluation tables
    end for
end procedure
```

Computing the evaluation tables after round $\log \ell - 1$ takes at most $\ell \cdot 2^\ell$ field additions. Thus, the total number of field additions is fewer than $n\ell + \ell 2^\ell$.
Field multiplications The number of field multiplications for the first $\log \ell - 1$ rounds is

$$(\mu/\ell + 1)(\mu/\ell + 2) \sum_{k=1}^{\log \ell - 1} 2^{2^k \mu/\ell} \cdot \ell = n \cdot O(\mu^2/\ell)$$

Computing the evaluation tables after round $\log \ell - 1$ takes fewer than $\ell \cdot 2^\ell$ field operations.
The number of field operations for all rounds $k \geq \log \ell$ is

$$\begin{aligned}
&\sum_{k=k'+1}^{\mu+\log \ell} (\mu/\ell + 2)^2 \frac{n\ell}{2^k} \text{ assume } \mu \geq 5\ell \\
&\leq 2(\mu/\ell)^2 \cdot n\ell \sum_{k=k'+1}^{\mu+\log \ell} 2^{-k} \\
&= \frac{2n\mu^2}{\ell} \cdot 2^{-k'} \cdot (1 - \frac{1}{n\ell}) \\
&< \frac{2n\mu^2}{\ell^2} = n \cdot O(\mu^2/\ell^2)
\end{aligned}$$

The number of field multiplications can be reduced to $n \cdot \tilde{O}(\mu/\ell)$ if prover employs FFT for polynomial computation.
Thus, the total number of field operations is $n \cdot \tilde{O}(\mu/\ell) + \ell 2^\ell$.
Query Complexity Verifier queries the oracle of $\tilde{\sigma}_{[\mu]}$ once.

7.4 Setting the group parameter ℓ

The total number of field operations for MulPerm is

$$\underbrace{n \cdot \tilde{O}(\ell)}_{\text{First Sumcheck}} + \underbrace{n \cdot \tilde{O}(\mu/\ell) + \ell 2^\ell}_{\text{Second Sumcheck}} = n \cdot \tilde{O}(\ell + \mu/\ell) + \ell 2^\ell$$

Setting $\ell = \sqrt{\mu}$ achieves the balance between the two terms inside $\tilde{O}$ and $\ell 2^\ell = \sqrt{\mu} 2^{\sqrt{\mu}} = o(n)$. Then MulPerm has an overall $n \cdot \tilde{O}(\sqrt{\log n})$ number of field operations and soundness error of

$$\begin{aligned}
&\Big(\underbrace{\mu}_{\text{Schwartz-Zippel}} + \underbrace{(\sqrt{\mu} + 1)(\mu + \log \sqrt{\mu})}_{\text{First Sumcheck}} + \underbrace{(\mu/\sqrt{\mu} + 1)(\mu + \log \sqrt{\mu})}_{\text{Second Sumcheck}} \Big) / |\mathbb{F}| \\
&= \frac{O(\mu^{1.5})}{|\mathbb{F}|} = \frac{\text{polylog}(n)}{|\mathbb{F}|}
\end{aligned}$$

7.5 Compiling MulPerm using PCS

MulPerm can be compiled using *any* PCS to yield an almost linear-time permutation argument, since all the oracles are $O(n)$-sized.

8 Prover-provided permutation

BiPerm and MulPerm assume the MLE of permutation σ (or $\sigma_{[\mu]}$ in MulPerm) is preprocessed. This is consistent with some applications such as proving the wiring identity of circuits. However, in other applications such as memory-checking, the two permuted tables are dynamic and cannot be determined until the program finishes running; the mapping σ therefore cannot be preprocessed and is provided by the prover during runtime. In this section, we present the details of how to extend MulPerm to such scenarios, and briefly discuss how to extend BiPerm in the last subsection (Section 8.3).

Theorem 6. *Let $n = 2^\mu$. Given $f, g \in \mathcal{F}_\mu^{(\leq 1)}$, the triple-sumcheck PIOPs for Equation 3, Equation 10 and Equation 11 described by the prover and verifier in Algorithm 8 proves $([[f]], [[g]]; f, g) \in \mathcal{R}_{\mathsf{PERM2}}$ with the following properties:*
- *Perfectly complete.*
- *Oracles: no oracle in the index. Two oracles in the instance, each of size n and not sparse. $O(\log n)$ oracles sent during runtime, which are $[[\tilde{\sigma}_{[\mu]}]], [[\tilde{\tau}_{[\mu]}]]$ and oracles to prover messages; and $[[\tilde{\tau}_{[\mu]}]]$ are size $n \log n$ and maps to $\{0, 1\}$, and the oracles to prover messages of size $O(\sqrt{\log n})$.*
- *Soundness error $O(\text{polylog}\, n/|\mathbb{F}|)$*
- *Proof size $O(\log n)$*
- *Prover Complexity is $n \cdot \tilde{O}(\sqrt{\log n})$.*
- *Verifier queries $[[f]], [[g]]$, queries $[[\tilde{\sigma}_{[\mu]}]]$ once and later at $\sqrt{\log n}$ positions in a batched fashion, and each of the oracles of the prover messages.*

We define the unindexed permutation relation with prover-provided σ in Definition 10.

Definition 10 (Permutation relation, unindexed). *The relation $\mathcal{R}_{\mathsf{PERM2}}$ is the set of tuples*

$$(\mathbb{x}; \mathbb{w}) = \left([[f]], [[g]]; f, g\right),$$

where $f, g \in \mathcal{F}_\mu^{(\leq 1)}$, such that there exists a permutation $\sigma : B_\mu \to B_\mu$ and $f(\boldsymbol{y}) = g(\sigma(\boldsymbol{y}))$ for all $\boldsymbol{y} \in B_\mu$.

Previously, the preprocessing of $\tilde{\sigma}_{[\mu]}$ waives the need to prove that σ is a permutation. Now, using BiPerm or MulPerm for this scenario, the prover needs to first compute the permutation σ between f, g, and additionally prove that σ is indeed a permutation, i.e. there exists $\sigma^{-1} : B_\mu \mapsto B_\mu$ such that $\sigma^{-1}(\sigma(\boldsymbol{y})) = \boldsymbol{y}$ over B_μ.

8.1 Proving $\boldsymbol{\sigma}$ is a permutation

Lemma 8. *Given a function σ with pre-image space B_μ, $\tau(\sigma(\boldsymbol{y})) = \boldsymbol{y}$ for every $\boldsymbol{y} \in B_\mu$ and the image space of σ is B_μ if and only if $\sigma : B_\mu \to B_\mu$ is a permutation and $\tau = \sigma^{-1}$.*

Algorithm 8 PIOP for $\mathcal{R}_{\mathsf{PERM2}}$.

```
1: procedure PERM2 Verifier($[[f]], [[g]]$)
2:     Receive $[[\tilde{\sigma}_{[\mu]}]], [[\tilde{\tau}_{[\mu]}]]$ from $\mathsf{P}$
3:     Ramdomly select $r \leftarrow\!\!\$\ \mathbb{F}$ and send $r$ to $\mathsf{P}$
4:     Define $[[f']], [[g']]$ such that $f'(\boldsymbol{y}) = \boldsymbol{y} + r \cdot f(\boldsymbol{y})$ and $g'(\boldsymbol{y}) = \tilde{\tau}(\boldsymbol{y}) + r \cdot g(\boldsymbol{y})$ for
   every $\boldsymbol{y} \in B_\mu$
5:     MulPerm Verifier($[[\tilde{\sigma}_{[\mu]}]], [[f']], [[g']]$).
