

Almost Linear-Time Permutation Check

Benedikt Bünz¹, Jessica Chen¹, Zachary DeStefano¹, and Binyi Chen² *

¹ New York University,

² Tsinghua 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. Moreover, both have a soundness error that grows linearly with the input size.

We reduce permutation argument to sumcheck argument and present two permutation arguments that have $\text{polylog}(n)/|\mathbb{F}|$ soundness error. 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 general construction, MulPerm, which has no restriction on the PCS choice and its prover performs essentially linear field operations, $n \cdot \tilde{O}(\sqrt{\log(n)})$. Both permutation arguments generalize to lookups. We demonstrate how our arguments can be used to improve SNARK systems such as HyperPlonk (Eurocrypt 2023) and Spartan (Crypto 2020), and build a GKR-based protocol for proving non-uniform circuits. Further, MulPerm enables the fastest succinct argument with sublinear verification for *arbitrary* circuits over *small* fields. Plugging MulPerm into prior small field constructions overcomes the previous lack of an efficient permutation argument with sufficiently small soundness error.

Keywords: Proof system · Permutation check · Sumcheck

* bb, jessicachen, zd@nyu.edu, by-chen@mail.tsinghua.edu.cn

Table of Contents

1	Introduction	4
1.1	Our contribution	5
1.2	Applications	8
1.3	BiPerm and Shout	10
1.4	R1CS-style GKR	11
1.5	Additional related work	11
2	Technical overview	12
2.1	BiPerm, a fully linear permutation check for some PCS	14
2.2	MulPerm, an almost-linear permutation check for any PCS	14
2.3	Prover-provided permutation and lookup checks	16
3	Applications to SNARKs	17
3.1	Single-commit HyperPlonk	17
3.2	Spartan with logarithmic verifier time	17
3.3	R1CS-style GKR	18
4	Preliminaries	20
4.1	Interactive proofs and arguments of knowledge	21
4.2	Polynomial Interactive Oracle Reduction and Polynomial Interactive Oracle Proof	22
4.3	PIOP compilation	24
4.4	Sumcheck protocol	24
4.5	Multilinear polynomial commitments	25
4.6	Permutation and its inverse	27
5	Reduce permcheck to sumcheck	28
5.1	Degree- μ arithmetization of $\tilde{\mathbb{I}}_\sigma$	30
5.2	Lower-degree arithmetization of $\tilde{\mathbb{I}}_\sigma$ to improve prover complexity	32
6	BiPerm: Linear-time permutation check for sparse PCS	34
6.1	Compiling BiPerm using PCS	37
7	MulPerm: Almost linear-time permutation check for any PCS	37
7.1	Evaluating $\tilde{\mathbf{p}}$ over the boolean hypercube	40
7.2	First sumcheck	41
7.3	Second sumcheck	43
7.4	Setting the group parameter ℓ	47
7.5	Compiling MulPerm using PCS	48
8	Prover-provided permutation	48
8.1	Proving σ is a permutation	49
8.2	Improving the prover cost when proving σ maps to binaries	50
8.3	Extending BiPerm to prover-provided permutation	52
9	Generalization to Lookups	54
9.1	Sumcheck PIOP for lookups	55
9.2	Prover-provided lookup reduced to sumcheck	57
10	Summary and Future Directions	59

A	Computing the collapsed evaluation table before switching algorithms in the second sumcheck	66
---	--	----

1 Introduction

Proof systems enable a prover to convince a verifier that they correctly executed some computation. In modern proof systems, the computation is formulated as an arithmetic circuit over some finite field $\mathbb{F}$, and the proving protocol is constructed from an underlying information-theoretic protocol called a Polynomial Interactive Oracle Proof (PIOP). The PIOP can then be compiled into a Succinct Non-interactive ARGument of Knowledge (SNARK) using a suitable Polynomial Commitment Scheme (PCS) and a hash-function modeled as a random oracle.

In a PIOP, the verifier sends random challenges, and the prover sends polynomial (univariate or multilinear) oracles; the verifier 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. The key objective when designing PIOPs is to reduce the following: prover time, the number and length of prover-sent oracles, query complexity to the oracles, verifier time, and soundness error. An optimal PIOP would only require sending one oracle that depends on the witness, have linear prover time, logarithmic (or constant) verifier time and proof size, and soundness error as small as possible.

To investigate how the PIOPs underlying SNARKs for arithmetic circuits fare, we need to study two key components: gate check and wire check. The gate check proves that every gate (addition or multiplication) computes its output correctly, e.g. $\text{input}_{\text{left}} \cdot \text{input}_{\text{right}} = \text{output}$. The state-of-the-art approach to gate check is already highly efficient. The gate constraints for all gates can be checked using a single invocation of the sumcheck protocol, which requires a single query to the witness oracle, linear prover time, logarithmic proof size, and verifier time, and logarithmic soundness error [Set20; CBBZ23].

The wire check is more complicated to perform efficiently. It proves the correctness of wiring; that is, the input of one gate corresponds to the relevant output of another gate. The ways of handling wire check differ among previous works [GWC19; CBBZ23; Set20; BBHR18b; SL20] but all rely on a common primitive – permutation check (permcheck). In a permutation check, the prover proves that $f(x) = g(\sigma(x)) \forall x \in H$ for some functions f and g ³ that map from some set H to the wire values in some finite field $\mathbb{F}$; σ is a permutation on H , such that $\sigma(i) = j$ for $i, j \in H$ if and only if the i th and j th wire are identical.

Beyond forming the core of SNARK protocols, permutation checks also have applications to memory-checking, 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 suboptimal PIOPs.

The first approach is based on product check (ProdCheck). It 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 set on H [Lip89; Nef01; BG12]⁴. The prover proves the correct evaluation of these products at a random point $r \in \mathbb{F}$. There are some downsides to this ap-

³ In some applications $f = g$

⁴ This check proves that there exists a permutation between f and g but can be easily turned into a check that a *specific* permutation of $f(H)$ yields $g(H)$.

proach. 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 span the entire field even if f and g only interpolate small elements. When the permutation check PIOP is compiled, this oracle corresponds to an additional commitment, and 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 . Moreover, the PIOP has soundness error $n/|\mathbb{F}|$, since it checks equality between two degree- n polynomials.

The second approach is based on GKR. It views the computation of the above 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 cost 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. Prior work [SL20] has proposed combining the protocols and first running $\log \lambda$ rounds of GKR and then using the commitment method. While this improves the verifier time and proof size to $\log \lambda \cdot \log n$, it does not resolve the soundness issue.

1.1 Our contribution

We propose two new permutation check PIOPs, BiPerm and MulPerm, that only run a few (one or two) rounds of sumcheck protocols and do not require committing to additional polynomials during the runtime of the protocol. In Table 1, we give an overview of our permutation PIOPs and compare them to prior work. In Table 2, we evaluate the prover cost and verifier time when the PIOPs are instantiated with different PCS schemes, applied to a preprocessed permutation.

PIOP	$ \text{Comm.} (\mathbb{F})$	Rounds	$ \mathbf{i} $	$wt(\mathbf{i})$	PCS	$\delta_s(./ \mathbb{F})$
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
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: $|\text{Comm.}|(\mathbb{F})$ is the number of field elements we commit to beyond the witness, $|\mathbf{i}|$ is the length of oracles preprocessed in the index; $wt(\mathbf{i})$ is the weight of index, i.e. the nonzero number of bits in the index; $|\mathbf{P} \text{ msgs}|$ 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 [SL20]; and Shout refers to the protocol in [ST25]. Sparse PCS refers to a PCS that allows committing to and opening an n -sparse polynomial in $O(n)$ time.

PIOP	KZG (MSM)		Dory (MSM)		FRI (#H)		Ligero (#H)		# $\mathbb{F}$ ops	V time
	Pre	Prove	Pre	Prove	Pre	Prove	Pre	Prove		
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	$k \log n$
BiPerm/Shout	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]. B means $\{0, 1\}$, and $\mathbb{W} \subseteq \mathbb{F}$ is a subfield containing small elements. See the caption of Table 1 for explanation of the other notations. Only the dominating factor is listed. The costs that translate into significant overheads are red.

Interpreting the comparisons. The comparison between the different permutation checks is multi-dimensional. The ProdCheck-based permutation check protocols, while widely deployed, have the worst cryptographic commitment cost (proportional to n full field elements) and would therefore likely have the largest overall prover time. Note that the costs listed in Table 1 for ProdCheck-based permutation checks apply to LogUp-based ones as well, since they are essentially different views of the same protocol. LogUp is based on the logarithmic derivative of the grand product, a format which lends itself somewhat more naturally to a sumcheck formulation with the same asymptotic costs and similar constants. Both of these protocols require committing to n full field elements.

Compared to the GKR-based protocols [Set20; PH23], our protocols have better soundness, proof size, and verifier time. BiPerm and Shout [ST25] are essentially optimal protocols; however, they require preprocessing and opening a n -sparse oracle of size $O(n^{1.5})$, which implies they can only be reasonably instantiated with certain group-based PCS schemes that have sublinear opening costs, such as Hyrax [WTsTW18], Dory [Lee21], or KZH [KZHB25]. MulPerm has slightly larger prover field operations than GKR-based protocols, with the benefit of improved soundness and lower verifier cost and proof size. We believe that in many practical deployments, these field operations are dwarfed by the other cryptographic commitment operations, which are identical in both MulPerm and the GKR-based protocols.

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.

Fastest small field prover for arbitrary circuits MulPerm also enables the fastest (interactive) argument, with succinct verification, for arbitrary circuits over fields of size $\text{polylog}(n)$. This result is achieved by taking a linear-time PIOP (e.g. Spartan [Set20], or HyperPlonk [CBBZ23]) with MulPerm as the permutation PIOP (see Section 3) and compiling it to an interactive argument using a linear-time polynomial commitment (e.g [BCG20; GLSTW23]). The resulting argument has at most constant soundness-error over any field $\mathbb{F}$, s.t. $|\mathbb{F}| = \Omega(\log^{1.5} n)$.

This result follows a line of work on constructing arguments for small fields with the goal of reducing proving overhead. The best of these works achieve a linear-time prover [BCG20; RR21; BCGL22; HR22], but none of them apply to arbitrary circuits. This is because, due to a prior lack of efficient permutation argument with sufficiently small soundness error, these constructions have to

circumvent permutation check by working in settings where the circuit computation is small, structured or can be preprocessed. Thus, they either work with highly-structured computations such as automata, require verifier time linear in the size of the circuit, or require a private preprocessing step. An important open question is how to achieve *entirely* linear-time arguments for small fields and arbitrary circuits.

Practical implications of soundness on prover time We remark that MulPerm is the only protocol that has small soundness error ($\text{polylog}(n)/|\mathbb{F}|$) and can work with any PCS. This is relevant from a practical perspective, even when operating over an exponentially-sized field. Practitioners want to work over fields such as $\text{GF}(2^{128})$ [DP25] or other extension fields that are just below 2^{128} in size. These field elements can be represented using two 64-bit elements on modern CPUs and are, therefore, highly efficient [DP25]. However, achieving even 100-bit security⁵ is difficult when the soundness is proportional to $n/|\mathbb{F}|$, as $n > 2^{20}$ for most interesting applications. To resolve this, practitioners have to move to larger fields, use repetition, or rely on challenge grinding [Sta21], all of which increase the prover time.

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

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

1.2 Applications

We outline a few 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

⁵ Ethereum targets 128-bit security [Kad25].

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.

Further, Lasso [STW24] at its core relies on the Spark compiler. Replacing SPARK with our permutation argument also reduces the verification time, proof size and soundness error of Lasso.

Lookup arguments Lookup arguments check that the witness value is contained in a table of values. They are useful for range proofs and non-arithmetic operations [BCGJM18]. In range proofs, the witness value is looked up in a table consisting of all the values in the range; in non-arithmetic operations, the operation input and output is looked up in a table that contains all possible pairs of inputs and outputs [AST24].

We generalize MulPerm to 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], MulLookup commit to less additional information, and the prover’s only dependence on the table size T is the time to evaluate a multilinear evaluation of the table. MulLookup preserves the efficiency and soundness properties of MulPerm. The prover only needs to commit to the mapping from the witness to the table, and no additional commitment operation after that is needed. The prover cost is $n \cdot \tilde{O}(\sqrt{\log(n)})$ field operations for table of size $T < n$, and $O(n(\log T - \log n))$ for $T > n$. At the end of the protocol, the prover needs to open the witness and the table polynomials at a single point. While this requires $\Theta(T)$ field operations generically, it could be much cheaper in practice since the table polynomial can be efficiently evaluated for many tables of interest. For ex-

ample, in range proofs, the multilinear polynomial describing the table is simply $t(\mathbf{X}) = \sum_{i=0}^{\log T - 1} \mathbf{X}_i 2^i$, which can be evaluated in time $O(\log T)$.

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, if these large tables are structured, one can use the techniques from Lasso [STW24] in combination with our arguments.

PIOP	P time	V time	wt(Comm.)	Table Restriction
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 the lookup PIOPs in Lasso [STW24], LogUp [Hab22], HyperPlonk [CBBZ23], and Plookup [GW20], compared to MulLookup. 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 its 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.

Linear-time permutation Shortly subsequent to our work, [CTHK25] claims to achieve strictly linear-time prover cost with even lower cost than BiPerm. Their protocol is essentially identical to ours, except instantiated with a degree-1 arithmetization of $\mathbb{1}_\sigma$ (i.e. a direct commitment to the permutation matrix). However, as noted in 4.6, such formulation leads to a polynomial with 2μ variables. At the end of the sumcheck, this polynomial needs to be opened. The best PCS opening algorithms today still requires an MSM of size $\sqrt{2^{2\mu}} = n$ with λ -bit elements, which means the prover cost is at least $n\lambda$ group operations. In contrast, BiPerm and MulPerm do not have such a dependence on $n\lambda$ in either group or field operations.

2 Technical overview

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

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

Here, $\mathbb{1}_\sigma$ is the indicator function of the permutation σ .

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

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 time⁶.

The simplest idea is arithmetizing $\mathbb{1}_\sigma$ as a product of μ multilinear extensions of $\mathbb{1}_{\sigma_i}, \forall i \in [\mu]$, such that

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

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

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

Assuming that f and g are multilinear, this can be proven using a sumcheck, in which every univariate prover message has degree $(\mu + 1)$; this approach has prover complexity $O(n \cdot \mu^2)$ field operations and 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.

⁶ The best polynomial commitments have opening algorithms that take time $\sqrt{|f|}$, where $|f|$ is the size of the committed polynomial.

