Lasso: Unlocking the lookup singularity

Srinath Setty[*] Justin Thaler[†] Riad Wahby[‡]

Abstract

This paper describes two new lookup arguments called Lasso and SuperLasso. When the table is structured, the prover does not have to explicitly materialize its entries, only "paying" in runtime for table entries that are actually accessed by lookup operations. This applies to tables commonly used to implement range checks, bitwise operations, "nonnative" big-number arithmetic, and even transitions of a full-fledged CPU such as RISC-V. This enables the use of much larger tables than prior works (say, of size 2^{128} or larger).

Specifically, for m lookups into a table of size n, and for any integer parameter $c > 0$, the dominant cost for Lasso's prover is cryptographically committing to $3 \cdot c \cdot m + c \cdot n^{1/c}$ field elements. Moreover, $2 \cdot c \cdot m + c \cdot n^{1/c}$ of those field elements are from the set $\{0, \ldots, \max\{m, n^{1/c}\} - 1\}$. When using multiexponentiation-based commitment schemes, this results in prover costs dominated by just c multiexponentiations of size m, where c is chosen to ensure that $n^{1/c} \leq m$. For SuperLasso, even the remaining cm committed field elements are in $\{0, 1, \ldots, n^{1/c}\}$ for tables storing small values, resulting in a prover time dominated by only $O(c(m + n^{1/c}))$ group *operations*.

For unstructured tables, Lasso's and SuperLasso's provers additionally incur only $O(\sqrt{n})$ cryptographic operations and $O(n)$ finite field operations. This offers the first transparent lookup argument for arbitrary tables with sublinear-in-table-size cryptographic work for the prover.

Lasso and SuperLasso achieve state-of-the-art proving time for small (unstructured) tables in addition to large ones. For example, by setting $c = 1$, SuperLasso's prover commits to roughly $3m + n$ field elements, all of which (up to low-order terms) are in $\{0, 1, \ldots, \max\{m, n, q\}\}$, where q is the largest value in the table. If m, n, and q are much smaller than the field size, this substantially speeds up commitment.

We achieve these results by leveraging three new tools. (1) A stronger security analysis for *Spark*, Spartan's *optimal* commitment scheme for *sparse* polynomials (CRYPTO 2020). Spartan's security analysis assumed that certain metadata associated with a sparse polynomial is committed by an honest party. This is acceptable for its purpose in Spartan, but not for Lasso where the sparse polynomial is committed by the prover. We prove that Spark remains secure even when that metadata is committed by a malicious party. This provides the first "standard" commitment scheme for sparse multilinear polynomials with optimal prover costs. This result is of independent interest.

(2) A new linear-time sum-check protocol that we call *sparse-dense sum-check* protocol. This protocol computes the inner product between two vectors of length n. As long as one of the two vectors has at most $m \geq n^{1/c}$ non-zero entries and the other is suitably structured, our protocol incurs only $O(c \cdot m)$ field work for the prover.

(3) Surge, a generalization of Spark that, for many tables, eliminates Lasso's need for the sparse-dense sum-check protocol, and ensures that *all* committed field elements are in $\{0, 1, \ldots, \max\{m, n^{1/c}\} - 1\}$. Surge underlies SuperLasso.

Finally, we describe a new polynomial commitment scheme, called Sona, whose cost profile is especially suitable for use with Lasso. Sona composes a simplified version of Hyrax's polynomial commitment scheme (S&P 2018) with Nova (CRYPTO 2022) to provide a constant-sized polynomial commitments and evaluation proofs *without* trusted setup. In particular, a commitment is a single hash value, computing the commitment requires only a linear-sized multiexponentiation, computing evaluation proofs requires *sub-linear* cryptographic work, and the verification time for an evaluation proof is a constant-sized multiexponentiation.

[*] Microsoft Research
[†] a16 crypto research and Georgetown University
[‡] Carnegie Mellon University

Contents

1 Introduction

Suppose that an untrusted prover $\mathcal{P}$ claims to know a witness w satisfying some property. For example, w might be a pre-image of a designated value y of a cryptographic hash function h, i.e., a w such that $h(w) = y$. A trivial proof is for $\mathcal{P}$ to send w to the verifier $\mathcal{V}$, who checks that w satisfies the claimed property.

A zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) achieves the same, but with better verification costs (and proof sizes) and privacy properties. Succinct means that verifying a proof is much faster than checking the witness directly (this also implies that proofs are much smaller than the size of the statement proven). Zero-knowledge means that the verifier does not learn anything about the witness beyond the validity of the statement proven.

Fast algorithms via lookup tables. A common technique in the design of fast algorithms is to use *lookup tables*. These are pre-computed tables of values that, once computed, enable certain operations to be computed quickly. For example, in *tabulation-based universal hashing* [PT12, PT13], the hashing algorithm is specified via some small number c of tables $T_1, \ldots, T_c$, each of size $n^{1/c}$. Each cell of each table is filled with a random q-bit number in a preprocessing step. To hash a key x of length n, the key is split into c "chunks" $x_1, \ldots, x_c \in \{0,1\}^{n/c}$, and the hash value is defined to be the bitwise XOR of c *table lookups* i.e., $\oplus_{i=1}^{c} T_i[x_i]$.

Lookup tables are also useful in the context of SNARKs. Recall that to apply SNARKs to prove the correct execution of computer programs, one must express the execution of the program in a specific form that is amenable to probabilistic checking (e.g., as arithmetic circuits or generalizations thereof). Lookup tables can facilitate the use of substantially smaller circuits.

For example, imagine that a prover wishes to establish that at no point in a program's execution did any integer ever exceed 2^{128}, say, because were that to happen then an uncorrected "overflow error" would occur. A naive approach to accomplish this inside a circuit-satisfiability instance is to have the circuit take as part of its "non-deterministic advice inputs" 128 field elements for each number x arising during the execution. If the prover is honest, these 128 advice elements will be set to the binary representation of x. The circuit must check that all of the 128 advice elements are in $\{0,1\}$ and that they indeed equal the binary representation of x, i.e., $x = \sum_{i=0}^{127} 2^i \cdot b_i$, where $b_0, \ldots, b_{127}$ denotes the advice elements. This is very expensive: a simple overflow check turns into at least 129 constraints and an additional 128 field elements in the prover's witness that must be cryptographically committed by the prover.[1]

Lookup tables offer a better approach. Imagine for a moment that the prover and the verifier initialize a lookup table containing all integers between 0 and $2^{128} - 1$. Then the overflow check above amounts to simply confirming that x is in the table, i.e., the overflow check *is* a single table lookup. Of course, a table of size 2^{128} is far too large to be explicitly represented—even by the prover. This work describes techniques to enable such a table lookup without requiring a table such as this to ever be explicitly materialized, by either the prover or the verifier.

Table lookups are now used pervasively in deployed applications that employ SNARKs. They are very useful for representing "non-arithmetic" operations efficiently inside circuits [BCG+18, GW20b, GW20a]. The above example is often called a *range check* for the range $\{0, 1, \ldots, 2^{128} - 1\}$. Other example operations for which lookups are useful include bitwise operations such as XOR and AND [BCG+18], and any operations that require big-number arithmetic.