6:     BinMap Verifier($[[\tilde{\sigma}_{[\mu]}]]$)
7: end procedure
8:
9: procedure PERM2 Prover($f, g$)
10:    Compute $\sigma$ such that $f(\boldsymbol{y}) = g(\sigma(\boldsymbol{y}))$ for every $\boldsymbol{y} \in B_\mu$. Define $\tau := \sigma^{-1}$
11:    Interpolate $\sigma, \tau$ into $(\mu + \log\mu)$ variate polynomials $\sigma_{[\mu]}, \tau_{[\mu]}$, respectively. Com-
   pute the MLE $\tilde{\sigma}_{[\mu]}, \tilde{\tau}_{[\mu]}$ and send oracle $[[\tilde{\sigma}_{[\mu]}]], [[\tilde{\tau}_{[\mu]}]]$ to $\mathsf{V}$
12:    Receive random challenge $r \in \mathbb{F}$ from $\mathsf{V}$
13:    Define $f'$ such that $f'(\boldsymbol{y}) = \boldsymbol{y} + r \cdot f(\boldsymbol{y})$
14:    MulPerm Prover($\tilde{\sigma}_{[\mu]}, f'$)
15:    BinMap prover($\tilde{\sigma}_{[\mu]}$)
16: end procedure
```

Proof. The forward direction is trivial. We prove the backward direction below. Since $\tau(\sigma(\boldsymbol{y})) = \boldsymbol{y}$ for every $\boldsymbol{y} \in B_\mu$, it must be that $\sigma : B_\mu \mapsto \mathbb{H}$ and $\tau : \mathbb{H} \mapsto B_\mu$ for some space $\mathbb{H}$. Since the image space of σ is B_μ, $\mathbb{H} = B_\mu$ and $\sigma : B_\mu \mapsto B_\mu$. Since the pre-image space and the image space of σ is the same, it must be a permutation, and $\tau = \sigma^{-1}$.

In order to prove this, the prover provides an oracle to $\tilde{\tau}_{[\mu]}$ which is supposed to be $\tilde{\sigma}_{[\mu]}^{-1}$, and proves that 1. $\tau(\sigma(\boldsymbol{y})) = \boldsymbol{y}$ for every $\boldsymbol{y} \in B_\mu$ and 2. σ maps to binaries over B_μ.

The first task can be reduced to the following sumcheck through similar process as described in Section 4.6,

$$\sum_{\boldsymbol{x} \in B_\mu} \boldsymbol{x} \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{\alpha}) = \tilde{\tau}(\boldsymbol{\alpha}) \tag{10}$$

The second task is the same as proving $\tilde{\sigma}_{[\mu]}(i, \boldsymbol{x}) \in \{0, 1\}$ for every $\boldsymbol{x} \in B_\mu, i \in [\mu]$, which can be trivially reduced to the following sumcheck using random challenges $\boldsymbol{s} \in B_{\mu + \log\mu}$,

$$\sum_{\boldsymbol{x} \in B_\mu, i \in [\mu]} \mathsf{eq}\big((\boldsymbol{x}, \langle i \rangle), \boldsymbol{s}\big) \cdot \underbrace{\tilde{\sigma}_{[\mu]}(i, \boldsymbol{x})(1 - \tilde{\sigma}_{[\mu]}(i, \boldsymbol{x}))}_{h(\boldsymbol{x}, i)} = 0 \tag{11}$$

Now we prove the completeness and soundness parts of Theorem 6.

Proof.

Completeness Completeness follows from the fact that there is a bidirectional implication between each step in the reductions to sumcheck formulations.

Knowledge Soundness We know from Lemma 4 that using sumcheck PIOP for Equation 3 incurs soundness error $O(\log^2 n/|\mathbb{F}|)$. By similar reasoning, the sumcheck PIOP for Equation 10 also incurs soundness error $O(\log^2 n/|\mathbb{F}|)$. The sumcheck for Equation 11 is over a virtual polynomial that has $\mu+\log\mu$ variables and individual degree at most 3, so its soundness error is upper bounded by $3(\mu+\log\mu)/|\mathbb{F}| = O(\log n/|\mathbb{F}|)$. Thus, the total soundness error is $O(\log^2 n/|\mathbb{F}|)$.

Remark 2. Using a random linear permutation, proving $\tau(\sigma(\boldsymbol{y})) = \boldsymbol{y}$ for every $\boldsymbol{y} \in B_\mu$ can be batched using with the usual permutation check which proves that $f(\boldsymbol{y}) = g(\sigma(\boldsymbol{y}))$ for every $\boldsymbol{y} \in B_\mu$. We refer to [CBBZ23] for the proof of this technique. Specifically, define

$$f'(\boldsymbol{y}) = \boldsymbol{y} + R \cdot f(\boldsymbol{y}), \quad g'(\boldsymbol{y}) = \tau(\boldsymbol{y}) + R \cdot g(\boldsymbol{y}) \quad \forall \boldsymbol{y} \in B_\mu$$

and prove $f'(\boldsymbol{y}) = g'(\sigma(\boldsymbol{y}))$ for every $\boldsymbol{y} \in B_\mu$. In Algorithm 8, we describe this optimized algorithm.

8.2 Improving the prover cost when proving $\boldsymbol{\sigma}$ maps to binaries

If the sumcheck for Equation 11 is computed naïvely, the prover cost is $\sum_{k=1}^{\mu} 2^{\mu-k}\mu + \sum_{k=1}^{\log\mu} 2^{\log\mu-k} = 2^\mu\mu - \mu + \mu - 1 = n\mu - 1$ multiplications, which is too expensive. We discuss ways to improve this sumcheck.

The prover wants to prove

$$\sum_{\boldsymbol{x}\in B_\mu, i\in[\mu]} \mathsf{eq}\big((\boldsymbol{x}, \langle i\rangle), \boldsymbol{s}\big) \cdot \underbrace{\tilde{\sigma}_{[\mu]}(i,\boldsymbol{x})(1-\tilde{\sigma}_{[\mu]}(i,\boldsymbol{x}))}_{h(\boldsymbol{x},i)} = 0$$

Let $\mu' := \mu + \log\mu$. For convenience of notation, we rewrite the above as,

$$\sum_{\boldsymbol{x}'\in B_{\mu'}} \mathsf{eq}(\boldsymbol{x}', \boldsymbol{s}) \cdot h(\boldsymbol{x}') = 0 \tag{12}$$

In the k-th round of this sumcheck, where $\boldsymbol{\alpha} \in \mathbb{F}^{k-1}$ are challenges from the previous rounds, the prover needs to compute the univariate prover message

$$\sum_{\boldsymbol{x}'\in B_{\mu'-k}} \mathsf{eq}((\boldsymbol{\alpha}, X, \boldsymbol{x}'), \boldsymbol{s}) h(\boldsymbol{\alpha}, X, \boldsymbol{x}')$$

Computing the k-th round directly takes $2^{\mu'-k}$ field additions and $8\cdot 2^{\mu'-k}$ field multiplications.[6]

[6] The factor 8 comes from the fact that the prover message is degree 3 in each entry of $\boldsymbol{x}$ and is the product of three smaller polynomials. It therefore needs to be computed by multiplying together three sets of evaluation points, each containing four points.