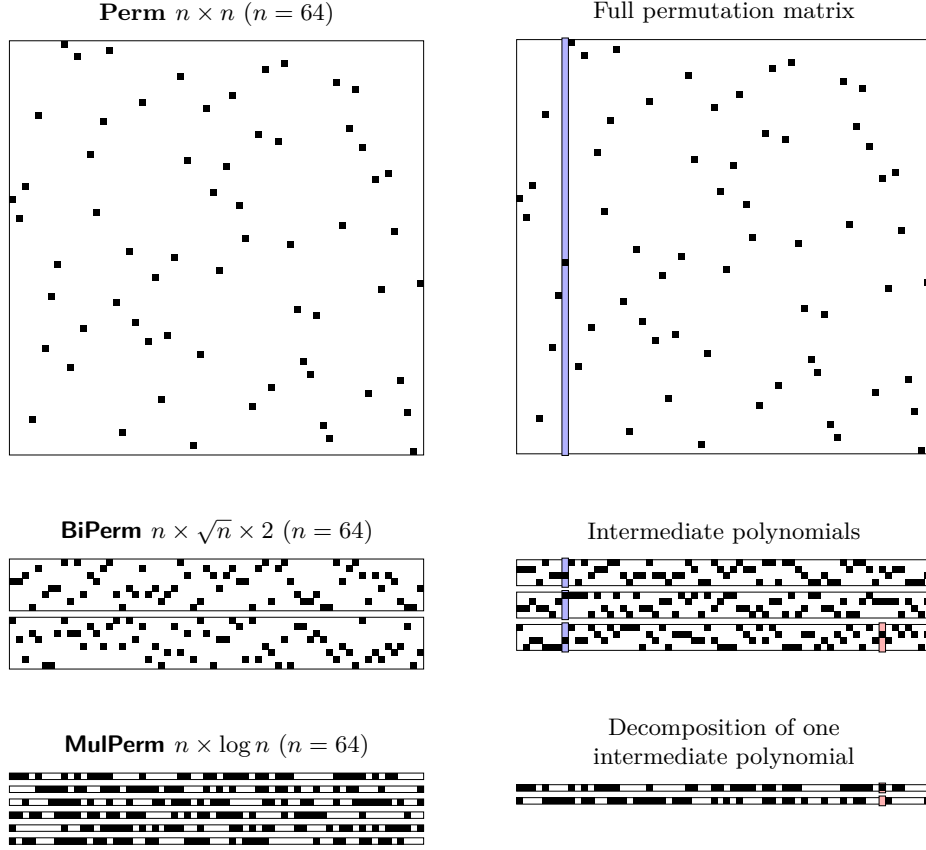


Fig. 1. High-level visual illustrations of the different polynomials used in the permutation arguments in this paper. **Left:** Illustration of the preprocessed eq polynomials in naïve Perm (Section 5), BiPerm (Section 6), and MulPerm (Section 7). Each rectangle is a 0/1 matrix representing evaluations of a committed polynomial where black cell represents the value 1 at that point (all other points are 0). Within each matrix, each row represents a $\mathbf{X} \in \{0, 1\}^\mu$ and each column represents a $\mathbf{Y} \in \{0, 1\}^\mu$ (potentially encoded across multiple polynomials). In Perm and BiPerm, each preprocessed matrix is n -sparse. In MulPerm, each preprocessed matrix is $n/2$ -sparse on average. For this $n = 64$ 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.

2.1 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:

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

This almost directly gives us our first protocol BiPerm. As a limitation BiPerm, uses $\tilde{\mathbb{1}}_{\sigma_L}(x, y_L)$ and $\tilde{\mathbb{1}}_{\sigma_R}(x, y_R)$, 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 Ligerio [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.

2.2 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, $\mathbf{p}_1, \dots, \mathbf{p}_\ell$. The prover can then use a sumcheck to prove that the degree- ℓ sumcheck statement

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

Proving this using a sumcheck argument reduces to evaluating $\mathbf{p}_1 \dots, \mathbf{p}_\ell$ 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 $\mathbf{p}_i$ using precommitted $\tilde{\mathbb{I}}_{[\sigma_i]} := \text{eq}(\sigma_i(x), y_i)$, which map from $\mathbb{F}^{\mu+1}$ to $\mathbb{F}$, i.e., are linear in size. Each $\mathbf{p}_i$ can, thus, be written as

$$\mathbf{p}_j(x, y^{(j)}) = \prod_{i=1}^{\mu/\ell} \text{eq}(\sigma_{j\mu/\ell+i}(x), y_i^{(j)})$$

where $y^{(i)}$ is the $[j \cdot \frac{\mu}{\ell} : (j+1) \cdot \frac{\mu}{\ell}]$ slice of vector y . Our key insight is that $\text{eq}(\sigma_i(x), y_i)$ maps to $\{0, 1\}$ for any $(x, y_i) \in \{0, 1\}^{\mu+1}$. In this second sumcheck, the prover needs to compute prover message $r_i(X_1) = \text{eq}(\sigma_i(X_1, x'), y_i)$ 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, $r(X_1) = \mathbf{p}_i(X_1, x', y_i)$ 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 k th 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.

Lookup arguments We also apply our techniques to lookup arguments. In lookup arguments, the permutation is replaced with some arbitrary map ρ^7 between g , defined over $\{0, 1\}^\mu$, and a table f , defined over $\{0, 1\}^\kappa$, such that

$$\sum_{\mathbf{y} \in \{0, 1\}^\kappa} f(\mathbf{y}) \mathbb{I}_\rho(\mathbf{x}, \mathbf{y}) = g(\mathbf{x}) \quad \forall \mathbf{x} \in \{0, 1\}^\mu. \quad (1)$$

Here $\mathbb{I}_\rho(\mathbf{x}, \mathbf{y})$ is 1 if $\rho(\mathbf{x}) = \mathbf{y}$, i.e. $g(\mathbf{x}) = f(\mathbf{y})$ for $\mathbf{x} \in \{0, 1\}^\mu$ and $\mathbf{y} \in \{0, 1\}^\kappa$. While this has a very similar structure to the permutation check, there are a number of challenges. Equation 1 is multilinear in neither $\mathbf{x}$ nor $\mathbf{y}$. This means we first need to run a sumcheck over $\mathbf{x}$ and $\mathbf{y}$ to show that

$$\sum_{\mathbf{x} \in \{0, 1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \sum_{\mathbf{y} \in \{0, 1\}^\kappa} f(\mathbf{y}) \mathbb{I}_\rho(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{x} \in \{0, 1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) g(\mathbf{x})$$

for a random vector $\mathbf{s} \in \mathbb{F}^\mu$.

Secondly, for large tables such that $\kappa > \mu$, a naïve implementation of the sumcheck takes 2^κ time. However, we note that $\mathbb{I}_\rho(y, x)$ is incredibly sparse and has at most n non-zero entries. We use this to reduce the prover time of the outer

⁷ Here, instead of σ , which we use to denote permutations, we use the symbol ρ to highlight the fact that the maps in lookup arguments are likely not bijective, which leads to non-trivial differences in the resulting sumchecks.

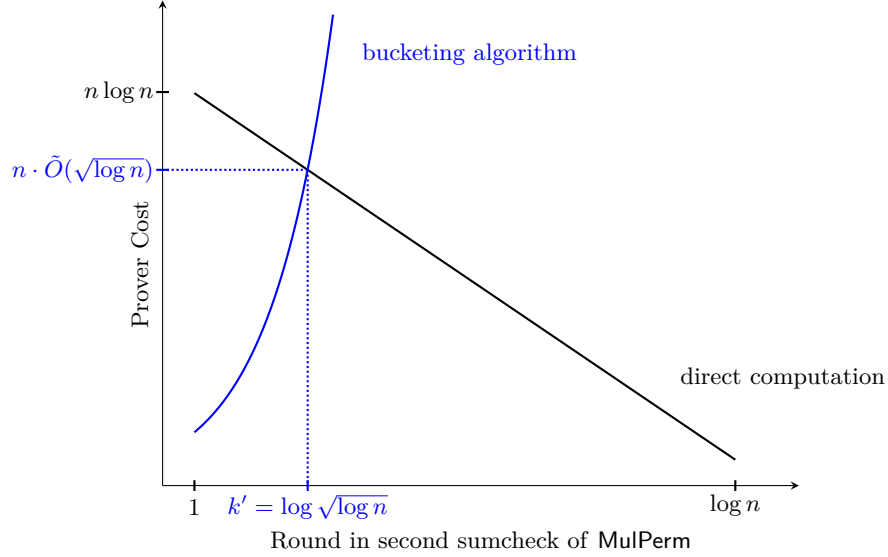


Fig. 2. For each round of the second sumcheck (Algorithm 7) in **MulPerm**: the black line shows the prover cost if directly computing the prover message, and the blue line shows the prover cost if using the bucketing algorithm (Algorithm 6). The idea is to use the bucketing trick up to round k' and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

sumcheck to $n \cdot (\kappa - \mu)$. Finally, $\mathbb{1}_\rho(y, x)$ can now be arithmetized as a product of κ eq polynomials. We reparameterize our protocol to achieve a prover time of $n \cdot \tilde{O}(\sqrt{\kappa})$ for the inner sumcheck. Our protocol also requires one evaluation of the 2^κ -sized lookup table f . This can be highly efficient for some tables, such as those for range proofs and all the RISC-V instructions described in Jolt [AST24].

2.3 Prover-provided permutation and lookup checks

Up till now, we assumed that the mapping is fixed and preprocessed. However, in many applications such as memory-checking and lookup checks, the prover actually computes and commits to the map during runtime. This presents a new challenge: what if the prover is dishonest and the provided map is not well-formed? To prevent this, the prover must additionally prove the structure of the provided map.

In a prover-provided permutation check, the prover needs to prove that the provided permutation σ only maps to binaries and that it is indeed a permutation, i.e. the domain and the codomain is the same set of elements. To do this, we have the prover commit to the inverse of σ as τ and prove that $\sigma(\tau(x)) = x \ \forall x \in \{0, 1\}^\mu$. This shows that σ has an inverse on $\{0, 1\}^\mu$, which implies it is bijective.

In a lookup check, it suffices to prove that the outputs of the map is binary. For a lookup that maps from $\{0, 1\}^\mu$ to $\{0, 1\}^\mu$, this can be done in $O(n \log \log n)$ time, which is less than the cost in the rest of the argument.

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 Single-commit 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 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	$2n$ $\mathbb{F}$ elements	n	$\log(n)$	Any
With MulPerm	$ \mathbb{w} $	$n \cdot \tilde{O}(\sqrt{\log(n)})$	$\log(n)$	Any
With BiPerm	$ \mathbb{w} $	n	$2\log(n)$	Sparse

Table 5. HyperPlonk [CBBZ23] with different permutation arguments. The Quarks-style (ProdCheck-based) permutation argument is described in HyperPlonk. $|\mathbb{w}|$ refers to the total weight of the witness and n to its length.

3.2 Spartan with logarithmic verifier time

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.⁸ 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 $val(), row(), col() : \{0, 1\}^\mu \mapsto \{0, 1\}^s$, such that when given input $\mathbf{j} \in \{0, 1\}^\mu$, outputs the value, row index, and column index of the j th nonzero element in the matrix, respectively. Then, for every $\mathbf{x}, \mathbf{y} \in \{0, 1\}^s$,

$$\tilde{M}(\mathbf{x}, \mathbf{y}) := \sum_{\mathbf{j} \in \{0, 1\}^\mu} \tilde{val}(\mathbf{j}) \cdot \tilde{\mathbb{I}}_{row}(\mathbf{j}, \mathbf{x}) \cdot \tilde{\mathbb{I}}_{col}(\mathbf{j}, \mathbf{y})$$

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

$$\tilde{M}(\mathbf{r}_x, \mathbf{r}_y) := \sum_{\mathbf{j} \in \{0, 1\}^\mu} \tilde{val}(\mathbf{j}) \cdot \tilde{\mathbb{I}}_{row}(\mathbf{j}, \mathbf{r}_x) \cdot \tilde{\mathbb{I}}_{col}(\mathbf{j}, \mathbf{r}_y)$$

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 arguments 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 $i \in [L - 1]$, $\mathbf{z}_i$ are the inputs to Layer i and $\mathbf{z}_{i+1}$ are the outputs of Layer i , such that $A^{(i)} \mathbf{z}_i \circ B^{(i)} \mathbf{z}_i = \mathbf{z}_{i+1}$ where matrices $A^{(i)}, B^{(i)} \in \mathbb{F}^{|\mathbf{z}_{i+1}| \times |\mathbf{z}_i|}$. Let $\mu_i = \log |\mathbf{z}_i|$.

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

$$(\mathbf{i}; \mathbf{o}; \mathbf{w}) = (\tilde{A}^{(i)}, \tilde{B}^{(i)}, [[\tilde{A}^{(i)}]], [[\tilde{B}^{(i)}]], [[\tilde{z}_i]]; \tilde{z}_i \mid i \in [L])$$

⁸ 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.

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

$$\left(\sum_{\mathbf{y} \in \{0, 1\}^{\mu_i}} \tilde{A}^{(i)}(\mathbf{x}, \mathbf{y}) \cdot \tilde{z}_i(\mathbf{y}) \right) \left(\sum_{\mathbf{y} \in \{0, 1\}^{\mu_i}} \tilde{B}^{(i)}(\mathbf{x}, \mathbf{y}) \cdot \tilde{z}_i(\mathbf{y}) \right) = \tilde{z}_{i+1}(\mathbf{x}) \quad \forall \mathbf{x} \in \{0, 1\}^{\mu_{i+1}}$$

The task of proving the equation above for every $x \in \{0, 1\}^{\mu_{i+1}}$ can be reduced to a single sumcheck using random challenge $\mathbf{r} \in \mathbb{F}^{\mu_i}$

$$\sum_{\mathbf{x} \in \{0, 1\}^{\mu_{i+1}}} \text{eq}(\mathbf{x}, \mathbf{r}) \cdot \left(\sum_{\mathbf{y} \in \{0, 1\}^{\mu_i}} \tilde{A}^{(i)}(\mathbf{x}, \mathbf{y}) \cdot \tilde{z}_i(\mathbf{y}) \right) \left(\sum_{\mathbf{y} \in \{0, 1\}^{\mu_i}} \tilde{B}^{(i)}(\mathbf{x}, \mathbf{y}) \cdot \tilde{z}_i(\mathbf{y}) \right) = \tilde{z}_{i+1}(\mathbf{r}) \quad (2)$$

Again, we adopt the way of encoding sparse polynomial $\tilde{M} \in \{\tilde{A}^{(i)}, \tilde{B}^{(i)} \mid i \in [L]\}$ 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 $\{0, 1\}^s$ to their corresponding boolean hypercube; when given input $\mathbf{j} \in \{0, 1\}^s$, $val(), row(), col()$ output the value, row index, and column index of the j th nonzero element in the matrix, respectively. Then, for every $\mathbf{x} \in \{0, 1\}^{\mu_{i+1}}, \mathbf{y} \in \{0, 1\}^{\mu_i}$,

$$\tilde{M}(\mathbf{x}, \mathbf{y}) := \sum_{\mathbf{j} \in \{0, 1\}^s} \tilde{val}(\mathbf{j}) \cdot \tilde{\mathbb{I}}_{row}(\mathbf{j}, \mathbf{x}) \cdot \tilde{\mathbb{I}}_{col}(\mathbf{j}, \mathbf{y})$$

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 $(\tilde{A}^{(i)}, \tilde{B}^{(i)}, [[\tilde{A}^{(i)}]], [[\tilde{B}^{(i)}]]; [[\tilde{z}_i]]; \tilde{z}_i \mid i \in [L]) \in \mathcal{R}_{\text{RICSGKR}}$. Run sumcheck for Equation 2 for every layer. After each sumcheck of Layer $i \in [L - 1]$, verifier is left with prover claimed evaluations $A^{(i)}(\mathbf{r}_x, \mathbf{r}_y), B^{(i)}(\mathbf{r}_x, \mathbf{r}_y)$, where $\mathbf{r}_x \in \{0, 1\}^{\mu_{i+1}}, \mathbf{r}_y \in \{0, 1\}^{\mu_i}$ are challenges. The verifier can check these evaluations via the following sumcheck,

$$\tilde{M}(\mathbf{r}_x, \mathbf{r}_y) := \sum_{\mathbf{j} \in \{0, 1\}^s} \tilde{val}(\mathbf{j}) \cdot \tilde{\mathbb{I}}_{row}(\mathbf{j}, \mathbf{r}_x) \cdot \tilde{\mathbb{I}}_{col}(\mathbf{j}, \mathbf{r}_y)$$

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 val, row, 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 $\tilde{M}^{(i)}(\mathbf{r}_x, \mathbf{r}_y)$ can be batched into one sumcheck using random linear combination instead of being proved individually.