Lookup arguments. To formalize the above discussion regarding the utility of lookup tables in SNARKs, a (non-interactive) *lookup argument* is a SNARK for the following claim made by the prover.

Definition 1.1 (Statement proven in a lookup argument). *Given a commitment* cm_a *and a public set* T *of* N *field elements, represented as vector* $t = (t_0, \ldots, t_{N-1}) \in \mathbb{F}^N$ *to which the verifier has (possibly) been provided*

[1] As we explain later (Section 1.1.3), for certain commitment schemes, the prover's cost to commit to vectors consisting of many $\{0,1\}$ values can be much cheaper than if the to vectors contain arbitrary field elements. However, other SNARK prover costs (e.g., number of field operations) will grow linearly with the number of advice elements and constraints in the circuit to which the SNARK is applied, irrespective of whether the advice elements are $\{0,1\}$-valued.

a commitment cm_t*, the prover knows an opening* $a = (a_0, \ldots, a_{m-1}) \in \mathbb{F}^m$ *of* cm_a *such that all elements of* a *are in* T*. That is, for each* $i = 0, \ldots, m-1$*, there is a* $j \in \{0, \ldots, N-1\}$ *such that* $a_i = t_j$*.*

The set T in Definition 1.1 is the contents of a lookup table and the vector a is the sequence of "lookups" into the table. The prover in the lookup argument proves to the verifier that every element of a is in T.

Remark 1. *Definition 1.1 is a standard formulation of lookup arguments in SNARKs [ZGK*$^+$*22]. It treats the table as an unordered list of values—*T *is a* set *and, accordingly, reordering the vector* t *does not alter the validity of the prover's claim. However, a lookup argument satisfying Definition 1.1 can be transformed into one in which the order of* t *matters, at least as long as all the table entries are not "too close" to the field characteristic. See Section 4.4 for details.*

A recent flurry of works (Caulk [ZBK$^+$22], Caulk+ [PK22], flookup [GK22], Baloo [ZGK$^+$22], and cq [EFG22]) have sought to give lookup arguments in which the prover's runtime is sublinear in the table size N. This is important in applications where the lookup table itself is much larger than the number of lookups into that table. As a simple and concrete example, if the verifier wishes to confirm that $a_0, \ldots, a_{m-1}$ are all in a large range (say, in $\{0, 1, \ldots, 2^{32} - 1\}$), then performing a number of cryptographic operations linear in N will be slow or possibly infeasible. For performance reasons, these papers also express a desire for the commitment scheme used to commit to a and t to be additively homomorphic. However, these prior works all require generating a structured reference string of size N as well as additional pre-processing involving $O(N \log N)$ group exponentiations. This limits the size of the tables to which they can be applied. For example, the largest structured reference strings generated today are many gigabytes in size and still only support $N < 2^{30}$.

1.1 A new lookup argument

We describe two new lookup arguments, Lasso and SuperLasso (Lasso is short for LASSO-of-Truth: Lookup Arguments via Sum-check and Sparse-polynomial-commitments, including for Oversized Tables). Lasso is based on two technical components: (1) the *sum-check protocol* [LFKN90] and (2) a commitment scheme for sparse multilinear polynomials (which is itself based on the sum-check protocol). SuperLasso replaces both (1) and (2) with a new generalization of sparse polynomial commitments. All of these components require new results of independent interest.

1.1.1 First component: optimal commitment scheme for sparse polynomials (Spark)

We seek a way for an untrusted prover to cryptographically commit to a multilinear polynomial p and later reveal a requested evaluation $p(r)$ of p along with a proof that the provided value is indeed equal to the committed polynomial's evaluation at r. Crucially, Lasso requires that the the prover's runtime depends only on the sparsity of the polynomial.[2] For the latter, Spartan [Set20] provides such a commitment scheme (which it calls *Spark*), but it assumed that certain metadata associated with the sparse polynomial is committed honestly, which was sufficient for its purposes. A naive extension of the scheme to handle a malicious committer incurs concrete and asymptotic overheads, which is undesirable. Nevertheless, we prove that Spartan's scheme in fact satisfies a stronger security property without any modifications (i.e., it is secure even if the committer is malicious). This provides the first "standard" sparse polynomial commitment scheme with optimal prover costs, a result of independent interest. Furthermore, we specialize the sparse polynomial commitment scheme in the setting of Lasso to obtain concrete efficiency benefits.

1.1.2 Second component: the *sparse-dense* sum-check protocol

Prior work on the *linear-time sum-check protocol* shows how one can force a untrusted prover $\mathcal{P}$ to truthfully compute the inner product of two vectors $u, t \in \mathbb{F}^N$ with $O(N)$ field operations [CTY11, Tha13]. However, Lasso forces the prover $\mathcal{P}$ to honestly compute an inner product between two vectors, say, u and t, each of

[2] For multilinear polynomials, m-sparse refers to polynomials $p\colon \mathbb{F}^\ell \to \mathbb{F}$ in ℓ variables such that $p(x) \neq 0$ for at most m values of $x \in \{0,1\}^\ell$. In other words, p has at most m non-zero coefficients in the so-called multilinear Lagrange polynomial basis. There are $n \coloneqq 2^\ell$ Lagrange basis polynomials, so if $m \ll 2^\ell$, then only a tiny fraction of the possible coefficients are non-zero. In contrast, if $m = \Theta(2^\ell)$, then we refer to p as a *dense* polynomial.

length N, where N may be very large. Specifically, N is the size of the lookup table to which Lasso is applied, and t is the vector that lists all of the table entries.

If N is large enough (say, $N \geq 2^{64}$ or $N \geq 2^{128}$), then $O(N)$ field operations is untenable. Fortunately, the vector u arising in Lasso is guaranteed to be *sparse*, in the sense that it has at most m entries that are non-zero, one for each of the m lookups into the lookup table t. Is it possible for $\mathcal{P}$ prove that $\langle u, t \rangle$ equals some claimed value, while performing a number of field operations proportional to m and not N?

Exploiting the table structure for a fast sum-check prover. We show that, indeed, this is the case, whenever the "dense" vector t is *structured* in a precise technical sense that is satisfied by many of the most important lookup tables used in practice (Section 4.3 provides details). Note that any result of this flavor inherently requires t to be structured in some way. Indeed, the sum-check protocol requires the prover to (at least) compute a random evaluation of a certain polynomial $\widetilde{t}$ derived from t, and this task alone takes time at least N for a general, unstructured length-N vector t. Also, the $O(m)$ field operations that we achieve for $\mathcal{P}$ is optimal. This is because, even if t is structured, simply computing $\langle u, t \rangle$ takes up to $O(m)$ time, since we can make no assumptions about u other than that it is m-sparse.