Prover buckets in their head Consider what happens if we fold b variants into h, then for $\boldsymbol{x}' \in B_{\mu'-1}$ we have

$$h^{(1)}(\boldsymbol{x}', \boldsymbol{s}_\mu) = h(\boldsymbol{x}'||0)\mathsf{eq}(0, \boldsymbol{s}_\mu) + h(\boldsymbol{x}'||1)\mathsf{eq}(1, \boldsymbol{s}_\mu)$$

for $\boldsymbol{x}' \in B_{\mu'-b}, \boldsymbol{s}'' = \boldsymbol{s}_{[:-b]}$ we have

$$h^{(b)}(\boldsymbol{x}', \boldsymbol{s}'') = h^{(b-1)}(\boldsymbol{x}'||0, \boldsymbol{s}''_{[1:]})\mathsf{eq}(0, \boldsymbol{s}''_{[-1]}) + h^{(b-1)}(\boldsymbol{x}'||1, \boldsymbol{s}''_{[1:]})\mathsf{eq}(1, \boldsymbol{s}''_{[-1]})$$

Replace Equation 12 with $h^{(b)}$:

$$\sum_{\boldsymbol{x}' \in B_{\mu'-b}} \mathsf{eq}(\boldsymbol{x}', \boldsymbol{s}_{[:\mu'-b]})h^{(b)}(\boldsymbol{x}', \boldsymbol{s}_{[:-b]})$$

The k-th round then becomes

$$\sum_{\boldsymbol{x}' \in B_{\mu'-b-k}} \mathsf{eq}\big((\boldsymbol{\alpha}, X, \boldsymbol{x}'), \boldsymbol{s}_{[:\mu'-b]}\big)h^{(b)}\big((\boldsymbol{\alpha}, X, \boldsymbol{x}'), s_{[:-b]}\big)$$

As discussed in Section 7.3, the univariate $\tilde{\sigma}_{[\mu]}$ polynomial has 2^{2^k} possible identities in the kth round, therefore so does h, and the number of possible identities for $h^{(b)}$ is $2^{2^{b+k}}$. Using a bucketing algorithm similar to Algorithm 6, the prover compute the round polynomial by computing the eq partial sum corresponding to each one of the $2^{2^{b+k}}$ possible polynomial taken on by $h^{(b)}$, and then sum up across all those buckets. The number of additions is $2^{\mu'-k-b}$, and the number of multiplications is $8 \cdot 2^{2^{b+k}}$.

Lemma 9. *Let $n = 2^\mu$. The sumcheck PIOP for Equation 11 can be computed using $n \cdot o(\log\log n)$ field additions and $o(n)$ field multiplications. The proof size is $O(\log\log n)$, and the verifier queries $[[\tilde{\sigma}_{[\mu]}]]$ once.*

Proof. Set $b := \log\mu' - k - 2$ and use the above algorithm in every round $k \leq \log\mu'$. Then in each of these rounds, prover performs $2^{\mu'-k-(\log\mu'-k-2)} = 2^{\mu'-\log\mu'+2}$ field additions and $8 \cdot 2^{2^{\log\mu'-k-2+k}} = 8 \cdot 2^{(\mu+\log\mu)/4} = 8(n\mu)^{1/4}$ field multiplications. Thus, the cost of the first $k \leq \log\mu'$ rounds is $2^{\mu'-\log\mu'+2} \cdot \log\mu' = \frac{4n\mu\log\mu'}{\mu'} = O(n\log\mu') = O(n\log\log n)$ additions, and $8(n\mu)^{1/4} \cdot \log\mu' = 8(n\mu)^{1/4} \cdot \log(\mu + \log\mu) = o(n)$ multiplications.

The number of field operations in the remaining rounds is $\sum_{k=\log\mu'+1}^{\mu'} 2^{\mu'-k} = 2^{\mu'-\log\mu'} - 1 = o(n)$. Putting together the cost of this protocol, the total number of field operations is $O(n\log\log n)$.

Remark 3. For sufficiently large n, the prover cost of running $\mathsf{MulPerm}$ dominates in proving prover-provided permutation.

Algorithm 9 PIOP for proving a multilinear polynomial $\tilde{\sigma}_{[\mu]}$ maps to binaries in $\mathcal{R}_{\mathsf{PERM2}}$ with $\mathsf{MulPerm}$. $\mu' = \mu + \log\mu$, and $h(\boldsymbol{x}, i) = \tilde{\sigma}_{[\mu]}(i, \boldsymbol{x})(1 - \tilde{\sigma}_{[\mu]}(i, \boldsymbol{x}))$. The algorithm of BinMapBucket is described in Section 8.2

```
1: procedure BinMap Verifier($[[\tilde{\sigma}_{[\mu]}]]$)
2:     Randomly select $\boldsymbol{s} \leftarrow\!\!\$\ \mathbb{F}^{\mu'}$. Send $\boldsymbol{s}$ to $\mathsf{P}$
3:     $S \leftarrow 0$
4:     $\boldsymbol{\alpha} \leftarrow []$
5:     for $k = 1 \ldots \mu'$ do
6:         Receive $[[u_k]]$ from $\mathsf{P}$
7:         if $S \neq u_k(0) + u_k(1)$ then
8:             reject
9:         end if
10:        Randomly select $\alpha_k \leftarrow\!\!\$\ \mathbb{F}$. Append to $\boldsymbol{\alpha}$. Send $\alpha_k$ to $\mathsf{P}$
11:    end for
12:    Query $[[\tilde{\sigma}_{[\mu]}]]$ to get $V := \tilde{\sigma}(\boldsymbol{\gamma}[:\mu], \boldsymbol{\gamma}[\mu+1:])$
13:    if $S \neq \mathsf{eq}(\boldsymbol{\gamma}, \boldsymbol{s}) \cdot V(1-V)$ then
14:        reject
15:    end if
16:    accept
17: end procedure
18:
19: procedure BinMap Prover($\tilde{\sigma}_{[\mu]}$)
20:    Receive $\boldsymbol{s} \in \mathbb{F}^{\mu'}$ from $\mathsf{V}$
21:    $\boldsymbol{\alpha} \leftarrow []$
22:    for $k \leftarrow 1 \ldots \log\mu'$ do
23:        $b \leftarrow \log\mu' - k - 2$
24:        $u_k(X) \leftarrow$ BinMapBucket($\tilde{\sigma}_{[\mu]}, b, \boldsymbol{s}, \boldsymbol{\alpha}$)
25:        Send $[[u_k]]$ to $\mathsf{V}$
26:        Receive $\alpha_k$ from $\mathsf{V}$ and append it to $\boldsymbol{\alpha}$
27:    end for
28:    Evaluation tables of $\tilde{\sigma}_{[\mu]}(\langle i \rangle, (\boldsymbol{\alpha}, \boldsymbol{x}))\ \ \forall i \in [\mu] \leftarrow$ Collapse($\tilde{\sigma}_{[\mu]}, \alpha, \log\mu'$)
29: end procedure
30: for $k \leftarrow \log\mu' \ldots \mu'$ do
31:    $u_k(X) \leftarrow 0$
32:    for $\boldsymbol{x} \in B_{\mu'-k}$ do
33:        $u_{\boldsymbol{x}}(X) \leftarrow \mathsf{eq}\big((\boldsymbol{\alpha}, X, \boldsymbol{x}), \boldsymbol{s}\big) \cdot h(\boldsymbol{\alpha}, X, \boldsymbol{x})$
34:        $u_k(X) \leftarrow u_k(X) + u_{\boldsymbol{x}}(X)$
35:    end for
36:    Send $[[u_k]]$ to $\mathsf{V}$
37:    Receive $\alpha_k$ from $\mathsf{V}$ and append it to $\boldsymbol{\alpha}$
38:    Fold the evaluation tables
39: end for
```

8.3 Extending BiPerm to prover-provided permutation

If BiPerm is used, then we can use the algorithm described in Section 4.1.2 of Twist and Shout [ST25] to check the booleanity of $\tilde{\mathbb{1}}_L, \tilde{\mathbb{1}}_R$, for which the prover needs to perform $O(n) + 4\sqrt{n} = O(n)$ field operations. This algorithm is only compatible with BiPerm and not MulPerm because it requires sparse PCS for compilation and that the arithmetization of the indicator function number must be the product of only a constant number of functions. The resulting PIOP for prover-provided permutation has $O(n)$ prover complexity.