Example: Inner product of $\mathbf{w}$ with constants $\mathbf{c}$. Suppose $\mathbf{w} \in H^n$ are the inputs, and $\mathbf{c} \in H^n$ is a vector of constants. We want to prove their inner product $\langle \mathbf{w}, \mathbf{c} \rangle = \sum_{i=1}^n \mathbf{c}_i \mathbf{w}_i$ 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 $z_{in} = [1, \mathbf{w}]$, $A = [0, \mathbf{c}]^T$ and $B = [1, 0^n]^T$. The prover does not need to commit to anything other than the inputs $\mathbf{w}$.

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 $\{0, 1\}^\mu := \{0, 1\}^\mu \subseteq \mathbb{F}^\mu$ to denote the μ -dimensional boolean hypercube.

We use $\mathbb{F}_\mu^{(\leq d)}$ to mean the set of μ -variate, at most degree- d polynomial. We use $\text{PERM}(H)$ to denote the set of permutation functions that map between H , and $\text{FUN}(H, H')$ 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 $\{1, 2, \dots, s\}$.

For $a, b \in \mathbb{N}$, $a \leq b$ and vector $\mathbf{x} \in \mathbb{F}^{(\geq b)}$, $\mathbf{x}_{[a:b]}$ denotes the vector slice $[\mathbf{x}_a, \mathbf{x}_{a+1}, \dots, \mathbf{x}_b]$; $\mathbf{x}_{[1:a]} := \mathbf{x}_{[1:a]}$, and $\mathbf{x}_{[a:]} = \mathbf{x}_{[a:|\mathbf{x}|]}$.

We use $\tilde{O}$ notation to hide polylogarithmic factors in asymptotic 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 $\text{eq} : \mathbb{F}^\mu \times \mathbb{F}^\mu \mapsto \mathbb{F}$ for arbitrary $\mu \in \mathbb{N}$ as

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

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

Lemma 1 (Multilinear extensions). For every function $f : \{0, 1\}^\mu \mapsto \mathbb{F}$, there is a unique multilinear polynomial $\tilde{f} \in \mathbb{F}[X_1, \dots, X_\mu]$ such that $\tilde{f}(\mathbf{b}) = f(\mathbf{b})$ for all $\mathbf{b} \in \{0, 1\}^\mu$. We call $\tilde{f}$ the multilinear extension of f , and $\tilde{f}$ can be expressed as

$$\tilde{f}(\mathbf{X}) = \sum_{\mathbf{b} \in \{0, 1\}^\mu} f(\mathbf{b}) \cdot \text{eq}(\mathbf{b}, \mathbf{X})$$

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 $r_1, \dots, r_\mu$ be μ field elements selected independently and uniformly from set S . Then*

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

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

- Perfect Completeness: for all $(\mathbf{i}; \mathbf{x}; \mathbf{w}) \in \mathcal{R}$

$$\Pr \left[\langle \text{P}(\text{pp}, \mathbf{x}, \mathbf{w}), \text{V}(\text{vp}, \mathbf{x}) \rangle = 1 \mid \begin{array}{l} \text{gp} \leftarrow \text{Setup}(1^\lambda) \\ (\text{vp}, \text{pp}) \leftarrow \mathcal{I}(\text{gp}, \mathbf{i}) \end{array} \right] = 1$$

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

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

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 $(\mathbf{i}, \mathbf{x}) \notin \mathcal{L}(\mathcal{R})$

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

- δ -Knowledge Soundness: There exists a polynomial $\text{poly}(\cdot)$ 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:

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

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.

4.2 Polynomial Interactive Oracle Reduction and Polynomial Interactive Oracle Proof

Polynomial Interactive Oracle Reductions (PIORs) are a generalization of Polynomial Interactive Oracle Proofs (PIOPs). Below we give a definition that follows from the definition of Interactive Oracle Reductions (IORs) in [warp]. Our definition is equivalent to the PIOR definition in [SSPP26].

Definition 4 (Polynomial Interactive Oracle Reduction). *A public-coin PIOR from a relation $\mathcal{R}$ to relations $\mathcal{R}[1]' \times \dots \times \mathcal{R}[n]'$ is a triple $\text{PIOR} = (\mathcal{I}, \mathbf{P}, \mathbf{V})$ that works as follows*

- *Offline phase. The indexer $\mathcal{I}$ is a deterministic algorithm that receives as input an index $\mathbf{i}$ and outputs a tuple $(\mathbf{i}', \iota, \mathbb{I})$ consisting of a new index $\mathbf{i}'$, a short index ι (explicit input for $\mathbf{V}$), and a long index $\mathbb{I}$ (implicit input for $\mathbf{V}$). Concretely, it computes oracles to multilinear polynomials in $\mathbf{i}$ and encode them as an oracle description in $\mathbb{I}$.*
- *Online phase. The prover $\mathbf{P}$ is an interactive algorithm that receives as input $(\mathbf{i}; \mathbf{x}; \mathbf{0}; \mathbf{w}) \in \mathcal{R}$, and the verifier is an interactive oracle algorithm that receives as (explicit) input $(\iota, \mathbf{x})$ and query access to $\mathbf{0}$. They interact over k rounds as follows.*

In each round $i \in [k]$, $\mathbf{P}$ sends a collection of polynomial oracles π_i and $\mathbf{V}$ sends a uniformly random message $\rho_i \in \{0, 1\}^{r_i}$. After k rounds of interaction, $\mathbf{V}$ makes queries to $\mathbf{0}$ and the oracles $\pi_1, \dots, \pi_k$ sent by $\mathbf{P}$. After that:

- *$\mathbf{V}$ outputs $\text{out}_1, \dots, \text{out}_n$. For every $i \in [n]$, out_i is either accept/reject, or can be parsed as new instances and oracle instances $(\mathbf{x}'_i, \mathbf{0}'_i)$, and*
- *$\mathbf{P}$ outputs new witnesses $\mathbf{w}'_i$ for every $i \in [n]$ such that $(\mathbf{i}'; \mathbf{x}'_i; \mathbf{0}'_i; \mathbf{w}'_i) \in \mathcal{R}'$.*

Without loss of generality, $\mathbf{V}$ ’s output is a deterministic function of its input and the transcript.

We say that PIOR has (perfect) completeness and soundness error $\epsilon : \mathbb{N} \mapsto [0, 1]$ if the following holds. **Completeness** For every $(i; \mathbb{x}; \mathbb{o}; \mathbb{w}) \in \mathcal{R}$,

$$\Pr \left[\begin{array}{l} \forall i \in [n], \text{ out}_i = \text{accept} \vee \\ \left(\text{out}_i = (\mathbb{x}'_i; \mathbb{o}'_i) \wedge (i'; \mathbb{x}'_i; \mathbb{o}'_i; \mathbb{w}_i) \in \mathcal{R}' \right) \end{array} \middle| \begin{array}{l} (i', \ell, \mathbb{I}) \leftarrow \mathcal{I}(i) \\ \pi_1 \leftarrow P(i; \mathbb{x}; \mathbb{o}; \mathbb{w}) \\ \rho_1 \leftarrow \{0, 1\}^{r_1} \\ \vdots \\ \pi_k \leftarrow P(i; \mathbb{x}; \mathbb{o}; \mathbb{w}; \rho_1, \dots, \rho_{k-1}) \\ \rho_k \leftarrow \{0, 1\}^{r_k} \\ \mathbb{w}'_1, \dots, \mathbb{w}'_n \leftarrow P(i; \mathbb{x}; \mathbb{o}; \mathbb{w}; \rho_1, \dots, \rho_k) \\ \text{out}_1, \dots, \text{out}_n \\ \leftarrow V^{\mathbb{o}, \pi_1, \dots, \pi_k}((\ell, \mathbb{I}); \mathbb{x}; \rho_1, \dots, \rho_k) \end{array} \right] = 1$$

Soundness For every $(i; \mathbb{x}; \mathbb{o}; \mathbb{w}) \notin \mathcal{R}$ and unbounded malicious prover $\tilde{P}$,

$$\Pr \left[\begin{array}{l} \forall i \in [n], \text{ out}_i = \text{accept} \vee \\ \left(\text{out}_i = (\mathbb{x}'_i; \mathbb{o}'_i) \wedge (i'; \mathbb{x}'_i; \mathbb{o}'_i; \mathbb{w}_i) \in \mathcal{R}' \right) \end{array} \middle| \begin{array}{l} (i', \ell, \mathbb{I}) \leftarrow \mathcal{I}(i) \\ \pi_1 \leftarrow \tilde{P}(i; \mathbb{x}; \mathbb{o}; \mathbb{w}) \\ \rho_1 \leftarrow \{0, 1\}^{r_1} \\ \vdots \\ \pi_k \leftarrow \tilde{P}(i; \mathbb{x}; \mathbb{o}; \mathbb{w}; \rho_1, \dots, \rho_{k-1}) \\ \rho_k \leftarrow \{0, 1\}^{r_k} \\ \mathbb{w}'_1, \dots, \mathbb{w}'_n \leftarrow P(i; \mathbb{x}; \mathbb{o}; \mathbb{w}; \rho_1, \dots, \rho_k) \\ \text{out}_1, \dots, \text{out}_n \\ \leftarrow V^{\mathbb{o}, \pi_1, \dots, \pi_k}((\ell, \mathbb{I}); \mathbb{x}; \rho_1, \dots, \rho_k) \end{array} \right] \leq \epsilon(i; \mathbb{x}, \mathbb{o})$$

Efficiency measures

- # Rounds: Number of rounds of interaction. A round is a prover message and a verifier message.
- Oracles #, Length, Size: Number of oracles sent by the prover, the total number of bits in the oracles, the number of field elements in the oracles.
- Prover Work: Number of field operations performed by the prover, excluding the work needed to compute oracles, if any.
- Proof Size: Number of elements communicated by the prover to the verifier, excluding the oracles, if any.
- Verifier Work: Number of field operations performed by the verifier.
- Verifier Oracle Queries: Number of queries made by the verifier to the oracles.

Remark 1. If a PIOR for relation $\mathcal{R}$ is of the form $\mathcal{R} \rightarrow \perp$, or equivalently, the verifier outputs nothing else other than **accept/reject** at the end of the protocol, then it is a PIOP for $\mathcal{R}$.

Remark 2. One can obtain a PIOP for relation $\mathcal{R}$ by continuing a PIOR for $\mathcal{R}$ with the PIOP(s) for the reduced relation(s).

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 $(i; \mathbb{x}, \mathbb{w}) \in \mathcal{R}$,

$\mathbb{w}$ 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 $O(|\mathbb{w}|)$.

4.3 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.

4.4 Sumcheck protocol

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

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

Remark 3. Instead of a single polynomial f , the witness w of $\mathcal{R}_{\text{SUM}}$ may contain multiple polynomials that make up a virtual polynomial f , and the oracle instance ϕ contains the polynomial oracle to each of these witness polynomials. The auxiliary data aux in the instance contains information regarding the structure of the sumcheck. aux is omitted when it is empty.

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

- For $k = 1, \dots, \mu$:
 - The prover computes $u_k(X) := \sum_{x \in \{0,1\}^{\mu-k}} f(\alpha, X, x)$, where $\alpha := [\alpha_i]_{i=1}^{k-1}$ 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 $f(\alpha) = S$.

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

Prover messages sent as oracles. Like in HyperPlonk [CBBZ23], our prover sends 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.

4.5 Multilinear polynomial commitments.

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

- $\text{Setup}(1^\lambda) \rightarrow \text{gp}$ generates public parameters gp ;
- $\text{Commit}(\text{gp}; x) \rightarrow (C; r)$ 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).
- $\text{Open}(\text{gp}, C, x, r) \rightarrow b \in \{0, 1\}$ 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 $\mathcal{A}$:

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

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

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

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 $\Gamma = (\text{Setup}, \text{Commit}, \text{Open}, \text{Eval})$ where $(\text{Setup}, \text{Commit}, \text{Open})$ is a binding commitment scheme for a message space $R[X]$ of polynomials over some ring R , and

- $\text{Eval}((\text{vp}, \text{pp}), C, z, y, d, \mu; f) \rightarrow b \in \{0, 1\}$ is an interactive public-coin protocol between a PPT prover $\mathbf{P}$ and verifier $\mathbf{V}$. Both $\mathbf{P}$ and $\mathbf{V}$ have as input a commitment C , points $z \in \mathbb{F}^\mu$ 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 $f \in \mathbb{F}_\mu^{(\leq d)}$. The protocol convinces the verifier that $f(z) = y$.

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 $f \in \mathbb{F}_\mu^{(\leq d)}$ and all points $z \in \mathbb{F}^\mu$,

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

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 $y_1, \dots, y_t$ 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 .

4.6 Permutation and its inverse

Definition 8 (Multilinear versions of permutation and its inverse).

Given a permutation $\sigma : \{0, 1\}^\mu \mapsto \{0, 1\}^\mu$, we can write it as a multilinear map:

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

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

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

such that for all $i \in [\mu]$, $\tilde{\sigma}_i^{-1}(\cdot)$ is the multilinear extension of the i th bit of $\sigma^{-1}(\cdot)$. These multilinear maps can be interpolated, treating the subscripts as explicit binary inputs, producing multilinear functions $\tilde{\sigma}_{[\mu]} : \{0, 1\}^{\mu+\log \mu} \mapsto B$ and $\tilde{\sigma}_{[\mu]}^{-1} : \{0, 1\}^{\mu+\log \mu} \mapsto B$, such that

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

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

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

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

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

Specifically, we consider their indicator functions, $\{0, 1\}^\mu \times \{0, 1\}^\mu \mapsto B$, of these relations.

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

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

$$\sum_{\mathbf{x}, \mathbf{y} \in \{0, 1\}^\mu} \mathbb{1}_\sigma(\mathbf{X}, \mathbf{Y}) \cdot \text{eq}(\mathbf{x} || \mathbf{y}, \mathbf{X} || \mathbf{Y}) = \sum_{\mathbf{x} \in \{0, 1\}^\mu} \text{eq}(\mathbf{x} || \sigma(\mathbf{x}), \mathbf{X} || \mathbf{Y})$$

however, this degree-1 formulation has 2μ variables. In the protocol, this polynomial will be committed and later opened. With the best PCS opening algorithms today, the prover cost is at least $n\lambda$ group operations.

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 $\mathbf{X}$, $\mathcal{R}_\sigma$ can be expressed as the intersection of μ relations. If we define

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

This implies that

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

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

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

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

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

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 .