Exploiting table structure to avoid cryptographic commitment to t. At the end of the sum-check protocol, the verifier needs to obtain certain information about t. Specifically, t is interpreted as a $(\log N)$-variate[3] *polynomial* $\widetilde{t}$, and $\mathcal{V}$ must learn $\widetilde{t}(r)$ for a randomly chosen point $r \in \mathbb{F}^{\log N}$ that is chosen by $\mathcal{V}$ over the course of the sum-check protocol (here, $\widetilde{t}$ is the *multilinear extension polynomial* of the vector t, a notion we define in Section 2.1). If t is *not* structured, then computing $\widetilde{t}(r)$ would require $O(N)$ time for $\mathcal{V}$. An alternative approach, which we exploit when Lasso is applied to unstructured tables t, is for $\widetilde{t}$ to be cryptographically committed in a pre-processing phase, using a standard cryptographic primitive called a *polynomial commitment scheme*. This way, the untrusted prover can be forced to provide $\widetilde{t}(r)$ to $\mathcal{V}$ at the end of the sum-check protocol, along with a proof π that the provided value is consistent with the committed polynomial. Of course, this approach (especially computing and checking π) comes at substantial expense to the both the prover and verifier.

For structured tables, $\widetilde{t}(r)$ can instead be computed *by the verifier itself* in $O(\log N)$ time (in fact, this is essentially what we *mean* when we say that t is structured—see Section 3.3 for details).[4] This avoids the substantial performance costs of having $\widetilde{t}$ be cryptographically committed during pre-processing, and having the untrusted prover provide $\widetilde{t}(r)$ at the end of the sum-check protocol.

To summarize, our protocol for computing $\langle u, t \rangle$ when u is m-sparse directly applies the sum-check protocol of [LFKN90], but our implementation of an $O(m)$-time prover for structured vectors t contains fundamentally new ideas (see Section 3.2 for an overview).

1.1.3 Putting the first two components together: Lasso

In Lasso, the verifier is given as input a commitment to the multilinear extension polynomial $\widetilde{a}$ of the vector $a \in \mathbb{F}^m$ of lookups. $\mathcal{P}$ claims that for each entry a_i of a, there exists an index $j(i)$ such that

$$a_i = t_{j(i)}. \tag{1}$$

So the natural "witness" for $\mathcal{P}$'s claim is simply the list of indices $j(0), \ldots, j(m-1)$.

In order to check that Equation (66) holds for all $i = 0, 1, \ldots, m-1$, $\mathcal{P}$ will specify the witness in a convenient, "sparse" format. Specifically, let M be the $m \times N$ matrix whose i'th row specifies the index $j(i)$ in unary, i.e., the i'th row of M has a 1 in column $j(i)$ and zeros elsewhere. Note that if $m \ll N$, then M is highly sparse, and highly rectangular. M is a convenient format in which to specify the witness because checking that Equation (66) holds for all i is equivalent to checking that $Mt = a$. This simply involves a matrix-vector

[3] All logarithms in this paper are to base 2 unless otherwise noted.

[4] To ensure a sum-check prover that performs $O(cm)$ field operations rather than $O(m \log N)$ or $O(m \log^2 N)$, we require that t satisfy some additional properties beyond that $\widetilde{t}(r)$ can be evaluated in $O(\log N)$ time. Between this work and an upcoming companion paper called JOLT, we demonstrate that a wide variety of important lookup tables do satisfy these additional properties. See Theorem 2 for details.

multiplication (albeit one where the matrix M is very large and very sparse), and there are highly performative probabilistic techniques for verifying matrix-vector multiplications.[5]

Specifically, let $b := Mt$. Then (as explained in Section 4) the multilinear extension polynomial $\widetilde{b}$ of b can be expressed as

$$\widetilde{b}(r) = \sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r,j) \cdot \widetilde{t}(j), \tag{2}$$

where $\widetilde{M}$ and $\widetilde{t}$ are the multilinear extensions of M and t. Moreover, by the standard Schwartz-Zippel lemma, up to a negligible soundness error, checking that $Mt = a$ is equivalent to checking that $\widetilde{b}(r) = \widetilde{a}(r)$ where $r \in \mathbb{F}^{\log m}$ is a random vector chosen by $\mathcal{V}$.

Accordingly, Lasso proceeds as follows.

- $\mathcal{P}$ cryptographically commits to the polynomial $\widetilde{M}$ (using the sparse polynomial commitment scheme described in Section 1.1.1).[6]
- $\mathcal{V}$ picks a random vector $r \in \mathbb{F}^{\log M}$ and sends r to $\mathcal{P}$.
- Let u be the vector of length N (with entries indexed by vectors in $\{0,1\}^{\log N}$) whose j'th entry is $\widetilde{M}(r,j)$. Under this definition, the right hand side of Equation (2) is simply $\langle u, t \rangle$. And since M is m-sparse, it turns out that u is guaranteed to be m-sparse as well. So the prover and verifier can apply the sparse-dense sum-check protocol to compute $\langle u, t \rangle = \widetilde{b}(r)$. $\mathcal{V}$ confirms that $\widetilde{b}(r) = \widetilde{a}(r)$.

At the end of the sum-check protocol, $\mathcal{V}$ needs to evaluate $\widetilde{a}(r)$, $\widetilde{t}(r')$ and $\widetilde{M}(r,r')$ for a random point $(r,r') \in \mathbb{F}^{\log m} \times \mathbb{F}^{\log N}$. The evaluations of $\widetilde{a}$ and $\widetilde{M}$ are obtained from the prover via the evaluation procedure of the polynomial commitment schemes used to commit to them. If t is structured, then the verifier can compute $\widetilde{t}(r')$ on its own logarithmic time. If t is unstructured, $\widetilde{t}$ can be committed in pre-processing and $\widetilde{t}(r)$ provided by the prover, along with a proof that the claimed evaluation is correct.

Remark 2. *Lasso can be used with any SNARK, including those that prove R1CS or Plonkish satisfiability. This is particularly seamless for SNARKs that have the prover commit to the witness using a multilinear polynomial commitment scheme. This includes many known prover-efficient SNARKs [Set20, GLS+21, XZS22, CBBZ23]. If a SNARK does not natively use multilinear polynomial commitments (e.g., Marlin and Plonk [CHM+20, GWC19], which use univariate polynomial commitments), then one would need an auxiliary argument that the commitment cm_a used in Lasso is a commitment to the multilinear extension of the vector of all lookups performed in the SNARK.*

Lasso's costs for structured tables. For any desired integer $c \geq 1$, for structured tables, our prover applies the polynomial commitment scheme to $3 \cdot c$ polynomials over $\mathbb{F}$ of size m and c polynomials of size $N^{1/c}$, plus $O(m)$ operations in the field $\mathbb{F}$. This means that the dominant cost for $\mathcal{P}$ is cryptographically committing to $3 \cdot c \cdot m + c \cdot N^{1/c}$ field elements (namely the coefficients of the aforementioned polynomials over an appropriate basis).

On top of this, out of the $3 \cdot c \cdot m + c \cdot N^{1/c}$ committed field elements, $c \cdot m + cN^{1/c}$ of them are guaranteed to be in the set $\{0, \ldots, m-1\}$ and $c \cdot m$ of them are guaranteed to be in the set $\{0, \ldots, N^{1/c} - 1\}$. This has substantial implications for prover efficiency if using a commitment scheme based on multiexponentiations such as IPA/Bulletproofs [BCC+16, BBB+18], Dory [Lee21], Hyrax [WTS+18], or a new scheme called Sona that we introduce in this work (Section 1.2). Here, a multiexponentiation of size m in a multiplicative group $\mathbb{G}$ is an expression of the form $\prod_{i=1}^{m} g_i^{a_i}$ for group elements $g_1, \ldots, g_m$ and exponents $a_1, \ldots, a_m$.