9 Generalization to Lookups

Definition 11 (Lookup relation, indexed). *The indexed relation $\mathcal{R}_{\mathsf{LKUP}}$ is the set of tuples*

$$(\mathbb{i}; \mathbb{x}; \mathbb{w}) = \big(\tilde{\rho}, [[\tilde{\rho}]]; [[f]], [[g]]; f, g\big)$$

where $\tilde{\rho} \in \mathcal{F}_{\mu}^{(\leq 1)}$ is the multilinear extension of map $\rho \in \mathsf{FUN}(B_\mu, B_\kappa)$, $f \in \mathcal{F}_{\kappa}^{(\leq 1)}$, $g \in \mathcal{F}_{\mu}^{(\leq 1)}$, and

$$f(\rho(\boldsymbol{x})) = g(\boldsymbol{x}) \;\; \forall \boldsymbol{x} \in B_\mu \tag{13}$$

We note that instead of $\tilde{\rho}$, the index may contain polynomials that make up $\tilde{\rho}$ of different forms of this map.

Intuitively, f is the multilinear polynomial representing the κ-sized lookup table, and g is the multilinear polynomial representing the μ-sized witness vector. Lookup can be viewed in a similar light as permutation check. In fact, one can think of permutation check as a special case of lookup, where the lookup function ρ maps between hypercubes of the same dimension.

Similar to before, we define multilinear version of ρ as

$$\tilde{\rho} = (\tilde{\rho}_1(\boldsymbol{X}), \ldots, \tilde{\rho}_\kappa(\boldsymbol{X})) : \mathbb{F}^\mu \mapsto \mathbb{F}^\kappa$$

with $\tilde{\rho}_i(\boldsymbol{x}) \in \{0, 1\} \;\; \forall \boldsymbol{x} \in B_\mu \;\; \forall i \in [\kappa]$. Note here $\mathbb{F}^\kappa$ is the range, but the actual image space of $\tilde{\rho}$ can be much smaller. Similar to $\tilde{\sigma}_{[\mu]}$ in Section 7, we can also interpolate $\tilde{\rho}$ as a $(\log \kappa + \mu)$-variate multilinear polynomial $\tilde{\rho}_{[\kappa]} : \mathbb{F}^{\log \kappa + \mu} \mapsto \mathbb{F}$, such that $\tilde{\rho}_{[\kappa]}(\langle i \rangle, \boldsymbol{X}) = \tilde{\rho}_i(\boldsymbol{X})$ for every $i \in [\kappa]$.

Reducing Lookup to Sumcheck Similar to what we do in permutation check, we consider the lookup mapping as a relation, and its indicator function $\mathbb{1}_\rho : B_\mu \times B_\kappa \mapsto B$,

$$\mathcal{R}_\rho = \{(\boldsymbol{x}, \boldsymbol{y}) : \rho(\boldsymbol{x}) = \boldsymbol{y}\} \qquad \mathbb{1}_\rho(\boldsymbol{X}, \boldsymbol{Y}) = \begin{cases} 1 & (\boldsymbol{X}, \boldsymbol{Y}) \in \mathcal{R}_\rho \\ 0 & (\boldsymbol{X}, \boldsymbol{Y}) \notin \mathcal{R}_\rho \end{cases}$$

Following the same steps we took when reducing the permutation check to sumcheck formulation, we can obtain a similar sumcheck formulation for lookup, where $\tilde{\mathbb{1}}_\rho$ is some arithmetization of the indicator function that is multilinear in $\boldsymbol{y}$:

$$\sum_{\boldsymbol{y}\in B_\kappa} f(\boldsymbol{y}) \cdot \tilde{\mathbb{1}}_\rho(\boldsymbol{x},\boldsymbol{y}) = g(\boldsymbol{x}) \;\; \forall \boldsymbol{x} \in B_\mu$$

Unfortunately, we cannot apply Schwartz-Zippel this time. As we previously noted, arithmetizations of $\tilde{\mathbb{1}}_\rho$ that is multilinear in $\boldsymbol{x}$ are expensive to deal with; moreover, the lookup function ρ is not injective and therefore does not have an inverse to work with. To get around this, we need to add an outer sum to the formulation, where $\boldsymbol{s} \in \mathbb{F}^\mu$ are random challenges:

$$\sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x},\boldsymbol{s}) \sum_{\boldsymbol{y}\in B_\kappa} f(\boldsymbol{y})\tilde{\mathbb{1}}_\rho(\boldsymbol{x},\boldsymbol{y}) = \sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x},\boldsymbol{s})g(\boldsymbol{x})$$

We flip the sums to make it more compatible with our techniques.

$$\sum_{\boldsymbol{y}\in B_\kappa} f(\boldsymbol{y}) \sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x},\boldsymbol{s})\tilde{\mathbb{1}}_\rho(\boldsymbol{x},\boldsymbol{y}) = \sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x},\boldsymbol{s})g(\boldsymbol{x})$$

The prover and the verifier will engage in two sumcheck protocols for some $S \in \mathbb{F}$. One for proving

$$\sum_{\boldsymbol{y}\in B_\kappa} f(\boldsymbol{y}) \sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x},\boldsymbol{s})\tilde{\mathbb{1}}_\rho(\boldsymbol{x},\boldsymbol{y}) = S \tag{14}$$

and another one for proving $\sum_{\boldsymbol{x}\in B_\mu} \mathsf{eq}(\boldsymbol{x},\boldsymbol{s})g(\boldsymbol{x}) = S$. The second sumcheck is easy to perform and will hence be omitted in the following discussion. We will focus on the sumcheck for Equation 14.

9.1 Sumcheck PIOP for lookups

In this section, we describe how to extend MulPerm to lookups.