5 Reduce permcheck to sumcheck

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

$$(\mathbf{i}; \mathbf{o}; \mathbf{w}) = \left((\tilde{\sigma}, [[\tilde{\sigma}]]); ([[f]], [[g]]); (f, g) \right),$$

where $\tilde{\sigma} \in \mathbb{F}_\mu^{(\leq 1)}$ is the multilinear extension of permutation $\sigma \in \text{PERM}(\{0, 1\}^\mu)$, $f, g \in \mathbb{F}_\mu^{(\leq 1)}$, and $f(\mathbf{x}) = g(\tilde{\sigma}(\mathbf{x}))$ for all $\mathbf{x} \in \{0, 1\}^\mu$.

Remark 4. Instead of $\tilde{\sigma}$, the index $\mathbf{i}$ of $\mathcal{R}_{\text{PERM}}$ may contain polynomials that make up $\tilde{\sigma}$ or different forms of the permutation σ .

In the following, we show that permcheck can be reduced to sumcheck with small error.

PIOR 5.1: $\mathcal{R}_{\text{PERM}} \rightarrow \mathcal{R}_{\text{SUM}}$

Parse: $\mathbf{i} = (\tilde{\sigma}, [[\tilde{\sigma}]]), \mathbf{o} = ([[f]], [[g]]), \mathbf{w} = (f, g)$

1. Indexer derives the indicator polynomial $\tilde{\mathbb{1}}_\sigma(\mathbf{X}, \mathbf{Y})$ with 2μ variables such that it is multilinear in the second half of its variables $\mathbf{Y}$, and its polynomial oracle $[[\tilde{\mathbb{1}}_\sigma]]$.
2. Verifier selects a random point $\alpha \xleftarrow{\$} \mathbb{F}^\mu$ and sends it to the prover.
3. Prover computes $S \leftarrow g(\alpha)$ and sends it to the verifier.
4. Output the following for $\mathcal{R}_{\text{SUM}}$:
 $\mathbf{i}' := (\tilde{\mathbb{1}}_\sigma, [[\tilde{\mathbb{1}}_\sigma]]), \quad \mathbf{x}' := (S, \text{aux} := (\alpha, \text{aux}')), \quad \mathbf{o}' := [[f]], \quad \mathbf{w}' := f$

where $\mathbf{aux}'$ includes structural information of the following equation,

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \tilde{\mathbb{1}}_\sigma(\mathbf{x}, \boldsymbol{\alpha}) = S \quad (3)$$

Theorem 3. *PIOR 5.1 has perfect completeness and soundness error $\frac{\mu}{|\mathbb{F}|}$.*

Proof. Let $n = 2^\mu$.

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

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

$$f(\mathbf{x}) = g(\sigma(\mathbf{x})) \quad \forall \mathbf{x} \in \{0,1\}^\mu.$$

Because σ is an automorphism of $\{0,1\}^\mu$, this is equivalent to

$$f(\sigma^{-1}(\mathbf{y})) = g(\mathbf{y}) \quad \forall \mathbf{y} \in \{0,1\}^\mu.$$

Using any indicator function of the inverse permutation $\mathbb{1}_{\sigma^{-1}}$, the composition on the left-hand side can be decomposed as

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \mathbb{1}_{\sigma^{-1}}(\mathbf{y}, \mathbf{x}) = g(\mathbf{y}) \quad \forall \mathbf{y} \in \{0,1\}^\mu$$

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

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \mathbb{1}_\sigma(\mathbf{x}, \mathbf{y}) = g(\mathbf{y}) \quad \forall \mathbf{y} \in \{0,1\}^\mu$$

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

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \tilde{\mathbb{1}}_\sigma(\mathbf{x}, \mathbf{y}) = g(\mathbf{y}) \quad \forall \mathbf{y} \in \{0,1\}^\mu. \quad (4)$$

A multilinear polynomial in μ variables is uniquely determined by its evaluation over $\{0,1\}^\mu$, so this check can be written as the formal equality of the polynomials on the left and right-hand sides

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \tilde{\mathbb{1}}_\sigma(\mathbf{x}, \mathbf{Y}) = g(\mathbf{Y}).$$

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

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \tilde{\mathbb{1}}_\sigma(\mathbf{x}, \boldsymbol{\alpha}) = g(\boldsymbol{\alpha})$$

Soundness The polynomial identity test at a random point $\boldsymbol{\alpha}$ has error $\frac{\mu}{|\mathbb{F}|}$ by Schwartz-Zippel Lemma.

Handling the sumcheck PIOR 5.1 reduces permcheck to the following sumcheck.

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \tilde{\mathbb{1}}_\sigma(\mathbf{x}, \boldsymbol{\alpha}) = S \quad (3)$$

How to run this sumcheck efficiently is actually nontrivial, and the efficiency is closely tied to the degree of the specific arithmetization of $\tilde{\mathbb{1}}_\sigma$. We start by exploring a naïve arithmetization of $\tilde{\mathbb{1}}_\sigma$ in the next subsection to motivate the different arithmetizations in BiPerm and MulPerm.

5.1 Degree- μ arithmetization of $\tilde{\mathbb{1}}_\sigma$

Viewing $\mathcal{R}_\sigma$ as the intersection of bit-wise relations $\bigcap_{i=1}^\mu \mathcal{R}_{\sigma_i}$, the indicator polynomial $\tilde{\mathbb{1}}_\sigma$ can be trivially arithmetized as the product of multilinear bit-wise indicator polynomials. The resulting arithmetization is degree- μ in every $\mathbf{X}_i$ and multilinear in $\mathbf{Y}$,

$$\tilde{\mathbb{1}}_\sigma(\mathbf{X}, \mathbf{Y}) = \prod_{i=1}^\mu \tilde{\mathbb{1}}_{\sigma_i}(\mathbf{X}, \mathbf{Y}) = \prod_{i=1}^\mu \text{eq}(\tilde{\sigma}_i(\mathbf{X}), \mathbf{Y}_i) \quad (5)$$

and $\tilde{\mathbb{1}}_\sigma$ and its oracle in the index for $\mathcal{R}_{\text{SUM}}$ consists of the collection of multilinear bit-wise indicator polynomials $\{\tilde{\mathbb{1}}_{\sigma_1}, \dots, \tilde{\mathbb{1}}_{\sigma_\mu}\}$. Substituting this arithmetization into Equation 3, we get the following sumcheck

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \prod_{i=1}^\mu \text{eq}(\tilde{\sigma}_i(\mathbf{x}), \boldsymbol{\alpha}_i) = S \quad (6)$$

Theorem 4. *Arithmetizing $\tilde{\mathbb{1}}_\sigma$ as described in (5), the sumcheck PIOP described by the prover and verifier in Algorithm 1 for proving $(\tilde{\mathbb{1}}_\sigma, [[\tilde{\mathbb{1}}_\sigma]]; (S, \text{aux} = (\boldsymbol{\alpha}, -)); [[f]]); f) \in \mathcal{R}_{\text{SUM}}$ has perfect completeness, soundness error $O(\frac{\log^2 n}{|\mathbb{F}|})$, and the following properties,*

# Rounds	$\log n$
# Oracles, Length, Size (i)	$\log n, n \log n, \frac{n \log n}{2}$
# Oracles, Length, Size ($\mathbf{x}$ and runtime)	$\log n + 1, n + \log n(\log n + 2), \text{Length} \cdot \log \mathbb{F} $
Prover Work	$n \cdot \tilde{O}(\log n) \mathbb{F}$
Verifier Work	$O(\log n) \mathbb{F}$
Verifier Oracle Queries	$\log n + 1$ query and 1 batch-query

Proof.

Completeness Perfect completeness is trivial and follows from the completeness of general sumcheck.

Algorithm 1 Sumcheck PIOP for equation 3 using arithmetization 5 for $\tilde{\mathbb{I}}_\sigma$

```

1: procedure SUMCHECK VERIFIER( $([[\tilde{\mathbb{I}}_{\sigma_1}]], \dots, [[\tilde{\mathbb{I}}_{\sigma_\mu}]]); (S, \text{aux}); [[f]]$ )
2:   for  $k \leftarrow 1 \dots \mu$  do
3:     Receive univariate polynomial oracle  $[[u_k]]$  from P
4:     Query  $[[u_k]]$  at 0 and 1
5:     if  $S \neq u_k(0) + u_k(1)$  then
6:       reject
7:     end if
8:     Randomly select  $\beta_k \xleftarrow{\$} \mathbb{F}$ . Send  $\beta_k$  to P
9:      $S \leftarrow u_k(\beta_k)$ 
10:  end for
11:  Query  $[[f]]$  to get  $V_f \leftarrow f(\beta)$ 
12:  Batch-query the oracles  $([[\tilde{\mathbb{I}}_{\sigma_1}]], \dots, [[\tilde{\mathbb{I}}_{\sigma_\mu}]])$  to get  $V_i \leftarrow \tilde{\mathbb{I}}_{\sigma_i}(\beta)$  for  $i \in [\mu]$ .
13:  if  $S \neq V_f \cdot \prod_{i=1}^{\mu} V_i$  then
14:    reject
15:  end if
16:  accept
17: end procedure
18:
19: procedure SUMCHECK PROVER( $(\tilde{\mathbb{I}}_{\sigma_1}, \dots, \tilde{\mathbb{I}}_{\sigma_\mu}); \text{aux}; f$ )
20:    $\beta \leftarrow []$ 
21:   for  $k \leftarrow 1 \dots \mu$  do
22:      $u_k(X) \leftarrow 0$ 
23:     for  $x \in \{0, 1\}^{\mu-k}$  do
24:        $u_x(X) \leftarrow f(\beta, X, x) \cdot \tilde{\mathbb{I}}_\sigma((\beta, X, x), \alpha)$  ▷ Using FFT
25:        $u_k(X) \leftarrow u_k(X) + u_x(X)$ 
26:     end for
27:     Send  $[[u_k]]$  to V
28:     Receive  $\beta_k$  from V
29:     Collapse the tables of  $\tilde{\mathbb{I}}_{\sigma_i}$  and  $f$ 
30:   end for
31: end procedure

```

Soundness Since the sumcheck is over a virtual polynomial with μ variables and individual degree $\mu + 1$, its soundness error is upper bounded by $\mu(\mu + 1)/|\mathbb{F}| = O(\log^2 n/|\mathbb{F}|)$.

Oracles There are μ oracles in the index, namely $[[\tilde{\mathbb{I}}_{\sigma_1}]], \dots, [[\tilde{\mathbb{I}}_{\sigma_\mu}]]$. Each of these has length n , so the total length is $n\mu$. Each oracle $\tilde{\mathbb{I}}_{\sigma_i}$ maps to $\{0, 1\}$ over the hypercube $\{0, 1\}^\mu$ and is half-sparse, which means the total size of the oracles in the index is $\frac{n\mu}{2}$.

There are one oracle in the instance, namely $[[f]]$. It has length n and size $n \log |\mathbb{F}|$ in bits. During runtime, one oracle of prover message is sent per round of sumcheck; since there are μ rounds, a total of μ oracles of sent. Each of these has length $\mu + 2$, so the total length is $\mu(\mu + 2)$, and the total size is $\mu(\mu + 2) \log |\mathbb{F}|$ in bits.

Prover Complexity 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 k th round of the sumcheck PIOP for $k \in [\mu]$, computing the polynomial $u_k(Y)$ requires summing up the polynomial $f(\beta, X, \mathbf{x}) \cdot \text{eq}(\alpha, \tilde{\sigma}(\beta, X, \mathbf{x}))$ over $\mathbf{x} \in \{0, 1\}^{\mu-k}$. The polynomial inside the sum is a product of $\mu + 1$ univariate polynomials, namely $f(\beta, X, \mathbf{x})$ and $\text{eq}(\alpha_i, \tilde{\sigma}_i(\beta, X, \mathbf{x}))$ 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 $n \cdot \tilde{O}(\log n)$. The total cost of collapsing evaluation tables is $n(\mu + 1) = n \cdot O(\log n)$ field operations.

Verifier Work Verifier does constant work per round, so the total verifier work is $O(\mu)$.

Verifier Oracle Queries The verifier queries each oracle of the prover messages, queries $[[f]]$ once, and batch queries $[[\tilde{\sigma}_1]], \dots, [[\tilde{\sigma}_\mu]]$.

5.2 Lower-degree arithmetization of $\tilde{\mathbb{I}}_\sigma$ to improve 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 k th round of sumcheck for $k \in [\mu]$, prover needs to compute the following univariate degree- μ eq

polynomial for every $\mathbf{x} \in \{0, 1\}^{\mu-k}$:

$$\begin{aligned} & \text{eq}(\boldsymbol{\alpha}, \tilde{\sigma}(\boldsymbol{\beta}, X, \mathbf{x})) \\ &= \prod_{i=1}^{\mu} \text{eq}(\boldsymbol{\alpha}_i, \tilde{\sigma}_i(\boldsymbol{\beta}, X, \mathbf{x})) \\ &= \prod_{i=1}^{\mu} \left((1 - \alpha_i) \cdot (1 - \tilde{\sigma}_i(\boldsymbol{\beta}, X, \mathbf{x})) + \alpha_i \cdot \tilde{\sigma}_i(\boldsymbol{\beta}, X, \mathbf{x}) \right) \end{aligned}$$

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 $\mathbf{X}$ in the arithmetization of $\mathbb{1}_{\sigma}(\mathbf{X}, \mathbf{Y})$. However, as pointed out previously in Section 4.6, the trivial multilinear extension of $\mathbb{1}_{\sigma}$

$$\sum_{\mathbf{x}, \mathbf{y} \in \{0, 1\}^{\mu}} \mathbb{1}_{\sigma}(\mathbf{X}, \mathbf{Y}) \cdot \text{eq}(\mathbf{x} \parallel \mathbf{y}, \mathbf{X} \parallel \mathbf{Y}) = \sum_{\mathbf{x} \in \{0, 1\}^{\mu}} \text{eq}(\mathbf{x} \parallel \sigma(\mathbf{x}), \mathbf{X} \parallel \mathbf{Y})$$

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 $\mathbb{1}_{\sigma}(\mathbf{X}, \mathbf{Y})$ that has degree smaller than μ in every $\mathbf{X}_i$.

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 relations ($\mathcal{R}_{\sigma} = \bigcap_{i=1}^{\mu} \mathcal{R}_{\sigma_i}$), 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 j th 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 $\mathbf{Y}^{(j)}$ denote the j th μ/ℓ -bit group of the any μ -sized vector $\mathbf{Y}$. Let $\mathbb{1}_j(\mathbf{X}, \mathbf{Y}^{(j)})$ be the indicator function of $\mathcal{R}_j$ for every $j \in [\ell]$

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

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

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

which is degree- ℓ in every $\mathbf{X}_i$ and multilinear in $\mathbf{Y}$.
A natural arithmetization of $\mathbb{1}_j(\mathbf{X}, \mathbf{Y}^{(j)})$ is the following,

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

Then the corresponding arithmetization of $\tilde{\mathbb{1}}_j$ is the multilinear extension $\tilde{\mathbf{p}}$. Substituting this arithmetization into Equation 3, we get the following sumcheck formulation

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \prod_{j=1}^{\ell} \tilde{\mathbf{p}}(\mathbf{x}, \boldsymbol{\alpha}^{(j)}, j) = v$$

The prover can either have the indexer preprocess and commit to every $\tilde{\mathbb{1}}_j$ (in BiPerm), or send the verifier $[[\tilde{\mathbf{p}}]]$ as part of the instance and prove the correctness of this multilinear extension (in MulPerm).