Indeed, an under-appreciated fact about standard multiexponentiation algorithms such as Pippenger's bucketing algorithm (see [EHB22] for an excellent exposition) is that computing a size-m multiexponentiation

[5] The use of M as a "witness" in lookup arguments has appeared in many prior works [ZBK+22, PK22, ZGK+22, EFG22]. Lasso's technique to check that $Mt = a$ deviates significantly from these prior works.

[6] In Lasso, the commitment scheme used to commit to M *forces* each row of M to be a unit vector. Alternatively, one could use probabilistic proof machinery to *check* that each row of the committed matrix M has exactly one entry equal to 1, and all other entries are zero. However, this alternate approach increases costs by a constant factor.

involving exponents in $\{0, \ldots, m-1\}$ requires only $O(m)$ group operations. This is in contrast to the case of arbitrary exponents, for which the runtime of Pippenger's algorithm is $\Theta(m \cdot \lambda/\log(\lambda n))$, where $\lambda \gg \log(n)$ is the logarithm of the size of the cryptographic group. This is a superlinear in m number of group operations for general multiexponentiations of size m (see Remark 3 below for details).

The effect of this observation can be dramatic. If all m exponents in a multiexponentiation are at most 2^b (i.e., the exponents are b-bit numbers), then the number of group operations required by Pippenger's algorithm is very close to

$$(b/k)(m + 2^k). \tag{3}$$

Here, $k \leq b$ is an integer parameter chosen to minimize Expression (3). If $b \leq \log m$, then it is optimal to choose k slightly less than b, in which case the number of group operations is very close to m (formally, at most $(1 + o(1)) \cdot m$). Whereas, if the exponents are bounded merely by the size $|\mathbb{G}|$ of the cryptographic group then it is optimal to set $b \approx \log m$, in which case the number of group operations is roughly $m \cdot \log |\mathbb{G}|/\log(m)$. Hence, the time difference between committing to vectors with elements in $\{1, \ldots, m\}$ vs. arbitrary field elements is roughly $\log |\mathbb{G}|/\log(m)$. For example, suppose that $m = 2^{20}$ and the group size is roughly 2^{256}. Then it can be 10× faster to commit to a length-m vector with entries in $\{0, \ldots, m-1\}$ compared to an arbitrary length-m vector, and the savings can be even higher over larger groups.

In summary, if one instantiates Lasso with one of the above commitment schemes based on multiexponentiations, the prover's work is dominated by only c multiexponentiations of size m and c multiexponentiations of size $N^{1/c}$.[7]

By using other polynomial commitment schemes that avoid both multiexponentiations and FFTs [GLS+21, XZS22], we can have the prover perform $O(m)$ field operations and hash evaluations. This is the first lookup argument for structured tables achieving this prover cost profile, which is asymptotically optimal. Even for these commitment schemes, the fact that almost all of the field elements committed by the Lasso prover lie in the set $\{0, \ldots, \max\{m, N^{1/c}\} - 1\}$ can have substantial benefits for prover efficiency. This is because, even if one works over an extension field, almost all of the field elements that the Lasso prover needs to commit will lie in the base field. This speeds up field operations.

Remark 3. *Prior lookup arguments refer to the cost of a general m-sized multiexponentiation as linear in m [ZGK+22, DGM21]. However, as discussed above, the fastest known multiexponentiation algorithm, due to Pippenger [Pip80], requires a number of group operations that is (slightly) superlinear in m, namely $O\left(m\lambda/\log(\lambda m)\right)$, where $\lambda = \Theta(\log |\mathbb{G}|)$ is the security parameter and $\mathbb{G}$ is the group in which the multiexponentiation is occurring. Here, λ must be considered superlogarithmic in m, to ensure that adversaries running in time 2^λ are superpolynomial time. Similarly, prior works [ZGK+22, EFG22] refer to group exponentiations as group operations, when in fact they require up to $O(\log |\mathbb{G}|)$ many group operations.*

The proof length and verifier's runtime in Lasso is $O\left(\log(N) + \log(m + N^{1/c}) \log\log(m + N^{1/c})\right)$ field operations and hash evaluations, plus the time required to process an evaluation proof from the polynomial commitment scheme. Here, the $O(\log\left(m + N^{1/c}\right) \log\log(m + N^{1/c})\big)$ term arises due to Lasso's use of a *grand product argument* within its *sparse polynomial commitment scheme* (Section 1.1.1). A grand product argument is a SNARK for computing the product of many (possibly committed) values.

Lasso can use essentially any grand product argument, and we consider one in particular to have attractive costs. This grand product argument is due to Setty and Lee [SL20, Section 6]. It achieves a cost profile offering a middle ground between two simpler grand product arguments. Specifically, Thaler [Tha13] gave a grand product argument (for m public inputs) with proofs consisting of $O(\log^2 m)$ field elements that required no cryptographic commitments from the prover (i.e., Thaler's protocol is actually an *interactive proof* for grand products). Setty and Lee [SL20, Section 5] gave a grand product argument with $O(\log m)$ proof size that does require the prover to commit to $O(m)$ additional field elements. Setty and Lee [SL20, Section 6] explain how to combine the two schemes in a way that can reduce the number of committed values relative to [SL20, Section 5] from $O(m)$ to, say, $m/\log^3(m)$, at the cost of a small increase in proof size, from $O(\log m)$ to $O(\log(m) \log\log m)$ field elements.

[7] The prover does have to compute a (batched) evaluation proof for the committed polynomials. However, for Hyrax, Dory, and Sona, this requires field work linear in m and cryptographic work of just $O(\sqrt{m})$. See Section 1.1.3 for details.

Lasso's costs for unstructured tables. Even for totally unstructured tables, Lasso achieves a novel cost profile. For such tables, the Lasso prover performs $O(N)$ finite field operations, and with a suitable choice of a polynomial commitment scheme, it performs $O(m + \sqrt{N})$ cryptographic operations. More specifically, relative to Lasso for structured tables, the prover also has to provide an evaluation proof for the multilinear polynomial $\tilde{t}$ representing the table t. There are several polynomial commitment schemes that enable this evaluation proof to be computed with $O(N)$ finite field work and $\sqrt{N}$ cryptographic operations (e.g., a single $\sqrt{N}$-sized multiexponentiation), such as Hyrax and Sona.

In practice, finite field operations are orders of magnitude faster than cryptographic operations. Hence, even though the Lasso prover must do a linear number of finite field operations for unstructured tables, this is equivalent to a (substantially) sublinear number of cryptographic operations. Concretely, for an appropriate choice of polynomial commitment scheme (Hyrax or Sona), the Lasso prover's time can be dominated by either 2 multiexponentiations of length m (by using parameter $c = 2$), or one of length N (by setting $c = 1$), whichever is cheaper.