Theorem 7. *Let $n = 2^\mu, T = 2^\kappa$ such that $T < 2^{\mu^{1.8}}$. Given $f \in \mathcal{F}_\kappa^{(\leq 1)}, g \in \mathcal{F}_\mu^{(\leq 1)}$ and preprocessed $\tilde{\rho}_{[\kappa]} : \mathbb{F}^{\log\kappa+\mu} \mapsto \mathbb{F}$, the triple-sumcheck PIOP described in Algorithm 10 that uses MulPerm for proving $\big(\tilde{\rho}_{[\kappa]}, [[\tilde{\rho}_{[\kappa]}]]; [[f]], [[g]]; f, g\big) \in \mathcal{R}_{\mathsf{LKUP}}$ has the following properties:*

- *Oracles: One oracle $[[\tilde{\rho}_{[\kappa]}]]$ in the index, which has size $n\kappa$ and maps to $\{0,1\}$ over $B_{\log\kappa+\mu}$. Two oracles f, g in the instance, of size T, n respectively. $O(\log T + \log n)$ oracles of prover messages sent during runtime, each of size at most $O(\sqrt{\log T})$.*
- *Soundness error $\frac{\mathrm{polylog}(n+T)}{|\mathbb{F}|}$,*
- *Proof size $O(\log T + \log n)$,*
- *Prover performs $n \cdot \tilde{O}(\sqrt{\log T})$ if $T \leq n$ and $O(n(\log T - \log n))$ field operations if $T > n$, as well as computing an evaluation of f.*
- *Verifier queries $[[f]], [[g]]$, queries $[[\tilde{\rho}_{[\kappa]}]]$ once and later at $\sqrt{\log T}$ positions in a batched fashion, and each of the oracles of the prover messages.*

The outer sum In the k-th round of the outer sumcheck, with challenges $\boldsymbol{\alpha} \in \mathbb{F}^{k-1}$, prover needs to compute the following univariate degree-2 polynomial,

$$\sum_{\boldsymbol{y} \in B_{\kappa-k}} f(\boldsymbol{\alpha}, Y, \boldsymbol{y}) \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, (\boldsymbol{\alpha}, Y, \boldsymbol{y})) = S$$

Observe that we do not actually need to compute the terms for every $\boldsymbol{y}$ and $\boldsymbol{x}$, but only those $\boldsymbol{y}$ that is in the image space of ρ and those $\boldsymbol{x}$ that map to these corresponding $\boldsymbol{y}$. Let the image space of ρ be I_ρ, and it is trivial that $|I_\rho| \leq \min(T, n), \log |I_\rho| \leq \min(\mu, \kappa)$. The prover message is equivalent to

$$\sum_{\boldsymbol{y} \in \{I_\rho\}_{[k:]}} f(\boldsymbol{\alpha}, Y, \boldsymbol{y}) \sum_{\boldsymbol{x} \in B_\mu | \rho(\boldsymbol{x})_{[k:]} = \boldsymbol{y}} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, (\boldsymbol{\alpha}, Y, \boldsymbol{y}))$$

Moreover, note that the evaluation table of $\tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{Y})$ is of size T; it is sparse and only non-zero at no more than n different points. As its evaluation table gets folded after each round, its number of non-zero entries may be folded together, decreasing its non-sparsity. The number of field operations required to compute the prover message is directly related to the non-sparsity of $\tilde{\mathbb{1}}_\rho$.

If $T \leq n$, then in the worst case, all entries in the table are non-zero, and the total number of field operations required for the outer sumcheck is at most $\frac{n}{2} + \frac{n}{4} + \cdots + 1 = O(n)$.

If $T > n$, then the worst case is that the table has n non-zero entries and they always get folded with zero entries in the earlier rounds. This implies that the prover needs to perform $O(n)$ field operations in each of these early rounds. Fortunately, the non-sparsity of the table is guaranteed to start decreasing once the table size shrinks to n, which happens after $\log(\frac{T}{n}) = \kappa - \mu$ rounds. Therefore, the total number of field operations required for the outer sumcheck is at most $O(n(\kappa - \mu) + n) = O(n(\kappa - \mu))$.

Remark 4. If the table is highly-structured, we can apply the algorithm in Appendix G.5.1 from Lasso [STW24] to reduce the prover cost of the outer sumcheck to $O(n)$.

After the outer sumcheck, the claim is reduced to

$$f(\boldsymbol{\alpha}) \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \cdot \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S'$$

for some $S' \in \mathbb{F}$ and random challenges $\boldsymbol{\alpha} \in \mathbb{F}^\mu$. The prover and the verifier can then use either BiPerm or $\mathsf{MulPerm}$ to prove

$$\sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \cdot \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S'/f(\boldsymbol{\alpha}) \tag{15}$$

Corollary 1. *Let $n = 2^\mu, T = 2^\kappa$ such that $T < 2^{(1-\epsilon)\cdot\mu^2}$ for a constant $\epsilon > 0$. Given $f, g \in \mathcal{F}_\mu^{(\leq 1)}$, and preprocessed $\tilde{\rho}_{[\kappa]} : B_{\log\kappa+\mu} \mapsto \{0,1\}$, $\mathsf{MulPerm}$ can be used for proving Equation 15 with the following properties:*

- *$O(\log n)$ oracles of prover messages sent during runtime, each of size $O(\sqrt{\log T})$.*
- *Soundness error $\frac{\mathrm{polylog}(n+T)}{|\mathbb{F}|}$,*
- *Proof size $O(\log n)$,*
- *Prover performs $n \cdot \tilde{O}(\sqrt{\log T})$ field operations and one evaluation of f.*
- *Verifier batch-queries $[[\tilde{\rho}_{[\kappa]}]]$ at $\sqrt{\log T}$ positions, and each of the oracles of the prover messages.*

Proof. Let p be any multilinear polynomial, $\rho \in \mathsf{FUN}(B_\mu, B_\kappa)$ is some arbitrary map, and $\tilde{\mathbb{1}}_\rho(\boldsymbol{X}, \boldsymbol{Y})$ is the arithmetization of indicator function for ρ that is multilinear in $\boldsymbol{Y}$. Since *no where* in the $\mathsf{MulPerm}$ algorithm do we take advantage of the fact that the preprocessed mapping is a permutation, it can be applied to proving any sumcheck of the form

$$\sum_{x \in B_\mu} p(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S$$

Thus, it cam be directly used to prove Equation 15.

Using $\mathsf{MulPerm}$, we arithmetize $\tilde{\mathbb{1}}_\rho(\boldsymbol{X}, \boldsymbol{Y})$ as the product of ℓ sub-indicator functions, each of which depend on κ/ℓ bits of $\boldsymbol{Y}$. The total number of field operations is

$$\underbrace{n \cdot \tilde{O}(\ell)}_{\text{First Sumcheck}} + \underbrace{n \cdot \tilde{O}(\kappa/\ell) + \ell 2^\ell}_{\text{Second Sumcheck}} = n \cdot \tilde{O}(\ell + \kappa/\ell) + \ell 2^\ell$$

Setting $\ell = \sqrt{\kappa}$ then achieves the balance between the two terms inside $\tilde{O}$ and since $T < 2^{(1-\epsilon)\mu^2} \implies \kappa < (1-\epsilon)\mu^2)$ we have that $\ell 2^\ell = \sqrt{\kappa} 2^{\sqrt{\kappa}} < \mu \cdot 2^{(1-\epsilon)\mu} = o(n)$. Then $\mathsf{MulPerm}$ has an overall $n \cdot \tilde{O}(\sqrt{\log T})$ number of field operations. With this parameterization the soundness error is

$$\Big(\underbrace{(\sqrt{\kappa}+1)(\mu + \log\sqrt{\kappa})}_{\text{First Sumcheck}} + \underbrace{(\kappa/\sqrt{\kappa}+1)(\mu + \log\sqrt{\kappa})}_{\text{Second Sumcheck}} \Big)/|\mathbb{F}|$$
$$= \frac{O(\mu\sqrt{\kappa})}{|\mathbb{F}|} = \frac{\mathrm{polylog}(n+T)}{|\mathbb{F}|}$$

9.2 Prover-provided lookup reduced to sumcheck

Definition 12 (Lookup relation, unindexed). *The unindexed relation $\mathcal{R}_{\mathsf{LKUP2}}$ is the set of tuples*

$$(\mathbb{x}; \mathbb{w}) = \big([[f]], [[g]]; f, g\big)$$

where $f \in \mathcal{F}_\kappa^{(\leq 1)}$, $g \in \mathcal{F}_\mu^{(\leq 1)}$, such that there exists a mapping $\rho \in \mathsf{FUN}(B_\mu, B_\kappa)$ and

$$f(\rho(\boldsymbol{x})) = g(\boldsymbol{x}) \ \ \forall \boldsymbol{x} \in B_\mu \tag{16}$$

Algorithm 10 PIOP for $\mathcal{R}_{\mathsf{LKUP}}$.