6 BiPerm: Linear-time permutation check for sparse PCS

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

BiPerm arithmetization of $\tilde{\mathbb{1}}_\sigma$: 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, each with 1.5μ variables; and let $\tilde{\mathbb{1}}_{\sigma_L}$ and $\tilde{\mathbb{1}}_{\sigma_R}$ be their multilinear extensions. By definition, $\mathcal{R}_\sigma = \mathcal{R}_{\sigma_L} \cap \mathcal{R}_{\sigma_R}$, which implies $\tilde{\mathbb{1}}_\sigma$ can be arithmetized as the product of the multilinear extensions of the left and right indicator polynomials,

$$\tilde{\mathbb{1}}_\sigma(\mathbf{X}, \mathbf{Y}) = \tilde{\mathbb{1}}_{\sigma_L}(\mathbf{X}, \mathbf{Y}_L) \cdot \tilde{\mathbb{1}}_{\sigma_R}(\mathbf{X}, \mathbf{Y}_R) \quad (7)$$

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

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot \tilde{\mathbb{1}}_{\sigma_L}(\mathbf{x}, \boldsymbol{\alpha}_L) \cdot \tilde{\mathbb{1}}_{\sigma_R}(\mathbf{x}, \boldsymbol{\alpha}_R) = S \quad (8)$$

The sumcheck PIOP for Equation 8 is described in Algorithm 2. The claims in Theorem 5 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.

Theorem 5 (Sumcheck in BiPerm). *Arithmetizing $\tilde{\mathbb{1}}_\sigma$ as described in (7), the sumcheck PIOP described by the prover and verifier in Algorithm 2 for proving*

$((\tilde{\mathbb{I}}_{\sigma_L}, \tilde{\mathbb{I}}_{\sigma_R}, [[\tilde{\mathbb{I}}_{\sigma_L}]], [[\tilde{\mathbb{I}}_{\sigma_R}]]); (S, \mathbf{aux} = (\boldsymbol{\alpha}, -)); [[f]]; f) \in \mathcal{R}_{\text{SUM}}$ has perfect completeness, $\frac{3 \log n}{|\mathbb{F}|}$ soundness error, and the following properties

# Rounds	$\log n$
# Oracles, Length, Size (i)	$2, n^{1.5}, 2n$
# Oracles, Length, Size (∞ and runtime)	$\log n + 1, n + 4 \log n, \text{Length} \cdot \log \mathbb{F} $
Prover Work	$O(n) \mathbb{F}$
Verifier Work	$O(\log n) \mathbb{F}$
Verifier Oracle Queries	$\log n + 3 \text{ queries}$

Proof. Let $n = 2^\mu$.

Completeness. Perfect completeness trivially follows from the completeness of general sumcheck.

Soundness. The sumcheck is over a virtual polynomial that has μ variables and degree at most 3 in each variable, so its soundness error is at most $3\mu/|\mathbb{F}|$.

Oracles. There are two oracles in the index, namely $[[\tilde{\mathbb{I}}_{\sigma_L}]], [[\tilde{\mathbb{I}}_{\sigma_R}]]$. Each of them is size $n^{1.5}$ and n -sparse.

There is one oracle $[[f]]$ in the instance, which contains n field element. During runtime, one oracle of prover message is sent per round of sumcheck, which means there are $\log n$ oracles sent in total, each of which contains four field elements since the prover message is univariate and degree-3.

Prover Work. Provided the evaluations of $\tilde{\mathbb{I}}_{\sigma_L}(\mathbf{X}, \boldsymbol{\alpha}_L)$ and $\tilde{\mathbb{I}}_{\sigma_R}(\mathbf{X}, \boldsymbol{\alpha}_R)$ over $\{0, 1\}^\mu$, 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 $\tilde{\mathbb{I}}_{\sigma_L}(\mathbf{X}, \boldsymbol{\alpha}_L)$ and $\tilde{\mathbb{I}}_{\sigma_R}(\mathbf{X}, \boldsymbol{\alpha}_R)$ with $O(n)$ work.

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

$$\begin{aligned}
\tilde{\mathbb{I}}_{\sigma_L}(\mathbf{X}, \mathbf{Y}_L) &= \sum_{\mathbf{x} \in \{0, 1\}^\mu, \mathbf{y}_L \in \{0, 1\}^{\mu/2}} \text{eq}(\mathbf{x} || \mathbf{y}_L, \mathbf{X} || \mathbf{Y}_L) \mathbb{I}_{\sigma_L}(\mathbf{x}, \mathbf{y}_L) \\
&= \sum_{\mathbf{x} \in \{0, 1\}^\mu} \text{eq}(\mathbf{x} || \sigma_L(\mathbf{x}), \mathbf{X} || \mathbf{Y}_L) \\
&= \sum_{\mathbf{x} \in \{0, 1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{X}) \cdot \text{eq}(\sigma_L(\mathbf{x}), \mathbf{Y}_L).
\end{aligned}$$

When $\mathbf{X} \in \{0, 1\}^\mu$, the only non-zero term in the sum is the case where $\mathbf{x} = \mathbf{X}$, so the expression simplifies as $\text{eq}(\sigma_L(\mathbf{X}), \mathbf{Y}_L)$.

It is well known that $\text{eq}(\mathbf{y}_L, \boldsymbol{\alpha}_L)$ can be evaluated at all points $\mathbf{y}_L \in \{0, 1\}^{\mu/2}$ in time $O(2^{\mu/2})$. As $\sigma_L(\mathbf{X})$ exclusively takes on values in $\{0, 1\}^{\mu/2}$, each evaluation of $\tilde{\mathbb{I}}_{\sigma_L}(\mathbf{x}, \boldsymbol{\alpha}_L) \forall \mathbf{x} \in \{0, 1\}^{\mu/2}$ is merely a lookup into a table of $\text{eq}(\mathbf{y}_L, \boldsymbol{\alpha}_L)$'s evaluations.

Computing $\text{eq}(\mathbf{y}_L, \boldsymbol{\alpha}_L) \forall \mathbf{y}_L \in \{0, 1\}^{\mu/2}$ 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.

Verifier Work Verifier does constant work per round, so the total verifier work is $O(\mu)$.

Verifier Oracle Queries The verifier queries each oracle of the prover messages, and queries $[[f]], [[\tilde{\mathbb{I}}_{\sigma_L}]], [[\tilde{\mathbb{I}}_{\sigma_R}]]$ once each.

Algorithm 2 Sumcheck PIOP in BiPerm for Equation3 using arithmetization 7 for $\tilde{\mathbb{I}}_{\sigma}$

```

1: procedure BIPERM VERIFIER( $(([\tilde{\mathbb{I}}_{\sigma_L}]), [[\tilde{\mathbb{I}}_{\sigma_R}]]); (S, \text{aux}); [[f]]$ )
2:    $\beta \leftarrow []$ 
3:   for  $k \leftarrow 1 \dots \mu$  do
4:     Receive  $[[u_k]]$  from P
5:     Query  $[[u_k]]$  at 0 and 1
6:     if  $S \neq u_k(0) + u_k(1)$  then
7:       reject
8:     end if
9:     Randomly select  $\beta_k \xleftarrow{\$} \mathbb{F}$ . Append to  $\beta$ . Send  $\beta_k$  to P
10:     $S \leftarrow u_k(\beta_k)$ 
11:  end for
12:  Query  $[[f]]$  to get  $V_f \leftarrow f(\beta)$ 
13:  Query  $[[\tilde{\mathbb{I}}_{\sigma_L}]]$  to get  $V_{\sigma_L} \leftarrow \tilde{\mathbb{I}}_{\sigma_L}(\beta, \alpha_{[\mu/2:]})$ 
14:  Query  $[[\tilde{\mathbb{I}}_{\sigma_R}]]$  to get  $V_{\sigma_R} \leftarrow \tilde{\mathbb{I}}_{\sigma_R}(\beta, \alpha_{[\mu/2+1:]})$ 
15:  if  $S \neq V_f \cdot V_{\sigma_L} \cdot V_{\sigma_R}$  then
16:    reject
17:  end if
18:  accept
19: end procedure
20:
21: procedure BIPERM PROVER( $(\tilde{\mathbb{I}}_{\sigma_L}, \tilde{\mathbb{I}}_{\sigma_R}); \text{aux}; f$ )
22:   Preprocessed tables with evaluations of  $f$ ,  $\tilde{\mathbb{I}}_{\sigma_L}(\cdot, \alpha_L)$  and  $\tilde{\mathbb{I}}_{\sigma_R}(\cdot, \alpha_R)$  over  $\{0, 1\}^{\mu}$ 
23:    $\beta \leftarrow []$ 
24:   for  $k \leftarrow 1 \dots \mu$  do
25:      $u_k(X) \leftarrow 0$ 
26:     for  $x \in \{0, 1\}^{\mu-k}$  do
27:        $u_x(X) \leftarrow f(\beta, X, x) \cdot \tilde{\mathbb{I}}_{\sigma_L}((\beta, X, x), \alpha_L) \cdot \tilde{\mathbb{I}}_{\sigma_R}((\beta, X, x), \alpha_R) \quad \triangleright \text{FFT}$ 
28:        $u_k(X) \leftarrow u_k(X) + u_x(X)$ 
29:     end for
30:     Send  $[[u_k(X)]]$  to V
31:     Receive  $\beta_k$  from V. Append to  $\beta$ 
32:     Collapse the evaluation tables of  $\tilde{\mathbb{I}}_{\sigma_L}, \tilde{\mathbb{I}}_{\sigma_R}, f$ 
33:   end for
34: 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 $\tilde{\mathbb{I}}_{\sigma_L}$ and $\tilde{\mathbb{I}}_{\sigma_R}$ 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 of the oracles in the index, $[[\tilde{\mathbb{I}}_{\sigma_L}]], [[\tilde{\mathbb{I}}_{\sigma_R}]]$, is larger than n . Without such PCS, committing to and opening the two 1.5μ -variate sub-indicator functions $(\tilde{\mathbb{I}}_{\sigma_L}, \tilde{\mathbb{I}}_{\sigma_R})$ 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]. Specifically, MulPerm arithmetizes $\tilde{\mathbb{I}}_{\sigma}$ as the product of ℓ sub-indicator functions, each of which dependent on μ/ℓ bits of $\mathbf{Y}$, where $\ell \leq \mu$ is some fixed parameter. Looking ahead, we eventually set $\ell = \sqrt{\mu}$ for MulPerm to minimize the prover cost.

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

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

Let $\mathbb{1}_j$ be the indicator function for $\mathcal{R}_j$ and $\tilde{\mathbb{I}}_j$ be its multilinear extension. Let $\mathbf{Y}^{(j)}$ denote the j th μ/ℓ -bit group of the any μ -sized vector $\mathbf{Y}$. By definition, $\mathcal{R}_{\sigma} = \bigcap_{j \in [\ell]} \mathcal{R}_j$, which implies $\tilde{\mathbb{I}}_{\sigma}$ can be arithmetized as the product of the multilinear extensions of all ℓ indicator polynomials. Let $\mathbf{Y}^{(j)}$ denote the j th

μ/ℓ -bit group of the any μ -sized vector $\mathbf{Y}$.

$$\tilde{\mathbb{I}}_\sigma(\mathbf{X}, \mathbf{Y}) = \prod_{j \in [\ell]} \tilde{\mathbb{I}}_j(\mathbf{X}, \mathbf{Y}^{(j)}) \quad (9)$$

Observe that since we are no longer preprocessing $\tilde{\mathbb{I}}_j$, we can hardcode the slice of α into the arithmetization. Particularly, interpret the random challenge $\alpha \in \mathbb{F}^\mu$ as a polynomial $\alpha : \mathbb{F}^{\log \mu} \mapsto \mathbb{F}$, such that $\alpha(\langle i \rangle) = \alpha_i$ for $i \in [\mu]$. Define partial product polynomial $\mathbf{p} : \{0, 1\}^{\mu + \log \ell} \mapsto \mathbb{F}$ as the following,

$$\mathbf{p}(\mathbf{x}, \mathbf{j}) := \prod_{i=1}^{\mu/\ell} \text{eq}(\tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, \mathbf{x}), \alpha(\langle j' + i \rangle)) \quad (10)$$

where $\langle j' + i \rangle$ denotes the $\log \mu$ -bit representation of $j' + i - 1$. This polynomial $\mathbf{p}(\mathbf{X}, \langle j \rangle)$ is the arithmetization of $\mathbb{I}_j(\mathbf{X}, \alpha^{(j)})$ for every $j \in [\ell]$. The arithmetization of the multilinear extension $\tilde{\mathbb{I}}_j$ is then simply:

$$\tilde{\mathbf{p}}(\mathbf{x}, \mathbf{j}) = \sum_{\mathbf{x}^* \in \{0, 1\}^\mu, \mathbf{j}^* \in \{0, 1\}^{\log \ell}} \text{eq}((\mathbf{x}^*, \mathbf{j}^*), (\mathbf{x}, \mathbf{j})) \cdot \mathbf{p}(\mathbf{x}^*, \mathbf{j}^*) \quad (11)$$

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

$$\sum_{\mathbf{x} \in \{0, 1\}^\mu} f(\mathbf{x}) \cdot \prod_{\mathbf{j} \in \{0, 1\}^{\log \ell}} \tilde{\mathbf{p}}(\mathbf{x}, \mathbf{j}) = S \quad (12)$$

As $\tilde{\mathbf{p}}$ is not preprocessed, the prover will need to prove that $\tilde{\mathbf{p}}$ is indeed the multilinear extension of $\mathbf{p}$, which can be accomplished by another sumcheck. Therefore, the PIOP for proving Equation 3 in **MulPerm** is a double-sumcheck PIOP.

Theorem 6 (Double-Sumcheck in MulPerm). *Arithmetizing $\tilde{\mathbb{I}}_\sigma$ as described in (9), the double-sumcheck PIOP described by the prover and verifier in Algorithm 3 for proving $((\tilde{\sigma}_{[\mu]}, [[\tilde{\sigma}_{[\mu]}]]); (S, \text{aux} = (\alpha, _)); [[f]]); f) \in \mathcal{R}_{\text{SUM}}$ has perfect completeness, $\frac{\text{polylog}(n)}{|\mathbb{F}|}$ soundness error, and the following properties:*

# Rounds	$O(\log n)$
# Oracles, Length, Size (i)	$1, n \log n, \frac{n \log n}{2}$
# Oracles, Length, Size (x and runtime)	$O(\log n), O(\log n \sqrt{\log n}), \text{Length} \cdot \log \mathbb{F} $
Prover Work	$n \cdot \tilde{O}(\sqrt{\log n}) \mathbb{F}$
Verifier Work	$O(\log n) \mathbb{F}$
Verifier Oracle Queries	$O(\log n)$ queries and 1 batch-query

Algorithm 3 Double-sumcheck PIOP in MulPerm for Equation 3 using arithmetization 9 for $\tilde{\mathbb{L}}_\sigma$. $\tilde{\mathbf{p}}$ is defined as in Equation 11.