1.1.4 Third component: Surge, a generalization of Spark

Our third component relies on a reinterpretation of Spark as a technique for computing the inner product of an m-sparse committed vector of length N with a dense, highly structured lookup table of size N. Specifically, the table consists of all $(\log N)$-variate Lagrange basis polynomials evaluated at a specific point $r \in \mathbb{F}^{\log N}$. In particular, this table is a *tensor product* of c smaller tables, each of size $N^{1/c}$. We observe that many other lookup tables can similarly be decomposed has product-like expressions of $O(c)$ tables of size $N^{1/c}$, and that Spark extends to support all such tables.

Exploiting this perspective, we describe Surge, a generalization of Spark that allows an untrusted prover to commit to any sparse vector and establish the sparse vector's inner product with any dense, structured vector. We refer to the structure required for this to work as *Spark-only structure* (SOS for short).

In more detail, an SOS table T is one that can be decomposed into $\alpha = O(c)$ "subtables" $T_1, \ldots, T_\alpha$ of size $N^{1/c}$, such that each T_i is structured (i.e., the multilinear extension of each T_i can be evaluated quickly), and any entry $T[j]$ of T can be expressed as a simple expression of a corresponding entry into each of $T_1, \ldots, T_\alpha$.

Surge allows us to completely eliminate the sparse-dense sum-check protocol for tables possessing SOS structure. We call the resulting lookup argument SuperLasso. Relative to Lasso, SuperLasso reduces verification costs, and for many lookup tables it ensures that *all* $3cm + cN^{1/c}$ field elements committed by the Spark prover are in the set $\{0, 1, \ldots, \max\{m, N^{1/c}\} - 1\}$. This holds whenever $T_1, \ldots, T_\alpha$ have entries in $\{0, 1, \ldots, \max\{m, N^{1/c}\} - 1\}$. This reduces the prover's total costs to just $O(m)$ group *operations* when using the Hyrax or Dory polynomial commitment schemes, or a new scheme called Sona described in Section 1.2.

The key to this advantage of Surge is that Spark itself, as a sparse polynomial commitment scheme, considers a table with arbitrary field elements as entries (namely, as Lagrange basis polynomials evaluated at the randomly-chosen evaluation point r). Whereas Surge uses (decompositions of) the actual lookup table of interest in the SNARK application. These tables often have small entries.

We refer to this Surge-based version of Lasso, which supports SOS tables *without* invoking the sparse-dense sum-check protocol, as SuperLasso.

SuperLasso for small tables. We highlight that SuperLasso has new and attractive costs when applied to small tables in addition to large ones. Specifically, by setting $c = 1$, the SuperLasso prover commits to only about $3m + N$ field elements, and all of the committed elements are $\{0, 1, \ldots, \max\{m, N, q\}\}$ where q is the size of the largest value in the table.[8] SuperLasso is the first lookup argument with this property, which substantially speeds up commitment computation when m, N, and q are all much smaller than the size of the field over which the commitment scheme is defined.

[8] If using the grand product argument from [SL20, Section 6], a low-order number, say at most $O(m/\log^3 m)$, of large field elements need to be committed (see Section 1.1.3 for discussion).

Lasso vs. SuperLasso. SuperLasso is a superior choice to Lasso itself not only for SOS tables, but also in any setting where having the prover commit to a length-N vector is not a bottleneck. This includes application to any unstructured lookup tables, or whenever $m \geq N$.

As previously mentioned, setting $c = 1$ in SuperLasso, the prover commits to $3m + N$ field elements (in the unstructured case, an honest party must also commit to the table itself). However, the $+N$ term refers to a length-N vector with at most m non-zero entries, all of which are guaranteed to be in $\{0, 1, \ldots, m\}$. This commitment can be very fast to compute, involving a size-m multiexponentiation with all exponents in $\{0, 1, \ldots, m\}$. Evaluation proofs for polynomial commitment schemes applied to length-N vectors can be slow if $N \gg m$, but in the case of unstructured tables, or when $m \geq N$, the Lasso and SuperLasso provers need to spend time producing such evaluation proofs for committed polynomials of size at least N regardless (and amortization of these costs is possible).

In these settings, the benefits of SuperLasso over Lasso is two-fold: (1) lower verification costs and easier implementation, due to cutting out an invocation of the sum-check protocol, and (2) if the table has small entries, the SuperLasso prover only has to commit to small field elements, whereas cm of the field elements committed by the Lasso prover can be arbitrary.

In summary, SuperLasso is the lookup argument of choice for SOS tables, unstructured tables, or settings where the number of lookups is larger than the table size. Lasso is preferable for tables that are structured but not SOS, and where the number of lookups is smaller than the table size.

We are currently unaware of specific tables of interest that are structured but not SOS. We cover Lasso before SuperLasso for two reasons. First, Lasso is conceptually extremely simple if one treats the sparse polynomial commitment scheme and the sum-check protocol as a black-box, whereas SuperLasso fundamentally relies upon Surge, a new generalization of a sparse polynomial commitment scheme. Second, the structure required for a quasilinear-in-m time prover in Lasso is weaker than the SOS condition required to apply SuperLasso with $c > 1$.

Figure 1 compares Lasso's and SuperLasso's costs with those of existing lookup arguments.

1.2 Sona: A new transparent polynomial commitment scheme

Hyrax [WTS$^+$18] provides a multilinear polynomial commitment scheme (for random evaluation queries) with attractive prover costs. To commit to an ℓ-variate multilinear polynomial (which means the polynomial has $m = 2^\ell$ coefficients), the prover performs $\sqrt{m}$ multiexponentiations each of length $\sqrt{m}$. To compute an evaluation proof, the prover performs $O(m)$ field operations and a $O(\sqrt{m})$ exponentiations (this requires applying Bulletproofs to prove an inner product instance consisting of vectors of length $\sqrt{m}$); an evaluation proof consists of $O(\log m)$ group elements.

The downside of Hyrax's commitment scheme is that the verification costs are large: commitments consist of $\sqrt{m}$ group elements, and to verify an evaluation proof, the verifier has to perform two multiexponentiations of size $\sqrt{m}$. Dory [Lee21] can be thought of as reducing the Hyrax verifier's costs from $O(\sqrt{m})$ to $O(\log m)$, at the cost of requiring pairings, and requiring the verifier to perform a logarithmic number of operations in the target group of a pairing-friendly group.

We propose a new polynomial commitment scheme (for random evaluation queries) called Sona, which reduces Hyrax's verification costs in a different way. It uses two tools: Nova [KST22] and BabyHyrax (a simplified version of Hyrax in a manner that we describe next). In particular, BabyHyrax's evaluation proofs consist of $O(\sqrt{m})$ field elements, but it requires *no* cryptographic operations (BabyHyrax does not invoke Bulletproofs and instead proves the inner product instance by sending the underlying vectors).

With these tools in hand, in Sona, rather than sending a commitment cm consisting of $\sqrt{n}$ group elements as in BabyHyrax, the Sona prover sends the *hash* $a = h(\mathsf{cm})$ of the group elements. And rather than sending an evaluation proof π that consists of $\sqrt{n}$ group elements and convinces the BabyHyrax verifier that $p(r) = v$, the Sona prover uses Nova to prove that it knows:

- A vector cm in $\mathbb{G}^{\sqrt{m}}$ such that $a = h(\mathsf{cm})$.