```
1: procedure LOOKUP VERIFIER($[[f]], [[g]]$)
2:     Receive $[[\tilde{\rho}_{[\kappa]}]]$ from $\mathsf{P}$
3:     Randomly select $\boldsymbol{s} \leftarrow\$ \mathbb{F}^{\mu}$ and send it to $\mathsf{P}$
4:     Receive claimed sum $S$ from $\mathsf{P}$
5:     SUMCHECK VERIFIER to check $\sum_{x \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) g(\boldsymbol{x}) = S$
6:     SUMCHECK VERIFIER through the outer sum for proving Equation 14
           $$\sum_{\boldsymbol{y} \in I_\rho} f(\boldsymbol{y}) \sum_{\boldsymbol{x} \in B_\mu | \rho(\boldsymbol{x}) = \boldsymbol{y}} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{y}) = S$$
       Let the challenges be $\boldsymbol{\alpha} \in \mathbb{F}^{\kappa}$, and the claim at the end be
           $$f(\boldsymbol{\alpha}) \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S'$$
7:     Query $[[f]]$ at $\boldsymbol{\alpha}$ to get evaluation $S_f$. Compute $S \leftarrow S'/S_f$
8:     Adapt $\mathsf{MulPerm}$ to check
           $$\sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S$$
9: end procedure
10:
11: procedure LOOKUP PROVER($f, g$)
12:     Compute $\rho$ such that $f(\rho(\boldsymbol{y})) = g(\boldsymbol{y})$ for every $\boldsymbol{y} \in B_\mu$.
13:     Interpolate the MLE $\tilde{\rho}_{[\kappa]}$ and send oracle $[[\tilde{\rho}_{[\kappa]}]]$ to $\mathsf{V}$
14:     Receive random challenges $\boldsymbol{s} \in \mathbb{F}^{\mu}$ from $\mathsf{V}$
15:     Compute $S \leftarrow \sum_{x \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) g(\boldsymbol{x})$. Send $S$ to $\mathsf{V}$
16:     SUMCHECK PROVER for proving $\sum_{x \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) g(\boldsymbol{x}) = S$
17:     SUMCHECK PROVER through the outer sum for proving Equation 14
           $$\sum_{\boldsymbol{y} \in I_\rho} f(\boldsymbol{y}) \sum_{\boldsymbol{x} \in B_\mu | \rho(\boldsymbol{x}) = \boldsymbol{y}} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{y}) = S$$
       Let the challenges be $\boldsymbol{\alpha} \in \mathbb{F}^{\kappa}$, and the claim at the end be
           $$f(\boldsymbol{\alpha}) \sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S'$$
18:     Adapt $\mathsf{MulPerm}$ to prove
           $$\sum_{\boldsymbol{x} \in B_\mu} \mathsf{eq}(\boldsymbol{x}, \boldsymbol{s}) \tilde{\mathbb{1}}_\rho(\boldsymbol{x}, \boldsymbol{\alpha}) = S'/f(\boldsymbol{\alpha})$$
19: end procedure
```

Let $n := 2^\mu, T = 2^\kappa$ and $\mu' = \log \kappa + \mu$. To prove $\big([[f]], [[g]]; f, g\big)$ for $f \in \mathcal{F}_\kappa^{(\leq 1)}$, $g \in \mathcal{F}_\mu^{(\leq 1)}$ using MulPerm, the prover will need to first compute $\tilde{\rho}_{[\kappa]}$ and send an oracle of it to the verifier. Moreover, since $\mathcal{R}_{\mathsf{LKUP2}}$ is unindexed, we need to prove $\tilde{\rho}_{[\kappa]}$ maps to $\{0, 1\}$ over $B_{\mu'}$, which can be reduced to another sumcheck as described in Section 8. Following that analysis, if $T \leq n$, then the prover cost is $n \cdot o(\log \log n)$ field operations, which does not dominate in the total cost of lookup PIOP. If $T > n$, the prover cost is $o(n\kappa/\mu) = o(n \log n)$ field operations. In either case, the cost for other parts of the lookup argument dominates.

10 Summary and Future Directions

To summarize, both of our permutation checks achieve logarithmic verifier complexity, soundness error that only grows polylogarithmically with the number of permuted elements, and their prover only needs to commit to the witness without invoking GKR. Beyond committing to the witness, BiPerm only incurs linear prover cost but requires using sparse PCS due to the large size of $\tilde{\mathbb{1}}_L, \tilde{\mathbb{1}}_R$; MulPerm incurs a slightly larger but close to linear prover cost and can be compiled using any PCS.

Our arguments can be generalized to lookups and support prover-provided mappings, and result in similar prover cost. Particularly, MulLookup, which is the generalization of MulPerm to lookups, has prover cost $n \cdot \tilde{O}(\sqrt{\log T})$, where T is the size of the lookup table and n is the size of the witness.

One open problem is to construct fully linear permutation argument and lookup argument with no PCS dependency that preserve the prover cost, verifier cost, and soundness error. Another open problem is to further reduce the soundness error of permutation and lookup arguments to constant over the size of the field.

Acknowledgments. We would like to thank Justin Thaler for feedback on early stages of this work. We thank Zach Langley and Jonathan Wang at Axiom for discussion regarding prover-provided permutation. This work is supported by the Google CyberNYC grant, the NYU Center for Cybersecurity, and generous gifts from Chaincode Labs, Aspen Labs, and Ingonyama.

References

[ACFY24] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. "STIR: Reed-Solomon Proximity Testing with Fewer Queries". In: *CRYPTO 2024, Part X*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14929. LNCS. Springer, Cham, Aug. 2024, pp. 380–413. DOI: `10.1007/978-3-031-68403-6_12`.

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. "WHIR: Reed-Solomon Proximity Testing with Super-Fast Verification". In: *EUROCRYPT 2025, Part IV*. Ed. by Serge Fehr and Pierre-Alain Fouque. Vol. 15604. LNCS. Springer, Cham, May 2025, pp. 214–243. DOI: `10.1007/978-3-031-91134-7_8`.

[AFK21] Thomas Attema, Serge Fehr, and Michael Klooß. *Fiat-Shamir Transformation of Multi-Round Interactive Proofs*. Cryptology ePrint Archive, Report 2021/1377. 2021. URL: `https://eprint.iacr.org/2021/1377`.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup". In: *ACM CCS 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM Press, 2017, pp. 2087–2104. DOI: `10.1145/3133956.3134104`.

[AST24] Arasu Arun, Srinath T. V. Setty, and Justin Thaler. "Jolt: SNARKs for Virtual Machines via Lookups". In: *EUROCRYPT 2024, Part VI*. Ed. by Marc Joye and Gregor Leander. Vol. 14656. LNCS. Springer, Cham, May 2024, pp. 3–33. DOI: `10.1007/978-3-031-58751-1_1`.

[BBBPWM18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018, pp. 315–334. DOI: `10.1109/SP.2018.00020`.