```

1: procedure MULPERM VERIFIER( $[[\tilde{\sigma}_{[\mu]}]]$ ;  $(S, \mathbf{aux})$ ;  $[[f]]$ )
2:   SUMCHECK1 VERIFIER( $S$ ;  $[[f]]$ ). If procedure rejects, output reject. Otherwise,
   gets returned values  $[P_j]_{j=1}^\ell$  and  $\beta \in \mathbb{F}^\mu$ .
3:   Randomly select  $\mathbf{t} \in \mathbb{F}^{\log \ell}$ . Send  $\mathbf{t}$  to P
4:   Compute  $S_{\tilde{\mathbf{p}}} \leftarrow \sum_{j \in [\ell]} \text{eq}(\mathbf{t}, \langle j \rangle) \cdot P_j$ 
5:   Define  $\mathbf{aux}' := (\mathbf{aux}, \beta || \mathbf{t})$ .
6:   SUMCHECK2 VERIFIER( $[[\tilde{\sigma}_{[\mu]}]]$ ;  $(S_{\tilde{\mathbf{p}}}, \mathbf{aux}')$ ).
7: end procedure
8:
9: procedure MULPERM PROVER( $\tilde{\sigma}_{[\mu]}$ ;  $(S, \mathbf{aux})$ ;  $f$ )
10:  Parse  $\mathbf{aux}$  as  $(\alpha \in \mathbb{F}^\mu, -)$ 
11:  Evaluation table of  $\tilde{\mathbf{p}}$  over  $\{0, 1\}^{\mu + \log \ell} \leftarrow \text{COMPUTE PARTIAL PRODUCTS}(\tilde{\sigma}_{[\mu]}, \alpha)$ 
12:   $\beta \in \mathbb{F}^\mu \leftarrow \text{SUMCHECK1 PROVER}(\mathbf{aux}; (f, \tilde{\mathbf{p}}))$ 
13:  Receive  $\mathbf{t} \in \mathbb{F}^{\log \ell}$  from V. Define  $\mathbf{aux}' := (\mathbf{aux}, \beta || \mathbf{t})$ .
14:  SUMCHECK2 PROVER( $\tilde{\sigma}_{[\mu]}$ ;  $\mathbf{aux}'$ )
15: end procedure

```

The double-sumcheck PIOP described in Algorithm 3 consists of three parts.

1. Prover calls `ComputePartialProducts` method to construct the evaluation table for $\tilde{\mathbf{p}}$, namely computing $\tilde{\mathbf{p}}(\mathbf{x}')$ for every $\mathbf{x}' \in \{0, 1\}^{\mu + \log \ell}$. The details of this algorithm this is described in Section 7.1.
2. Prover and verifier run a sumcheck PIOP to reduce

$$\sum_{\mathbf{x} \in \{0, 1\}^\mu} f(\mathbf{x}) \prod_{j \in [\ell]} \tilde{\mathbf{p}}(\mathbf{x}, \langle j \rangle) = S \quad (12)$$

to ℓ claims, $\tilde{\mathbf{p}}(\beta, \langle j \rangle) = P_j$ for every $j \in [\ell]$. The details of the first sumcheck is described in Section 7.2.

3. Verifier sends random challenges $\mathbf{t} \in \mathbb{F}^{\log \ell}$ to prover and computes $S_{\tilde{\mathbf{p}}} \leftarrow \sum_{j \in [\ell]} \text{eq}(\mathbf{t}, \langle j \rangle) \cdot P_j$. Let $\beta' := \beta || \mathbf{t}$. Prover and verifier engage in another sumcheck to prove $\tilde{\mathbf{p}}(\beta') = S_{\tilde{\mathbf{p}}}$, or equivalently,

$$\sum_{\mathbf{x} \in \{0, 1\}^\mu, j \in [\ell]} \text{eq}(\beta', \mathbf{x} || \langle j \rangle) \underbrace{\prod_{i=1}^{\mu/\ell} \text{eq}(\alpha(\langle j' + i \rangle), \tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, \mathbf{x}))}_{\tilde{\mathbf{p}}(\mathbf{x}, \langle j \rangle)} = S_{\tilde{\mathbf{p}}} \quad (13)$$

This second sumcheck is described in Section 7.3.

Remark 5. 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 12 and the lower layer computes $\tilde{\mathbf{p}}(\mathbf{x}')$ for every $\mathbf{x}'$ in the boolean hypercube. The two ideas are equivalent.

In the following subsections, we 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 $\tilde{\mathbf{p}}$ over the boolean hypercube

The prover needs to compute $\tilde{\mathbf{p}}(\mathbf{x}')$ for every $\mathbf{x}' \in \{0, 1\}^{\mu + \log \ell}$ before engaging in the sumcheck protocols. Notice that because $\tilde{\mathbf{p}}(\mathbf{x}') = \mathbf{p}(\mathbf{x}')$ over the Boolean hypercube, the prover can simply compute the partial product polynomials $\mathbf{p}$.

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 $\mathbf{eq}$ polynomial only has image space of size 2. Thus, the image space of $\mathbf{p}$, which is defined as a product of μ/ℓ bit-wise $\mathbf{eq}$ 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 $\tilde{\mathbf{p}}$.

Lemma 4. *Algorithm 4 computes $\tilde{\mathbf{p}}(\mathbf{x}')$ for all $\mathbf{x}' \in \{0, 1\}^{\mu + \log \ell}$ using $o(n)$ field operations.*

Proof. For any fixed $j \in [\ell]$, there are at most $2^{\mu/\ell}$ possible evaluations across all different $\mathbf{x} \in \{0, 1\}^{\mu/\ell}$. 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 $\mathbf{p}$. 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 $\tilde{\mathbf{p}}(\mathbf{x}')$ for every $\mathbf{x}' \in \{0, 1\}^{\mu + \log \ell}$

```

1: procedure COMPUTEPARTIALPRODUCTS( $\tilde{\sigma}_{[\mu]}, \alpha \in \mathbb{F}^\mu$ )
2:   Preprocessed evaluation tables of  $\tilde{\sigma}_{[\mu]}$  over  $\{0, 1\}^{\mu + \log \ell}$ 
3:   Let  $\{\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_{2^{\mu/\ell}}\}$  be the set of all  $\mu/\ell$ -bit strings
4:   Initialize empty evaluation table of  $\tilde{\mathbf{p}}$ 
5:   Initialize table  $\mathbf{S}$  of size  $2^{\mu/\ell} \times \ell$ 
6:   for  $i \leftarrow 1 \dots 2^{\mu/\ell}, j \in [\ell]$  do
7:     Compute  $\mathbf{S}_{ij} \leftarrow \mathbf{eq}(\alpha[j' + 1 : j' + \mu/\ell], \mathbf{s}_i)$ 
8:   end for
9:   for  $\mathbf{x} \in \{0, 1\}^\mu, j \in [\ell]$  do
10:    Look up  $i$  such that  $\mathbf{s}_i = [\tilde{\sigma}_{[\mu]}(\langle j' + 1 \rangle, \mathbf{x}), \dots, \tilde{\sigma}_{[\mu]}(\langle j' + \mu/\ell \rangle, \mathbf{x})]$ 
11:    Look up  $\mathbf{S}_{ij}$ , and use it to fill in the evaluation of  $\tilde{\mathbf{p}}(\mathbf{x}, \langle j \rangle)$ 
12:   end for
13:   Return the evaluation table of  $\tilde{\mathbf{p}}$ 
14: end procedure

```

7.2 First sumcheck

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \prod_{\mathbf{j} \in \{0,1\}^{\log \ell}} \tilde{\mathbf{p}}(\mathbf{x}, \mathbf{j}) = S \quad (12)$$

Lemma 5. *The sumcheck PIOP described by the prover and verifier in Algorithm 5 reduces Equation 12 to ℓ claims $\tilde{\mathbf{p}}(\beta, \mathbf{j}) = P_j$ for every $\mathbf{j} \in \{0,1\}^{\log \ell}$, where $\beta \in \mathbb{F}^\mu$ are challenges sent by the verifier. It has perfect completeness, soundness error $\frac{\mu(\ell+1)}{|\mathbb{F}|}$, and the following properties,*

# Rounds	$\log n$
# Oracles, Length, Size ($\mathbb{X}$ and runtime)	$\log n + 1, n + (\ell + 2) \log n, \text{Length} \cdot \log \mathbb{F} $
Prover Work	$n \cdot \tilde{O}(\ell) \mathbb{F}$
Verifier Work	$O(\ell) \mathbb{F}$
Verifier Oracle Queries	$\log n + 1$ query

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

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

Prover Complexity In the k th round for $1 \leq k \leq \mu$, there are $2^{\mu-k} \cdot (\ell + 1)^2$ field multiplications and $2^{\mu-k}$ additions. In the $(\mu + k)$ th round for $1 \leq k \leq \log \ell$, 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 MulPerm

```

1: procedure SUMCHECK1 VERIFIER( $(S, \text{aux}); [[f]]$ )
2:    $\beta \leftarrow []$ 
3:   for  $k \leftarrow 1 \dots \mu$  do
4:     Receive  $[[u_k]]$  from P. Query  $[[u_k]]$  at 0 and 1.
5:     if  $S \neq u_k(0) + u_k(1)$  then
6:       reject
7:     end if
8:     Randomly select  $\beta_k \xleftarrow{\$} \mathbb{F}$ . Append to  $\beta$ . Send  $\beta_k$  to P
9:      $S \leftarrow u_k(\beta_k)$ 
10:  end for
11:  Query  $[[f]]$  to get  $V_f \leftarrow f(\beta)$ 
12:  Receive  $[P_j]_{j=1}^\ell$  from P
13:  if  $S \neq V_f \cdot \prod_{j \in [\ell]} P_j$  then
14:    reject
15:  end if
16:  return  $[P_j]_{j=1}^\ell, \beta$ 
17: end procedure

18:
19: procedure SUMCHECK1 PROVER( $\text{aux}; (f, \tilde{\mathbf{p}})$ )
20:    $\beta \leftarrow []$ 
21:   for  $k \leftarrow 1 \dots \mu$  do
22:      $u_k(X) \leftarrow 0$ 
23:     for  $x \in \{0, 1\}^{\mu-k}$  do
24:        $u_x(X) \leftarrow f(\beta, X, x) \cdot \prod_{j \in \{0, 1\}^{\log \ell}} \tilde{\mathbf{p}}((\beta, X, x), j)$  ▷ Using FFT
25:        $u_k(X) \leftarrow u_k(X) + u_x(X)$ 
26:     end for
27:     Send  $[[u_k]]$  to V
28:     Receive  $\beta_k$  from V and append it to  $\beta$ 
29:     Collapse the evaluation tables
30:   end for
31:   Send  $P_j := \tilde{\mathbf{p}}(\beta, \langle j \rangle)$  for every  $j \in [\ell]$  to V
32:   return  $\beta$ 
33: end procedure

```

7.3 Second sumcheck

$$\sum_{\mathbf{x} \in \{0,1\}^\mu, \mathbf{j} \in \{0,1\}^{\log \ell}} \text{eq}(\beta', \mathbf{x} || \langle j \rangle) \underbrace{\prod_{i=1}^{\mu/\ell} \text{eq}(\alpha(\langle j' + i \rangle), \tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, \mathbf{x}))}_{\tilde{\mathbf{p}}(\mathbf{x}, \langle j \rangle)} = S_{\tilde{\mathbf{p}}} \quad (13)$$

The second sumcheck has degree $\mu/\ell + 1$ in each $\mathbf{x}_i$ and $2(\mu/\ell + 1)$ in each j_i , 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 $\text{eq}(\alpha(i), \sigma(i, \cdot))$ 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 $\tilde{\mathbf{p}}$ polynomial, then instead of computing them individually for every $\mathbf{x}'$ in every round of sumcheck, we can permute the buckets and then sum up the product of each identity with the polynomials $\text{eq}((\gamma, X, \mathbf{x}'), \beta')$ for $\mathbf{x}' \in \{0,1\}^{\mu+\log \ell-k}$ such that the corresponding partial product falls in the current bucket.

Lemma 6. *The sumcheck PIOP for proving Equation 13 described by the prover and verifier in Algorithm 7 has perfect completeness, soundness error $\frac{(\mu/\ell+1)(\mu+\log \ell)}{|\mathbb{F}|}$, and the following properties:*

# Rounds	$\log n + \log \ell$
# Oracles, Length, Size (i)	1, $n \log n$, $\text{Length} \frac{n \log n}{2}$
# Oracles, Length, Size ($\mathbf{x}$ and runtime)	$\mu + \log \ell$, $(\mu + \log \ell)(\mu/\ell + 2)$, $\text{Length} \cdot \log \mathbb{F} $
Prover Work	$n \cdot \tilde{O}(\mu/\ell) + \ell 2^\ell \mathbb{F}$
Verifier Work	$O(\mu + \log \ell) \mathbb{F}$
Verifier Oracle Queries	$\log n + \log \ell + 1$ batch-query

Bucketing Start by considering the first round of sumcheck, where the prover needs to compute $\sum_{\mathbf{x}' \in \{0,1\}^{\mu+\log \ell-1}} \text{eq}((X, \mathbf{x}'), \beta') \tilde{\mathbf{p}}(X, \mathbf{x}')$ or equivalently

$$\sum_{\mathbf{x} \in \{0,1\}^{\mu-1}, j \in [\ell]} \text{eq}((X, \mathbf{x}, \langle j \rangle), \beta') \tilde{\mathbf{p}}((X, \mathbf{x}), \langle j \rangle)$$

Recall that $\tilde{\mathbf{p}}$ is the multilinear extension of the product of μ/ℓ bit-wise eq between α 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 $\tilde{\mathbf{p}}((X, \mathbf{x}), \langle j \rangle)$, regardless of the value of $\mathbf{x}$. Let these identities be $\{\text{id}_1, \dots, \text{id}_{2^{2\mu/\ell} \cdot \ell}\}$. We can

transform the equation to the following:

$$\sum_{i=1}^{2^{2^{\mu/\ell} \cdot \ell}} \text{id}_i \sum_{\substack{\mathbf{x} \in \{0,1\}^{\mu-1}, j \in [\ell] \\ \text{s.t. } \tilde{\mathbf{p}}((X, \mathbf{x}), \langle j \rangle) = \text{id}_i}} \text{eq}((X, \mathbf{x}, \langle j \rangle), \beta')$$

In the k th round of sumcheck, the prover needs to compute the following univariate polynomial

$$\sum_{\mathbf{x} \in \{0,1\}^{\mu-k}, j \in [\ell]} \text{eq}((\gamma, X, \mathbf{x}, \langle j \rangle), \beta') \tilde{\mathbf{p}}((\gamma, X, \mathbf{x}), \langle j \rangle)$$

where $\gamma \in \mathbb{F}^{k-1}$ 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 $\tilde{\mathbf{p}}$ grows to $2^{2^k \mu/\ell} \cdot \ell$. Let these identities be $\{\text{id}_1, \dots, \text{id}_{2^{2^k \mu/\ell} \cdot \ell}\}$. Now we transform the equation to:

$$\sum_{i=1}^{2^{2^k \mu/\ell} \cdot \ell} \text{id}_i \sum_{\substack{\mathbf{x} \in \{0,1\}^{\mu-k}, j \in [\ell] \\ \text{s.t. } \tilde{\mathbf{p}}((\gamma, X, \mathbf{x}), \langle j \rangle) = \text{id}_i}} \text{eq}((\gamma, X, \mathbf{x}, \langle j \rangle), \beta)$$

Using the bucketing algorithm (Algorithm 6), the cost of computing the prover message in the k th 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 $\tilde{\sigma}(\langle i \rangle, (\gamma, \mathbf{x}))$ 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 A, 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 k th round of the second sumcheck is always

Algorithm 6 Bucketing Algorithm for Efficiently Computing the Univariate Prover Message in the k th round of the Second Sumcheck in **MulPerm**