Scheme	Table-specific Preprocessing	Proof size	Prover work group, field	Verifier work	Transparent?
Plookup [GW20b]	–	$5\mathbb{G}_1, 9\mathbb{F}$	$O(N)$, $O(N \log N)$	$2P$	No
Halo2 [BGH20]	–	$6\mathbb{G}_1, 5\mathbb{F}$	$O(N)$, $O(N \log N)$	$2P$	No
Caulk [ZBK+22]	$O(N \log N)$ exps	$14\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$15m$, $O(m^2 + m\log(N))$	$4P$	No
Caulk+ [PK22]	$O(N \log N)$ exps	$7\mathbb{G}_1, 1\mathbb{G}_2, 2\mathbb{F}$	$8m$, $O(m^2)$	$3P$	No
Flookup [GK22]	$O(N \log^2 N)$ exps	$7\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$O(m)$, $O(m \log^2 m)$	$3P$	No
Baloo [ZGK+22]	$O(N \log N)$ exps	$12\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$14m$, $O(m \log^2 m)$	$5P$	No
cq [EFG22]	$O(N \log N)$ exps	$8\mathbb{G}_1, 3\mathbb{F}$	$7m + o(m)$, $O(m \log m)$	$5P$	No
Lasso w/ Dory (structured table)	–	$O(\log(m))\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	$cm + o(cN^{\frac{1}{c}})$, $O(cm)$ $O(\sqrt{m})\ P$	$O(\log(m))\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	Yes
SuperLasso w/ Dory (SOS table)	–	$O(\log(m))\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	$o(cm + cN^{1/c})$, $O(cm)$ $O(\sqrt{m})\ P$	$O(\log(m))\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	Yes
Lasso w/ Dory (unstructured table)	–	$O(\log m)\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	$\min\{2m + O(\sqrt{N}), m + o(N)\}$, $O(m + N)$ $O(\sqrt{N})\ P$	$O(\log m)\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	Yes
Lasso w/ Sona (structured table)	–	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	$cm + o(cN^{\frac{1}{c}})$, $O(cm)$	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	Yes
SuperLasso w/ Sona (SOS table)	–	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	$o(cm + cN^{1/c})$, $O(cm)$	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	Yes
Lasso w/ Sona (unstructured table)	–	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	$\min\{2m{+}O(\sqrt{N}), N\}$, $O(m + N)$	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	Yes
Lasso w/ KZG +Gemini (structured table)	–	$O(\log m)\ \mathbb{G}_1$ $\tilde{O}(\log(m))\ \mathbb{F}$	$(c + 1)m + cN^{1/c}$, $O(m)$	$\tilde{O}(\log(m))\ \mathbb{F}$ $2P$ $O(\log m)\ \mathbb{G}_1$	No
Lasso w/ Orion (structured table)	–	$O(\log^2 m)\ \mathbb{H}$	-, $O(m)\ \mathbb{F}$ and $\mathbb{H}$	$O(\log^2(m))\ \mathbb{F}$ $O(\log^2(m))\ \mathbb{H}$	Yes

Figure 1: Dominant costs of prior lookup arguments vs. our work. Sona is the polynomial commitment scheme proposed in this work (Section 1.2). Other cost profiles for our schemes are possible by using other polynomial commitments. **Notation**: m is the number of lookups, N is the size of the lookup table. We assume $N \geq m$ for simplicity. For verification costs only, we assume that $m \leq \text{poly}(N)$, so that $\log m = \Theta(\log N)$ (if this does not hold, then verification costs in Lasso, but not SuperLasso, involve an additional additive $O(\log N)$ field elements and operations). The notation $\tilde{O}(\log m)$ notation hides a factor of $\log \log m$. Throughout, m "group work" for the prover refers to a multiexponentiation of size m, while "m exps" refers to m group exponentiations (m multiexponentiations are subject to a Pippinger speedup of a factor of roughly $O(\log(m\lambda))$ that m exponentiations are not). Operations involving $O(m)$ group *operations* (not exponentiations) are denoted via "$o(m)$ group work" to clarify that they are cheaper than a general m-sized multiexponentiation. SOS tables refer to those to which SuperLasso applies. $\mathbb{H}$ refers to hash evaluations, $\mathbb{F}$ to field operations, and $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ to relevant group elements or operations in a pairing-friendly group. P refers to pairing operations. KZG + Gemini refers to a polynomial commitment scheme for multilinear polynomials given in [BCHO22] obtained by transforming the KZG scheme for univariate polynomials. Finally, c denotes an arbitrary positive integer. The verifier costs with Brakedown, Orion, or Orion+ [GLS+21, XZS22, CBBZ23, DP23] (final row) also include $O(\sqrt{m})$-time pre-processing. Using Brakedown in place of Orion would yield a *field-agnostic* SNARK [GLS+21] at the cost of proofs of size $O(\sqrt{\lambda m})$ rather than $O(\lambda \cdot \log^2 m)$. Plookup and Halo2 are agnostic to the choice of polynomial commitment scheme, and that the reported costs and transparency properties in these rows refer to the case of using KZG commitments.

- A proof π that would have convinced the BabyHyrax verifier that cm is a commitment to a polynomial p such that $p(r) = v$.

The primary operations that Nova is applied to in this context are thus hashing a length-$\sqrt{m}$ vector cm, and applying the BabyHyrax verifier's checks on π, which mainly consists of two multiexponentiations of size $\sqrt{m}$ in $\mathbb{G}$. Applying the Nova prover to these computations results in $O(\sqrt{m}\log(\lambda)/\log(m))$ group operations for the prover. Hence, the prover's total work to compute an evaluation proof for Sona is $O(m)$ field operations and $O(\sqrt{m}\log(\lambda)/\log(m))$ group operations. Sona's evaluation proofs are a constant number of field elements and takes a constant-sized multiexponentiation to verify.

1.3 A companion work: JOLT, and the lookup singularity

In the context of SNARKs, a *front-end* is a transformation or compiler that turns any computer program into an *intermediate representation*—typically a variant of circuit-satisfiability—so that a back-end (i.e., a SNARK for circuit-satisfiability) can be applied to establish that the prover correctly ran the computer program on a witness. A companion paper called JOLT (for "Just One Lookup Table") shows that Lasso's ability to handle gigantic tables without either prover or verifier ever materializing the whole table (so long as the table is modestly "structured") enables substantial improvements in the front-end design.

The idea of JOLT is cleanest to describe in the context of a front-end for a simple virtual machine (VM), which in SNARK design has become synonymous with the notion of a CPU. A VM is defined by a set of primitive instructions (called the instruction set), one of which is executed at each step of the program. Typically, a front-end for a SNARK outputs a circuit, that for each step of the computation, (a) determines which instruction should be executed at that step and (b) executes the instruction. JOLT uses Lasso to replace part (b) at each step with a single lookup, into a gigantic lookup table.