[BBHR18a] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity". In: *ICALP 2018*. Ed. by Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella. Vol. 107. LIPIcs. Schloss Dagstuhl, July 2018, 14:1–14:17. DOI: `10.4230/LIPIcs.ICALP.2018.14`.

[BBHR18b] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Report 2018/046. 2018. URL: `https://eprint.iacr.org/2018/046`.

[BCGJM18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. "Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution". In: *ASIACRYPT 2018,* *Part I*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11272. LNCS. Springer, Cham, Dec. 2018, pp. 595–626. DOI: 10.1007/978-3-030-03326-2_20.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. "Transparent SNARKs from DARK Compilers". In: *EUROCRYPT 2020, Part I*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12105. LNCS. Springer, Cham, May 2020, pp. 677–706. DOI: 10.1007/978-3-030-45721-1_24.

[BG12] Stephanie Bayer and Jens Groth. "Efficient Zero-Knowledge Argument for Correctness of a Shuffle". In: *EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. LNCS. Springer, Berlin, Heidelberg, Apr. 2012, pp. 263–280. DOI: 10.1007/978-3-642-29011-4_17.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: *EUROCRYPT 2023, Part II*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14005. LNCS. Springer, Cham, Apr. 2023, pp. 499–530. DOI: 10.1007/978-3-031-30617-4_17.

[CHMMVW20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: *EUROCRYPT 2020, Part I*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12105. LNCS. Springer, Cham, May 2020, pp. 738–768. DOI: 10.1007/978-3-030-45721-1_26.

[DP25] Benjamin E. Diamond and Jim Posen. "Succinct Arguments over Towers of Binary Fields". In: *EUROCRYPT 2025, Part IV*. Ed. by Serge Fehr and Pierre-Alain Fouque. Vol. 15604. LNCS. Springer, Cham, May 2025, pp. 93–122. DOI: 10.1007/978-3-031-91134-7_4.

[EFG22] Liam Eagen, Dario Fiore, and Ariel Gabizon. *cq: Cached quotients for fast lookups*. Cryptology ePrint Archive, Report 2022/1763. 2022. URL: https://eprint.iacr.org/2022/1763.

[GK22] Ariel Gabizon and Dmitry Khovratovich. *flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size*. Cryptology ePrint Archive, Report 2022/1447. 2022. URL: https://eprint.iacr.org/2022/1447.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. "Delegating computation: interactive proofs for muggles". In: *40th ACM STOC*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 113–122. DOI: 10.1145/1374376.1374396.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[Hab22] Ulrich Haböck. *Multivariate lookups based on logarithmic derivatives.* Cryptology ePrint Archive, Report 2022/1530. 2022. URL: https://eprint.iacr.org/2022/1530.

[HLQXYZZ25] Yuncong Hu, Chongrong Li, Zhi Qiu, Tiancheng Xie, Yue Ying, Jiaheng Zhang, and Zhenfei Zhang. *GKR for Boolean Circuits with Sub-linear RAM Operations.* Cryptology ePrint Archive, Paper 2025/717. 2025. URL: https://eprint.iacr.org/2025/717.

[KRS25] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. "How to Prove False Statements: Practical Attacks on Fiat-Shamir". In: *CRYPTO 2025, Part VI.* Ed. by Yael Tauman Kalai and Seny F. Kamara. Vol. 16005. LNCS. Springer, Cham, Aug. 2025, pp. 3–26. DOI: 10.1007/978-3-032-01887-8_1.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *ASIACRYPT 2010.* Ed. by Masayuki Abe. Vol. 6477. LNCS. Springer, Berlin, Heidelberg, Dec. 2010, pp. 177–194. DOI: 10.1007/978-3-642-17373-8_11.

[KZHB25] George Kadianakis, Arantxa Zapico, Hossein Hafezi, and Benedikt Bünz. *KZH-Fold: Accountable Voting from Sublinear Accumulation.* Cryptology ePrint Archive, Report 2025/144. 2025. URL: https://eprint.iacr.org/2025/144.

[Lee21] Jonathan Lee. "Dory: Efficient, Transparent Arguments for Generalised Inner Products and Polynomial Commitments". In: *TCC 2021, Part II.* Ed. by Kobbi Nissim and Brent Waters. Vol. 13043. LNCS. Springer, Cham, Nov. 2021, pp. 1–34. DOI: 10.1007/978-3-030-90453-1_1.

[LFKN90] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. "Algebraic Methods for Interactive Proof Systems". In: *31st FOCS.* IEEE Computer Society Press, Oct. 1990, pp. 2–10. DOI: 10.1109/FSCS.1990.89518.

[Lip89] Richard J Lipton. *Fingerprinting sets.* Princeton University, Department of Computer Science, 1989.

[LZ19] Qipeng Liu and Mark Zhandry. "Revisiting Post-quantum Fiat-Shamir". In: *CRYPTO 2019, Part II.* Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11693. LNCS. Springer, Cham, Aug. 2019, pp. 326–355. DOI: 10.1007/978-3-030-26951-7_12.

[LZ25] Tianyi Liu and Yupeng Zhang. *Efficient SNARKs for Boolean Circuits via Sumcheck over Tower Fields.* Cryptology ePrint Archive, Report 2025/594. 2025. URL: https://eprint.iacr.org/2025/594.

[Nef01] C. Andrew Neff. “A Verifiable Secret Shuffle and Its Application to e-Voting”. In: *ACM CCS 2001*. Ed. by Michael K. Reiter and Pierangela Samarati. ACM Press, Nov. 2001, pp. 116–125. DOI: 10.1145/501983.502000.

[PH23] Shahar Papini and Ulrich Haböck. *Improving logarithmic derivative lookups using GKR*. Cryptology ePrint Archive, Report 2023/1284. 2023. URL: https://eprint.iacr.org/2023/1284.

[PST13] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. “Signatures of Correct Computation”. In: *TCC 2013*. Ed. by Amit Sahai. Vol. 7785. LNCS. Springer, Berlin, Heidelberg, Mar. 2013, pp. 222–242. DOI: 10.1007/978-3-642-36594-2_13.

[Set20] Srinath Setty. “Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup”. In: *CRYPTO 2020, Part III*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12172. LNCS. Springer, Cham, Aug. 2020, pp. 704–737. DOI: 10.1007/978-3-030-56877-1_25.

[SL20] Srinath Setty and Jonathan Lee. *Quarks: Quadruple-efficient transparent zkSNARKs*. Cryptology ePrint Archive, Report 2020/1275. 2020. URL: https://eprint.iacr.org/2020/1275.

[Sou23] Lev Soukhanov. *Power circuits: a new arithmetization for GKR-styled sumcheck*. Cryptology ePrint Archive, Report 2023/1611. 2023. URL: https://eprint.iacr.org/2023/1611.

[Sou25] Lev Soukhanov. *Logup*: faster, cheaper logup argument for small-table indexed lookups*. Cryptology ePrint Archive, Paper 2025/946. 2025. URL: https://eprint.iacr.org/2025/946.

[ST25] Srinath Setty and Justin Thaler. *Twist and Shout: Faster memory checking arguments via one-hot addressing and increments*. Cryptology ePrint Archive, Report 2025/105. 2025. URL: https://eprint.iacr.org/2025/105.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. *Customizable constraint systems for succinct arguments*. Cryptology ePrint Archive, Report 2023/552. 2023. URL: https://eprint.iacr.org/2023/552.

[STW24] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. “Unlocking the Lookup Singularity with Lasso”. In: *EUROCRYPT 2024, Part VI*. Ed. by Marc Joye and Gregor Leander. Vol. 14656. LNCS. Springer, Cham, May 2024, pp. 180–209. DOI: 10.1007/978-3-031-58751-1_7.