```

1: procedure BUCKET( $\tilde{\sigma}_{[\mu]}, \alpha \in \mathbb{F}^\mu, \beta' \in \mathbb{F}^{\mu+\log \ell}, \gamma \in \mathbb{F}^{k-1}, k$ )
2:   Let  $S$  be the set of all  $2^k \mu/\ell$ -bit strings.
3:   Initialize  $\ell$  tables  $t_1, \dots, t_\ell$ , 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  $j \in [\ell], s \in S$  do
5:     for  $i \in [\mu/\ell - 1]$  do
6:        $\mathbf{b} := (b_0, b_1, \dots, b_{2^k}) \leftarrow s[2^k(i-1) + 1 : 2^k i]$ .
7:       View  $\mathbf{b}$  as the evaluation table of a  $k$ -variate polynomial. Fold  $\mathbf{b}$  such that
       the first  $k-1$  variates are replaced by  $\gamma$ .  $\text{id}_\sigma$  is the final univariate polynomial.
8:       Univariate polynomial  $\text{id} \leftarrow \text{id} \cdot \text{eq}(\alpha(\langle j' + i \rangle), \text{id}_\sigma)$ 
9:     end for
10:    Fill the cell  $t_j[s, 1]$  with  $\text{id}$ 
11:  end for
12:  for  $x \in \{0, 1\}^{\mu-k}, j \in [\ell]$  do
13:    Initialize  $s$  as an empty string
14:    for  $i \in [\mu/\ell], x_L \in \{0, 1\}^k$  do
15:      Lookup  $\tilde{\sigma}_{[\mu]}(\langle j' + i \rangle, x_L || x)$  and append to  $s$ 
16:    end for
17:    Add  $x$  to the cell  $t_j[s, 2]$ 
18:  end for
19:   $u_k(X) \leftarrow 0$ 
20:  for  $s \in S, j \in [\ell]$  do
21:     $\text{id} \leftarrow t_j[s, 1]$ 
22:     $u_{\text{id}}(X) \leftarrow 0$ 
23:    for  $x \in t_j[s, 2]$  do
24:       $u_{\text{id}}(X) \leftarrow u_{\text{id}}(X) + \text{eq}((\gamma, X, x, \langle j \rangle), \beta')$ 
25:    end for
26:     $u_k(X) \leftarrow u_k(X) + \text{id} \cdot u_{\text{id}}(X)$ 
27:  end for
28:  return  $u_k(X)$ 
29: end procedure

```

Algorithm 7 Second Sumcheck PIOP in MulPerm

```

1: procedure SUMCHECK2 VERIFIER( $[[\tilde{\sigma}_{[\mu]}]]$ ;  $(S, \text{aux})$ )
2:   Parse  $\text{aux}$  as  $(\alpha \in \mathbb{F}^\mu, \beta' \in \mathbb{F}^{\mu+\log \ell}, \_)$ 
3:    $\gamma \leftarrow []$ 
4:   for  $k \leftarrow 1 \dots \mu + \log \ell$  do
5:     Receive  $[[u_k]]$  from P
6:     if  $S \neq u_k(0) + u_k(1)$  then
7:       reject
8:     end if
9:     Randomly select  $\gamma_k \xleftarrow{\$} \mathbb{F}$ . Append to  $\gamma$ . Send  $\gamma_k$  to P
10:     $S \leftarrow u_k(\gamma_k)$ 
11:  end for
12:   $\mathbf{x}^* \leftarrow \gamma[:\mu]$ ,  $\mathbf{j}^* \leftarrow \gamma[\mu+1:]$ 
13:  Batch-query  $[[\tilde{\sigma}_{[\mu]}]]$  to get  $V_i := \tilde{\sigma}(\mathbf{j}^* \cdot \langle \mu/\ell \rangle + \langle i \rangle, \mathbf{x}^*)$  for every  $i \in [\mu/\ell]$ 
14:  if  $S \neq \prod_{i \in [\mu/\ell]} \text{eq}(\alpha(\mathbf{j}^* \cdot \langle \mu/\ell \rangle + \langle i \rangle), V_i)$  then
15:    reject
16:  end if
17:  accept
18: end procedure

19:
20: procedure SUMCHECK2 PROVER( $\tilde{\sigma}_{[\mu]}$ ;  $\text{aux}$ )
21:   Parse  $\text{aux}$  as  $(\alpha \in \mathbb{F}^\mu, \beta' \in \mathbb{F}^{\mu+\log \ell}, \_)$ 
22:   Preprocessed evaluation tables of  $\tilde{\sigma}_{[\mu]}(\langle i \rangle, \mathbf{x})$  over  $\{0, 1\}^\mu$  for every  $i \in [\mu]$ 
23:    $\gamma \leftarrow []$ 
24:   for  $k \leftarrow 1 \dots \log \ell - 1$  do
25:      $u_k(X) \leftarrow \text{BUCKET}(\tilde{\sigma}_{[\mu]}, \alpha, \beta', \gamma, k)$  ▷ Using Bucketing Algorithm
26:     Send  $[[u_k]]$  to V
27:     Receive  $\gamma_k$  from V and append it to  $\gamma$ 
28:   end for
29:   Evaluation tables of  $\tilde{\sigma}_{[\mu]}(\langle i \rangle, (\gamma, \mathbf{x})) \ \forall i \in [\mu] \leftarrow \text{COLLAPSE}(\tilde{\sigma}_{[\mu]}, \gamma, \log \ell - 1)$ 
30:   for  $k \leftarrow \log \ell, \dots, \mu + \log \ell$  do
31:      $u_k(X) \leftarrow 0$ 
32:     for  $\mathbf{x}' \in \{0, 1\}^{\mu+\log \ell-k}$  do
33:        $u_{\mathbf{x}'}(X) \leftarrow \text{eq}((\gamma, X, \mathbf{x}'), \beta') p(\gamma, X, \mathbf{x}')$  ▷ Using FFT
34:        $u_k(X) \leftarrow u_k(X) + u_{\mathbf{x}'}(X)$ 
35:     end for
36:     Send  $[[u_k]]$  to V
37:     Receive  $\gamma_k$  from V and append it to  $\gamma$ 
38:     Collapse the evaluation table of  $\tilde{\sigma}_{[\mu]}$ 
39:   end for
40: end procedure

```

$\ell \cdot 2^{\mu+\log \ell-k}$. Therefore the total number of field additions for computing the prover messages is

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

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 $n \cdot \tilde{O}(\mu/\ell) + \ell 2^\ell$.

Verifier Query Complexity Verifier batch 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

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

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).

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

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

where $f, g \in \mathbb{F}_{\mu}^{(\leq 1)}$, such that there exists a permutation $\sigma : \{0, 1\}^{\mu} \rightarrow \{0, 1\}^{\mu}$ and $f(\mathbf{y}) = g(\sigma(\mathbf{y}))$ for all $\mathbf{y} \in \{0, 1\}^{\mu}$.

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 $\sigma^{-1} : \{0, 1\}^{\mu} \mapsto \{0, 1\}^{\mu}$ such that $\sigma^{-1}(\sigma(\mathbf{y})) = \mathbf{y}$ over $\{0, 1\}^{\mu}$.

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

```

1: procedure PERM2 VERIFIER( $[[f]]$ ,  $[[g]]$ )
2:   Receive  $[[\tilde{\sigma}_{[\mu]}]]$ ,  $[[\tilde{\tau}_{[\mu]}]]$  from  $\mathsf{P}$ 
3:   Randomly select  $r \xleftarrow{\$} \mathbb{F}$  and send  $r$  to  $\mathsf{P}$ 
4:   Define  $[[f']]$ ,  $[[g']]$  such that  $f'(\mathbf{y}) = \mathbf{y} + r \cdot f(\mathbf{y})$  and  $g'(\mathbf{y}) = \tilde{\tau}(\mathbf{y}) + r \cdot g(\mathbf{y})$  for
   every  $\mathbf{y} \in \{0, 1\}^\mu$ 
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  $f(\mathbf{y}) = g(\sigma(\mathbf{y}))$  for every  $\mathbf{y} \in \{0, 1\}^\mu$ . 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 oracles  $[[\tilde{\sigma}_{[\mu]}]]$ ,  $[[\tilde{\tau}_{[\mu]}]]$  to  $\mathsf{V}$ 
12:  Receive random challenge  $r \in \mathbb{F}$  from  $\mathsf{V}$ 
13:  Define  $f'$  such that  $f'(\mathbf{y}) = \mathbf{y} + r \cdot f(\mathbf{y})$ 
14:  MULPERM PROVER( $\tilde{\sigma}_{[\mu]}, f'$ )
15:  BINMAP PROVER( $\tilde{\sigma}_{[\mu]}$ )
16: end procedure

```

Theorem 7. *The triple-sumcheck PIOP described by the prover and verifier in Algorithm 8 for proving $([[f]], [[g]]; f, g) \in \mathcal{R}_{\text{PERM2}}$ has perfectly completeness, soundness error $O(\text{polylog } n/|\mathbb{F}|)$, and the following properties:*

# Rounds	$O(\log n)$
# Oracles, Length, Size ($\times$ and runtime)	$2, 2n, \text{Length} \cdot \log \mathbb{F} $
Prover Work	$n \cdot \tilde{O}(\sqrt{\log n}) \mathbb{F}$
Verifier Work	$O(\log n) \mathbb{F}$
Verifier Oracle Queries	$O(\log n)$ queries and 1 batch query

8.1 Proving σ is a permutation

Lemma 7. *Given a function σ with pre-image space $\{0, 1\}^\mu$, $\tau(\sigma(\mathbf{y})) = \mathbf{y}$ for every $\mathbf{y} \in \{0, 1\}^\mu$ and the image space of σ is $\{0, 1\}^\mu$ if and only if $\sigma : \{0, 1\}^\mu \rightarrow \{0, 1\}^\mu$ is a permutation and $\tau = \sigma^{-1}$.*

Proof. The forward direction is trivial. We prove the backward direction below. Since $\tau(\sigma(\mathbf{y})) = \mathbf{y}$ for every $\mathbf{y} \in \{0, 1\}^\mu$, it must be that $\sigma : \{0, 1\}^\mu \mapsto \mathbb{H}$ and $\tau : \mathbb{H} \mapsto \{0, 1\}^\mu$ for some space $\mathbb{H}$. Since the image space of σ is $\{0, 1\}^\mu$, $\mathbb{H} = \{0, 1\}^\mu$ and $\sigma : \{0, 1\}^\mu \mapsto \{0, 1\}^\mu$. Since the pre-image space and the image space of σ is the same, it must be a permutation, and $\tau = \sigma^{-1}$.

To prove σ is a permutation, prover provides an oracle to $\tilde{\tau}_{[\mu]}$ which is supposed to be $\tilde{\sigma}_{[\mu]}^{-1}$, and proves that a) $\tau(\sigma(\mathbf{y})) = \mathbf{y}$ for every $\mathbf{y} \in \{0, 1\}^\mu$ and b) σ

maps to binaries over $\{0, 1\}^\mu$.

The first task can be reduced to the following sumcheck through similar process as described in Section 5,

$$\sum_{\mathbf{x} \in \{0, 1\}^\mu} \mathbf{x} \cdot \tilde{\mathbb{I}}_\sigma(\mathbf{x}, \boldsymbol{\alpha}) = \tilde{\tau}(\boldsymbol{\alpha}) \quad (14)$$

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

$$\sum_{\mathbf{x} \in \{0, 1\}^\mu, i \in [\mu]} \text{eq}((\mathbf{x}, \langle i \rangle), \mathbf{s}) \cdot \underbrace{\tilde{\sigma}_{[\mu]}(i, \mathbf{x})(1 - \tilde{\sigma}_{[\mu]}(i, \mathbf{x}))}_{h(\mathbf{x}, i)} = 0 \quad (15)$$

Now we prove the completeness and soundness parts of Theorem 7.

Proof.

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

Soundness We know from Theorem 6 that using **MulPerm** incurs soundness error $O(\text{polylog}(n)/|\mathbb{F}|)$. The sumcheck for Equation 15 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 $O(\text{polylog}(n)/|\mathbb{F}|)$.

Remark 6. Using a random linear permutation, proving $\tau(\sigma(\mathbf{y})) = \mathbf{y}$ for every $\mathbf{y} \in \{0, 1\}^\mu$ can be batched using with the usual permutation check which proves that $f(\mathbf{y}) = g(\sigma(\mathbf{y}))$ for every $\mathbf{y} \in \{0, 1\}^\mu$. We refer to [CBBZ23] for the proof of this technique. Specifically, define

$$f'(\mathbf{y}) = \mathbf{y} + R \cdot f(\mathbf{y}), \quad g'(\mathbf{y}) = \tau(\mathbf{y}) + R \cdot g(\mathbf{y}) \quad \forall \mathbf{y} \in \{0, 1\}^\mu$$

and prove $f'(\mathbf{y}) = g'(\sigma(\mathbf{y}))$ for every $\mathbf{y} \in \{0, 1\}^\mu$. In Algorithm 8, we describe this optimized algorithm.

8.2 Improving the prover cost when proving σ maps to binaries

If the sumcheck for Equation 15 is computed naively, 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.

Prover wants to prove

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

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

$$\sum_{\mathbf{x}' \in \{0, 1\}^{\mu'}} \text{eq}(\mathbf{x}', \mathbf{s}) \cdot h(\mathbf{x}') = 0 \quad (16)$$

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

$$\sum_{\mathbf{x}' \in \{0,1\}^{\mu'-k}} \text{eq}((\alpha, X, \mathbf{x}'), \mathbf{s}) h(\alpha, X, \mathbf{x}')$$

Computing the k -th round directly takes $2^{\mu'-k}$ field additions and $8 \cdot 2^{\mu'-k}$ field multiplications.⁹

Prover buckets in their head Consider what happens if we fold b variants into h , then for $\mathbf{x}' \in \{0,1\}^{\mu'-1}$ we have

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

for $\mathbf{x}' \in \{0,1\}^{\mu'-b}$, $\mathbf{s}'' = \mathbf{s}_{[-b]}$ we have

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

Replace Equation 16 with $h^{(b)}$:

$$\sum_{\mathbf{x}' \in \{0,1\}^{\mu'-b}} \text{eq}(\mathbf{x}', \mathbf{s}_{[:\mu'-b]}) h^{(b)}(\mathbf{x}', \mathbf{s}_{[:b]})$$

The k -th round then becomes

$$\sum_{\mathbf{x}' \in \{0,1\}^{\mu'-b-k}} \text{eq}((\alpha, X, \mathbf{x}'), \mathbf{s}_{[:\mu'-b]}) h^{(b)}((\alpha, X, \mathbf{x}'), \mathbf{s}_{[:b]})$$

As discussed in Section 7.3, the univariate $\tilde{\sigma}_{[\mu]}$ polynomial has 2^{2^k} possible identities in the k th 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 8. *Let $n = 2^\mu$. The sumcheck PIOP for Equation 15 can be computed using $n \cdot o(\log \log n)$ field additions and $o(n)$ field multiplications. The proof size is $O(\log(n + \log n))$, 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 $8(n\mu)^{1/4} \cdot \log \mu' =$

⁹ The factor 8 comes from the fact that the prover message is degree-3 in each entry of $\mathbf{x}$ 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.

$8(n\mu)^{1/4} \cdot \log(\mu + \log \mu) = o(n)$ 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 7. For sufficiently large n , the prover cost of running **MulPerm** dominates in proving prover-provided permutation.

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 $\tilde{\mathbb{I}}_L, \tilde{\mathbb{I}}_R$, 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.

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