Specifically, consider the popular RISC-V instruction set [RIS], targeted by the SNARK-focused project RISC-Zero[9]. For each of the primitive RISC-V instructions f_i, the idea of JOLT to create a lookup table that contains the entire evaluation table of f_i. For example, if f_i takes two 64-bit inputs, the table will have 2^{128} entries, whose (x, y)'th entry is $f_i(x, y)$. One can "glue together" the tables for each instruction, into a single table of size 2^{128} times the number of instructions.

JOLT shows that for each of the RISC-V instructions (including multiplication instructions and division and remainder instructions, as well as for 32-bit floating point arithmetic), the resulting table has the structure that we require to ensure that the sparse-dense sum-check prover performs $O(m)$ field operations and the verifier runs in logarithmic time. This leads to a front-end for VMs such as RISC-V that outputs much smaller circuits than prior front-ends, and has additional benefits such as easier auditability. Preliminary estimates from JOLT show that, when applied to the RISC-V instruction set over 64-bit data types, the prover commits to ≤ 45 field elements per step of the RISC-V CPU. Of these field elements, thirteen lie in $\{0, 1\}$, only thirteen are larger than 2^{32}, and only nine are larger than 2^{64}. This means that JOLT's prover costs when applied to a T-step execution of the RISC-V CPU on 64-bit data types is equivalent to computing under 10 multiexponentiations of size T if using a 256-bit field. Put another way, the JOLT prover's runtime is equivalent to committing to under 10 arbitrary field elements per step of the RISC-V CPU.

We are optimistic that JOLT can in fact be built entirely via SOS tables, enabling application of SuperLasso rather than Lasso. This would reduce the costs of the JOLT prover (if using an multiexponentiation-based polynomial commitment) by another factor of 2×-4×, and offer a significant reduction in verification costs as well.

We believe Lasso and JOLT together essentially achieve a vision outlined by Barry Whitehat called *the lookup singularity* [Whi]. The lookup singularity seeks to transform arbitrary computer program into "circuits" that *only* perform lookups. Whitehat's post outlines many benefits to achieving this vision, from improved performance to auditability and formal verification of the correctness of the front-end.

1.4 A summary of attractive features of Lasso and SuperLasso

Using large, structured tables. For structured tables that arise commonly in practice, ours is the first lookup argument that avoids the need for the verifier to commit to the table. We thereby enable the use of much larger tables than prior works. For example, we can use a structured table of size, say, 2^{64}. If we set $c = 4$, we can support 2^{20} lookups into this table, with the prover's runtime dominated by cryptographically committing to 12 vectors length 2^{20} plus 4 of length 2^{16}. Moreover, 8 out of the first 12 vectors consist entirely of elements in $\{0, \ldots, 2^{20}\}$, reducing the cost to commit to them via multiexponentiation-based commitment schemes by an order of magnitude.

Moreover, prior lookup arguments that have achieved prover time sublinear in the table size N (after pre-processing) all require a structured reference string (often called an SRS or proving key) of length N [ZBK+22, PK22, GK22, ZGK+22, EFG22]. This is because these works all use properties specific to KZG polynomial commitments [KZG10]. This severely limits the size of the tables to which it can be applied. The largest structured reference strings generated to date have size under 2^{30},[10] and yield proving keys up to 13 GBs in size (even when compressed). The proving key must be downloaded by any party wishing to act

[9] `https://www.risczero.com/`

[10] See for example `https://research.protocol.ai/sites/snarks/` and `https://github.com/AleoHQ/aleo-setup`.

as the SNARK prover. Furthermore, these works require at least $O(N \log N)$ group exponentiations in a preprocessing step.

Efficient support for gigantic structured tables is key to the new front-end techniques described in our companion paper, JOLT (Section 1.3), yielding a SNARK supporting a virtual machine abstraction with unprecedented efficiency. We believe that this property is the most important and qualitatively novel feature of Lasso.

Minimizing pre-processing for unstructured tables. As just mentioned, prior works [ZBK$^+$22, PK22, ZGK$^+$22, GK22, EFG22] involve a pre-processing phase for the prover involving at least $O(N \log N)$ group exponentiations, following the generation of an SRS of size N.

The only pre-processing required in Lasso for unstructured tables is committing to the table vector t using any multilinear polynomial commitment scheme. Some prior work have not even "counted" this table commitment as pre-processing [ZGK$^+$22, Table 1], considering the computation of this commitment "baked into" the definition of a lookup argument. We follow this convention in Table 1.

For unstructured tables, Lasso and SuperLasso, when combined with an appropriate polynomial commitment scheme such as Sona, Dory, or Hyrax, are the *first* transparent lookup arguments for which the prover's cryptographic work is sublinear in the table size (after the table has been committed). Furthermore, for these and other choices of polynomial commitment scheme, the table commitment entails a multiexponentiation of size N, which faster than $N \log N$ group exponentiations by a factor of $\Theta\left(\log(N) \cdot \log(\lambda N)\right)$, which concretely means a speedup of several orders of magnitude.

Attractive constants and concrete costs. When the number of lookups m is substantially smaller than the table size N, prior works had the prover commit to at least 8 field elements per lookup [ZBK$^+$22, PK22, GK22, ZGK$^+$22, EFG22]. Of the prior works to achieve 8 committed field elements per lookup, the first, called Caulk+ [PK22], requires $O(m^2)$ field work for the prover, compared to Lasso's $O(c\dot{m})$ field work, where c is such that $m \approx N^{1/c}$ (note that it is always the case that $cm \leq O(m \log N)$ regardless of how much bigger N is than m, and for practical parameter regimes, c can often be regarded as a constant). The other prior work to achieve 8 committed field elements per lookup is called cq [EFG22]. Of these 8 field elements in cq, one is in the set $\{0, \ldots, m-1\}$, while the other 7 are arbitrary field elements.

For comparison, when c is chosen large enough to ensure that $N \ll m^c$ and the table is structured, Lasso has the prover commit to roughly $3c$ field elements per lookup. However, $2c$ out of the $3c$ are in the set $\{0, \ldots, m-1\}$, meaning that for multiexponentiation-based commitment schemes, the prover costs behave as if the prover merely commits to about c arbitrary field elements per lookup. For SuperLasso applied to many important lookup tables, *all* of the committed field elements are in the set $\{0, \ldots, \max\{m, N^{1/c}\} - 1\}$, resulting in an *asymptotically faster prover* than any prior lookup argument.

cq also requires the online prover to perform an FFT at a time cost of $O(m \log m)$ field operations, larger than the $O(cm)$ field operations that the Lasso prover does (if $N \leq m^{o(\log m)}$).

Lookup arguments predating cq [ZBK$^+$22, GK22, ZGK$^+$22] required the prover to commit to more than eight field elements per lookup and additionally required at least $O(m \log^2 m)$ field work for the prover.

As another comparison point, prior works [CBBZ23, Hab22] also describe lookup arguments based on the sum-check protocol but do not achieve prover costs that are sub-linear in the table size N. For example, for any $m \leq N$, Hyperplonk's lookup argument requires the prover to commit to $4N$ field elements and also requires the prover to sort the lookup table. The Lasso and SuperLasso provers can commit to the same number of field elements when $m = N$ by setting $c = 1$, but at least $3N$ of the committed elements are in the set $\{0, \ldots, N-1\}$ (with SuperLasso, all $4N$ committed field elements are in this set if all table elements are in it). Moreover, Hyperplonk's lookup prover is more expensive than ours when m is even slightly smaller than N. Hyperplonk also inherently requires a pre-processing phase for the verifier involving cryptographically committing to a vector of length N. We do not require any pre-processing for the verifier for structured tables.[11]

[11] Unless one uses a polynomial commitment scheme that necessitates a pre-processing step.