[Tha20] Justin Thaler. *Proofs, arguments, and zero-knowledge*. 2020.

[Wik21] Douglas Wikström. *Special Soundness in the Random Oracle Model*. Cryptology ePrint Archive, Report 2021/1265. 2021. URL: https://eprint.iacr.org/2021/1265.

[WTsTW18] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. “Doubly-Efficient zkSNARKs Without Trusted Setup”. In: *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018, pp. 926–943. DOI: 10.1109/SP.2018.00060.

[ZBKMNS22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. “Caulk: Lookup Arguments in Sublinear Time”. In: *ACM CCS 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM Press, Nov. 2022, pp. 3121–3134. DOI: 10.1145/3548606.3560646.

[ZGKMR22] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. *Baloo: Nearly Optimal Lookup Arguments*. Cryptology ePrint Archive, Report 2022/1565. 2022. URL: https://eprint.iacr.org/2022/1565.

A Proof of Lemma 4

Lemma 4. *Let $n = 2^\mu$. Given $f, g \in \mathcal{F}_\mu^{(\leq 1)}$, and preprocessed $\tilde{\sigma} \in \mathsf{PERM}(B_\mu)$, sumcheck PIOP for*

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{\alpha}) = g(\boldsymbol{\alpha}) \tag{3}$$

proves $\big(\tilde{\sigma}, [[\tilde{\sigma}]]; ([[f]], [[g]]); (f, g)\big) \in \mathcal{R}_{\mathsf{PERM}}$ with perfectly completeness.

Proof. $\big(\tilde{\sigma}, [[\tilde{\sigma}]]; ([[f]], [[g]]); (f, g)\big) \in \mathcal{R}_{\mathsf{PERM}}$ if and only if

$$f(\boldsymbol{x}) = g(\sigma(\boldsymbol{x})) \quad \forall \boldsymbol{x} \in B_\mu.$$

Because σ is an automorphism of B_μ, this is equivalent to

$$f(\sigma^{-1}(\boldsymbol{y})) = g(\boldsymbol{y}) \quad \forall \boldsymbol{y} \in B_\mu.$$

The composition on the left hand side can be decomposed as

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \mathbb{1}_{\sigma^{-1}}(\boldsymbol{y}, \boldsymbol{x}) = g(\boldsymbol{y}) \quad \forall \boldsymbol{y} \in B_\mu.$$

And the identity $\mathbb{1}_{\sigma^{-1}}(\boldsymbol{y}, \boldsymbol{x}) = \mathbb{1}_\sigma(\boldsymbol{x}, \boldsymbol{y})$ allows for writing this equivalently as

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \mathbb{1}_\sigma(\boldsymbol{x}, \boldsymbol{y}) = g(\boldsymbol{y}) \quad \forall \boldsymbol{y} \in B_\mu.$$

The right-hand side is multilinear in $\boldsymbol{y}$. Substituting $\mathbb{1}_\sigma$ with $\tilde{\mathbb{1}}_\sigma$, an arbitrary arithmetization of $\mathbb{1}_\sigma$ that is multilinear in $\boldsymbol{y}$, results in the following multilinear polynomial equality condition

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{y}) = g(\boldsymbol{y}) \quad \forall \boldsymbol{y} \in B_\mu. \tag{17}$$

A multilinear polynomial in μ variables is uniquely determined by its evaluation over B_μ, so this check can be written as the formal equality of the polynomials on the left and right-hand sides

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{Y}) = g(\boldsymbol{Y}).$$

Polynomial equality can be tested probabilistically by checking that both sides are equal when evaluated at a random $\boldsymbol{\alpha} \in \mathbb{F}^\mu$

$$\sum_{\boldsymbol{x} \in B_\mu} f(\boldsymbol{x}) \cdot \tilde{\mathbb{1}}_\sigma(\boldsymbol{x}, \boldsymbol{\alpha}) = g(\boldsymbol{\alpha}) \tag{3}$$

Finally, for efficiency the evaluation of the left-hand side can be outsourced to an untrusted prover via the sum-check protocol.

Perfect Completeness follows from the fact that there is a bidirectional implication between each step described above.

B Computing the collapsed evaluation table before switching algorithms in the second sumcheck

Before switching from the bucketing algorithm back to direct computation in round $\log \ell$, the prover needs to compute the evaluation tables of $\tilde{\sigma}\big(\langle i\rangle, (\boldsymbol{\gamma}, \boldsymbol{x})\big)$ for every $i \in [\mu]$ in order to continue the protocol. Algorithm 11 achieves this using fewer than $\ell \cdot 2^{\ell}$ field operations.

Algorithm 11 Algorithm for Efficiently Computing the Evaluation Table of $\tilde{\sigma}_{[\mu]}$ before switching algorithms in the Second Sumcheck in MulPerm

```
1: procedure COLLAPSE($\tilde{\sigma}_{[\mu]}, \boldsymbol{\gamma} \in \mathbb{F}^k, k$)
2:     Let $S$ be the set of all $2^k$ bit strings
3:     Initialize empty table $\boldsymbol{T}$ of length $2^{2^k}$. Note that $S$ naturally corresponds to the
       set of row indices.
4:     for $\boldsymbol{s} \in S$ do
5:         Field element $v \leftarrow$ FOLD($\boldsymbol{s}, \boldsymbol{\gamma}, k$)
6:         Fill cell $\boldsymbol{s}$ in $\boldsymbol{T}$ with $v$
7:     end for
8:     Initialize an empty evaluation table of $\tilde{\sigma}_{[\mu]}\big(\langle i\rangle, (\boldsymbol{\gamma}, \boldsymbol{x})\big)$ for every $i \in [\mu]$. Each
       of these $\mu$ tables have length $n/2^k$.
9:     for $\boldsymbol{i} \in [\mu], j \in [n/2^k]$ do
10:        $\boldsymbol{s} \leftarrow$ the $(j-1)n/2^k+1$ to $jn/2^k$ entries in the evaluation table of $\tilde{\sigma}_{[\mu]}\big(\langle i\rangle, \boldsymbol{x}\big)$
11:        $v \leftarrow$ value in cell $\boldsymbol{s}$ in $\boldsymbol{T}$
12:        Fill the $j$th cell in the evaluation table of $\tilde{\sigma}_{[\mu]}\big(\langle i\rangle, (\boldsymbol{\gamma}, \boldsymbol{x})\big)$ with $v$
13:    end for
14:    return evaluation tables of $\tilde{\sigma}_{[\mu]}\big(\langle i\rangle, (\boldsymbol{\gamma}, \boldsymbol{x})\big)$ for every $i \in [\mu]$
15: end procedure
16: procedure FOLD($\boldsymbol{s}, \boldsymbol{\gamma}, k$)
17:    for $i \in [k]$ do
18:        Initialize empty string $\boldsymbol{s}'$
19:        for $j \in [|\boldsymbol{s}|/2]$ do
20:            Append $\boldsymbol{s}_j + (\boldsymbol{s}_{|\boldsymbol{s}|/2+j} - \boldsymbol{s}_j) \cdot \boldsymbol{\gamma}_i$ to $\boldsymbol{s}'$
21:        end for
22:        $\boldsymbol{s} \leftarrow \boldsymbol{s}'$
23:    end for
24:    return $\boldsymbol{s}[1]$
25: end procedure
```