```

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

```

9 Generalization to Lookups

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

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

where $\tilde{\rho} \in \mathbb{F}_\mu^{(\leq 1)}$ is the multilinear extension of map $\rho \in \text{FUN}(\{0, 1\}^\mu, \{0, 1\}^\kappa)$, $f \in \mathbb{F}_\kappa^{(\leq 1)}$, $g \in \mathbb{F}_\mu^{(\leq 1)}$, and

$$f(\rho(\mathbf{x})) = g(\mathbf{x}) \quad \forall \mathbf{x} \in \{0, 1\}^\mu \quad (17)$$

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 2^κ -sized lookup table, and g is the multilinear polynomial representing the 2^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 the definition of permutation mapping σ , we define the multilinear version of ρ as

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

with $\tilde{\rho}_i(\mathbf{x}) \in \{0, 1\} \quad \forall \mathbf{x} \in \{0, 1\}^\mu \quad \forall i \in [\kappa]$. 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 $\tilde{\rho}_{[\kappa]}(\langle i \rangle, \mathbf{X}) = \tilde{\rho}_i(\mathbf{X})$ 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 $\mathbb{1}_\rho : \{0, 1\}^\mu \times \{0, 1\}^\kappa \mapsto B$,

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

Following the same steps we took when reducing the permutation check to sum-check 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 $\mathbf{y}$:

$$\sum_{\mathbf{y} \in \{0, 1\}^\kappa} f(\mathbf{y}) \tilde{\mathbb{1}}_\rho(\mathbf{x}, \mathbf{y}) = g(\mathbf{x}) \quad \forall \mathbf{x} \in \{0, 1\}^\mu. \quad (1)$$

Unfortunately, we cannot apply Schwartz-Zippel this time. As we previously noted, arithmetizations of $\tilde{\mathbb{1}}_\rho$ that is multilinear in $\mathbf{x}$ 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 $\mathbf{s} \in \mathbb{F}^\mu$ are random challenges:

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \sum_{\mathbf{y} \in \{0,1\}^\kappa} f(\mathbf{y}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) g(\mathbf{x})$$

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

$$\sum_{\mathbf{y} \in \{0,1\}^\kappa} f(\mathbf{y}) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) g(\mathbf{x})$$

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

$$\sum_{\mathbf{y} \in \{0,1\}^\kappa} f(\mathbf{y}) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \mathbf{y}) = S \quad (18)$$

and another one for proving $\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) g(\mathbf{x}) = S$. The second sumcheck is easy to perform and will hence be omitted in the following discussion. We will focus on the sumcheck for Equation 18.

9.1 Sumcheck PIOP for lookups

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

Theorem 8. *Let $n = 2^\mu, T = 2^\kappa$ such that $T < 2^{(1-\epsilon) \cdot \mu^2}$ for a constant $\epsilon > 0$. Given $f \in \mathbb{F}_\kappa^{(\leq 1)}, g \in \mathbb{F}_\mu^{(\leq 1)}$ 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 $(\tilde{\rho}_{[\kappa]}, [[\tilde{\rho}_{[\kappa]}]], [[f]], [[g]]; f, g) \in \mathcal{R}_{\text{LKUP}}$ has perfect completeness, soundness error $\frac{\text{polylog}(n+T)}{|\mathbb{F}|}$, and the following properties:*

# Rounds	$O(\log T + \log n)$
# Oracles, Length, Size (i)	1, $n \log T$, $n \log T$
# Oracles, Length, Size (x and runtime)	$O(\log T + \log n)$, $O(T + n)$, $\text{Length} \cdot \log \mathbb{F} $
Prover Work, if $T \leq n$	$n \cdot \tilde{O}(\sqrt{\log T}) \mathbb{F}$ and one evaluation of f
Prover Work, if $T > n$	$O(n(\log T - \log n)) \mathbb{F}$ and one evaluation of f
Verifier Work	$O(\log T + \log n) \mathbb{F}$
Verifier Oracle Queries	$O(\log T + \log n)$ queries and 1 batch query

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

$$\sum_{\mathbf{y} \in \{0,1\}^{\kappa-k}} f(\alpha, Y, \mathbf{y}) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, (\alpha, Y, \mathbf{y})) = S$$

Observe that we do not actually need to compute the terms for every $\mathbf{y}$ and $\mathbf{x}$, but only those $\mathbf{y}$ that is in the image space of ρ and those $\mathbf{x}$ that map to these corresponding $\mathbf{y}$. Thus, we introduce the following prover-side optimization. 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

$$\sum_{\mathbf{y} \in \{I_\rho\}_{[k:]}} f(\boldsymbol{\alpha}, Y, \mathbf{y}) \sum_{\mathbf{x} \in \{0,1\}^\mu | \rho(\mathbf{x})_{[k:]} = \mathbf{y}} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, (\boldsymbol{\alpha}, Y, \mathbf{y}))$$

Moreover, note that the evaluation table of $\tilde{\mathbb{I}}_\rho(\mathbf{x}, \mathbf{Y})$ 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 $\tilde{\mathbb{I}}_\rho$.

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 $\frac{n}{2} + \frac{n}{4} + \dots + 1 = O(n)$.

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 8. 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

$$f(\boldsymbol{\alpha}) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \cdot \tilde{\mathbb{I}}_\rho(\mathbf{x}, \boldsymbol{\alpha}) = S'$$

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 MulPerm to prove

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \cdot \tilde{\mathbb{I}}_\rho(\mathbf{x}, \boldsymbol{\alpha}) = S' / f(\boldsymbol{\alpha}) \quad (19)$$

Corollary 1. *Let $n = 2^\mu, T = 2^\kappa$ such that $T < 2^{(1-\epsilon) \cdot \mu^2}$ for a constant $\epsilon > 0$. Given $f, g \in \mathbb{F}_\mu^{(\leq 1)}$, and preprocessed $\tilde{\rho}_{[\kappa]} : \{0, 1\}^{\log \kappa + \mu} \mapsto \{0, 1\}$, using MulPerm for proving Equation 19 gives a PIOP with perfect completeness, soundness error $\frac{\text{polylog}(n+T)}{|\mathbb{F}|}$, and the following properties,*

# Rounds	$O(\log n)$
# Oracles, Length, Size ($\mathbb{x}$ and runtime)	$O(\log n), O(\log n \sqrt{\log T}), \text{Length} \cdot \log \mathbb{F} $
Prover Work	$n \cdot \tilde{O}(\sqrt{\log T}) \mathbb{F}$ and $(\log T - \log n)2^{\log n}$ evaluations of f
Verifier Work	$O(\log n) \mathbb{F}$
Verifier Oracle Queries	$O(\log n)$ queries and 1 batch query

Proof. Let p be any multilinear polynomial, $\rho \in \text{FUN}(\{0,1\}^\mu, \{0,1\}^\kappa)$ is some arbitrary map, and $\tilde{\mathbb{I}}_\rho(\mathbf{X}, \mathbf{Y})$ is the arithmetization of indicator function for ρ that is multilinear in $\mathbf{Y}$. Since *no where* in the 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

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} p(\mathbf{x}) \cdot \tilde{\mathbb{I}}_\rho(\mathbf{x}, \boldsymbol{\alpha}) = S$$

Thus, it can be directly used to prove Equation 19.

Using MulPerm, we arithmetize $\tilde{\mathbb{I}}_\rho(\mathbf{X}, \mathbf{Y})$ as the product of ℓ sub-indicator functions, each of which depend on κ/ℓ bits of $\mathbf{Y}$. 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 MulPerm has an overall $n \cdot \tilde{O}(\sqrt{\log T})$ number of field operations. With this parameterization the soundness error is

$$\begin{aligned} & \left(\underbrace{(\sqrt{\kappa} + 1)(\mu + \log \sqrt{\kappa})}_{\text{First Sumcheck}} + \underbrace{(\kappa/\sqrt{\kappa} + 1)(\mu + \log \sqrt{\kappa})}_{\text{Second Sumcheck}} \right) / |\mathbb{F}| \\ &= \frac{O(\mu\sqrt{\kappa})}{|\mathbb{F}|} = \frac{\text{polylog}(n + T)}{|\mathbb{F}|} \end{aligned}$$

9.2 Prover-provided lookup reduced to sumcheck

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

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

where $f \in \mathbb{F}_\kappa^{(\leq 1)}, g \in \mathbb{F}_\mu^{(\leq 1)}$, such that there exists a mapping $\rho \in \text{FUN}(\{0,1\}^\mu, \{0,1\}^\kappa)$ and

$$f(\rho(\mathbf{x})) = g(\mathbf{x}) \quad \forall \mathbf{x} \in \{0,1\}^\mu \quad (20)$$

Let $n := 2^\mu, T = 2^\kappa$ and $\mu' = \log \kappa + \mu$. To prove $([f], [g]; f, g)$ for $f \in \mathbb{F}_\kappa^{(\leq 1)}, g \in \mathbb{F}_\mu^{(\leq 1)}$ using MulPerm, the prover will need to first compute $\tilde{\rho}_{[\kappa]}$ and

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

- 1: **procedure** LOOKUP VERIFIER($[[f]], [[g]]$)
- 2: Receive $[[\tilde{\rho}_{[\kappa]}]]$ from P
- 3: Randomly select $\mathbf{s} \xleftarrow{\$} \mathbb{F}^\mu$ and send it to P
- 4: Receive claimed sum S from P
- 5: SUMCHECK VERIFIER to check $\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s})g(\mathbf{x}) = S$
- 6: SUMCHECK VERIFIER through the outer sum for proving Equation 18

$$\sum_{\mathbf{y} \in \{0,1\}^\kappa} f(\mathbf{y}) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \mathbf{y}) = S \quad (18)$$

Let the challenges be $\alpha \in \mathbb{F}^\kappa$, and the claim at the end be

$$f(\alpha) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \alpha) = S'$$

- 7: Query $[[f]]$ at α to get evaluation S_f . Compute $S \leftarrow S'/S_f$
- 8: Adapt MulPerm to check

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \alpha) = S$$

9: **end procedure**

- 10:
- 11: **procedure** LOOKUP PROVER(f, g)
- 12: Compute ρ such that $f(\rho(\mathbf{y})) = g(\mathbf{y})$ for every $\mathbf{y} \in \{0,1\}^\mu$.
- 13: Interpolate the MLE $\tilde{\rho}_{[\kappa]}$ and send oracle $[[\tilde{\rho}_{[\kappa]}]]$ to V
- 14: Receive random challenges $\mathbf{s} \in \mathbb{F}^\mu$ from V
- 15: Compute $S \leftarrow \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s})g(\mathbf{x})$. Send S to V
- 16: SUMCHECK PROVER for proving $\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s})g(\mathbf{x}) = S$
- 17: SUMCHECK PROVER through the outer sum for proving Equation 18, but in the following form, where I_ρ is the image space of ρ .

$$\sum_{\mathbf{y} \in I_\rho} f(\mathbf{y}) \sum_{\mathbf{x} \in \{0,1\}^\mu | \rho(\mathbf{x}) = \mathbf{y}} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \mathbf{y}) = S$$

Let the challenges be $\alpha \in \mathbb{F}^\kappa$, and the claim at the end be

$$f(\alpha) \sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \alpha) = S'$$

- 18: Adapt MulPerm to prove

$$\sum_{\mathbf{x} \in \{0,1\}^\mu} \text{eq}(\mathbf{x}, \mathbf{s}) \tilde{\mathbb{I}}_\rho(\mathbf{x}, \alpha) = S'/f(\alpha)$$

19: **end procedure**

send an oracle of it to the verifier. Moreover, since $\mathcal{R}_{\text{LKUP2}}$ is unindexed, we need to prove $\tilde{\rho}_{[\kappa]}$ maps to $\{0, 1\}$ over $\{0, 1\}^{\mu'}$, 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 $\tilde{\mathbb{I}}_L, \tilde{\mathbb{I}}_R$; **MulPerm** incurs a slightly larger but close to linear prover cost and can be compiled using any PCS.

Both of our permutation arguments can be generalized to support prover-provided permutations and lookups. 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.
- [BBPWM18] 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>.
- [BCG20] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. “Linear-Time Arguments with Sublinear Verification from Tensor Codes”. In: *TCC 2020, Part II*. Ed. by Rafael Pass and Krzysztof

- Pietrzak. Vol. 12551. LNCS. Springer, Cham, Nov. 2020, pp. 19–46. DOI: 10.1007/978-3-030-64378-2_2.
- [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.
- [BCGL22] Jonathan Bootle, Alessandro Chiesa, Ziyi Guan, and Siqi Liu. *Linear-Time Probabilistic Proofs with Sublinear Verification for Algebraic Automata Over Every Field*. Cryptology ePrint Archive, Report 2022/1056. 2022. URL: <https://eprint.iacr.org/2022/1056>.
- [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 Carmi 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 zk-SNARKs 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.
- [CTHK25] Bing-Jyue Chen, Lilia Tang, David Heath, and Daniel Kang. *Linear-time and Logarithmically-sound Permutation and Multiset SNARKs*. Cryptology ePrint Archive, Paper 2025/1967. 2025. URL: <https://eprint.iacr.org/2025/1967>.
- [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.
- [GLSTW23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. “Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS”. In: *CRYPTO 2023, Part II*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14082. LNCS. Springer, Cham, Aug. 2023, pp. 193–226. DOI: 10.1007/978-3-031-38545-2_7.
- [GW20] Ariel Gabizon and Zachary J. Williamson. *plookup: A simplified polynomial protocol for lookup tables*. Cryptology ePrint Archive, Paper 2020/315. 2020. URL: <https://eprint.iacr.org/2020/315>.
- [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>.
- [HR22] Justin Holmgren and Ron D. Rothblum. “Faster Sounder Succinct Arguments and IOPs”. In: *CRYPTO 2022, Part I*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13507. LNCS. Springer, Cham, Aug. 2022, pp. 474–503. DOI: 10.1007/978-3-031-15802-5_17.
- [Kad25] George Kadianakis. *Shipping an L1 zkEVM #2: The Security Foundations*. 2025. URL: <https://blog.ethereum.org/2025/12/18/zkevm-security-foundations>.
- [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.

- [RR21] Noga Ron-Zewi and Ron D. Rothblum. *Proving as Fast as Computing: Succinct Arguments with Constant Prover Overhead*. Cryptology ePrint Archive, Report 2021/1673. 2021. URL: <https://eprint.iacr.org/2021/1673>.
- [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>.
- [Sta21] StarkWare. *ethSTARK Documentation*. Cryptology ePrint Archive, Report 2021/582. 2021. URL: <https://eprint.iacr.org/2021/582>.
- [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 Ar-

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

```