We highlight that SuperLasso is extremely attractive for *arbitrary* tables when the number of lookups even when m is larger than the table size N. Not only does the SuperLasso prover (with $c = 1$) commit to only $3m+N$ field elements up to low-order terms, but all of the committed elements are in $\{0, 1, \dots, \max\{m, N, q\}\}$ where q is the size of the largest value in the table.

Flexibility. Our lookup arguments work with any multilinear polynomial commitment scheme. As mentioned above, several prior works with costs sublinear in the table size are specific to KZG commitments [ZBK+22, PK22, GK22, ZGK+22, EFG22]. This has additional implications for concrete performance, as KZG commitments require pairing-friendly elliptic curves, which typically have larger group elements and slower group operations than non-pairing-friendly curves.

1.5 Outline of the remainder of the paper

Section 2 covers technical preliminaries. Section 3 gives a detailed technical overview of Lasso, Spark, the sparse-dense sum-check protocol, Surge, and SuperLasso. Section 4 covers Lasso, treating Spark and the (prover implementation of the) sparse-dense sum-check protocol a black box. Section 5 covers Spark and our strengthened security analysis of it. Section 6 covers how to compute the sparse-dense sum-check prover's messages quickly. Section 7 covers Surge and SuperLasso.

Readers primarily interested in SuperLasso should read Sections 3, 4, 5, and 7 but can skip Section 3.2-3.3 and Section 6, which are devoted to the sparse-dense sum-check protocol.

2 Preliminaries

2.1 Multilinear extensions

An ℓ-variate polynomial $p \colon \mathbb{F}^\ell \to \mathbb{F}$ is said to be *multilinear* if p has degree at most one in each variable. Let $f \colon \{0,1\}^\ell \to \mathbb{F}$ be any function mapping the ℓ-dimensional Boolean hypercube to a field $\mathbb{F}$. A polynomial $g \colon \mathbb{F}^\ell \to \mathbb{F}$ is said to *extend* f if $g(x) = f(x)$ for all $x \in \{0,1\}^\ell$. It is well-known that for any $f \colon \{0,1\}^\ell \to \mathbb{F}$, there is a unique *multilinear* polynomial $\widetilde{f} \colon \mathbb{F} \to \mathbb{F}$ that extends f. The polynomial $\widetilde{f}$ is referred to as the *multilinear extension* (MLE) of f.

The *total degree* of an ℓ-variate polynomial p refers to the maximum sum of the exponents in any monomial of p. Observe that if p is multilinear, then its total degree is at most ℓ (but not all polynomials of total degree ℓ are multilinear).

A particular multilinear extension that arises frequently in the design of interactive proofs is the $\widetilde{\mathsf{eq}}$ is the MLE of the function $\mathsf{eq} : \{0,1\}^s \times \{0,1\}^s \to \mathbb{F}$ defined as follows:

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

An explicit expression for $\widetilde{\mathsf{eq}}$ is:

$$\widetilde{\mathsf{eq}}(x, e) = \prod_{i=1}^{s} \left(x_i e_i + (1 - x_i)(1 - e_i)\right). \tag{4}$$

Indeed, one can easily check that the right hand side of Equation (4) is a multilinear polynomial, and that if evaluated at any input $(x, e) \in \{0,1\}^s \times \{0,1\}^s$, it outputs 1 if $x = e$ and 0 otherwise. Hence, the right hand side of Equation (4) is the unique multilinear polynomial extending eq. Equation (4) implies that $\widetilde{\mathsf{eq}}(r_1, r_2)$ can be evaluated at any point $(r_1, r_2) \in \mathbb{F}^s \times \mathbb{F}^s$ in $O(s)$ time.[12]

[12] Throughout this manuscript, we consider any field addition or multiplication to require constant time.

Multilinear extensions of vectors. Given a vector $u \in \mathbb{F}^m$, we will often refer to the *multilinear extension of* u and denote this multilinear polynomial by $\widetilde{u}$. $\widetilde{u}$ is obtained by viewing u as a function mapping $\{0,1\}^{\log m} \to \mathbb{F}$ in the natural way: the function interprets its $(\log m)$-bit input $(i_1, \ldots, i_{\log m})$ as the binary representation of an integer i between 0 and $n-1$, and outputs u_i. $\widetilde{u}$ is defined to be the multilinear extension of this function.

Lagrange interpolation. An explicit expression for the MLE of any function is given by the following standard lemma (see [Tha22, Lemma 3.6]).

Lemma 1. *Let $f \colon \{0,1\}^\ell \to \mathbb{F}$ be any function. Then the following multilinear polynomial $\widetilde{f}$ extends f:*

$$\widetilde{f}(x_1, \ldots, x_\ell) = \sum_{w \in \{0,1\}^\ell} f(w) \cdot \chi_w(x_1, \ldots, x_\ell), \tag{5}$$

where, for any $w = (w_1, \ldots, w_\ell)$,

$$\chi_w(x_1, \ldots, x_\ell) \coloneqq \prod_{i=1}^{\ell} \left(x_i w_i + (1-x_i)(1-w_i)\right). \tag{6}$$

Equivalently, $\chi_w(x_1, \ldots, x_\ell) = \widetilde{\mathsf{eq}}(x_1, \ldots, x_\ell, w_1, \ldots, w_\ell)$.

The polynomials $\{\chi_w \colon w \in \{0,1\}^\ell\}$ are called the *Lagrange basis polynomials* for ℓ-variate multilinear polynomials. The evaluations $\{\widetilde{f}(w) \colon w \in \{0,1\}^\ell\}$ are sometimes called the coefficients of $\widetilde{f}$ *in the Lagrange basis*, terminology that is justified by Equation (5).

The sum-check protocol. Let g be some ℓ-variate polynomial defined over a finite field $\mathbb{F}$. The purpose of the sum-check protocol is for prover to provide the verifier with the following sum:

$$H \coloneqq \sum_{b \in \{0,1\}^\ell} g(b). \tag{7}$$

To compute H unaided, the verifier would have to evaluate g at all 2^ℓ points in $\{0,1\}^\ell$ and sum the results. The sum-check protocol allows the verifier to offload this hard work to the prover. It consists of ℓ rounds, one per variable of g. In round i, the prover sends a message consisting of d_i field elements, where d_i is the degree of g in its i'th variable, and the verifier responds with a single (randomly chosen) field element. The verifier's runtime is $O\left(\sum_{i=1}^{\ell} d_i\right)$, plus the time required to evaluate g at a single point $r \in \mathbb{F}^\ell$. In the typical case that $d_i = O(1)$ for each round i, this means the total verifier time is $O(\ell)$, plus the time required to evaluate g at a single point $r \in \mathbb{F}^\ell$. This is exponentially faster than the 2^ℓ time that would generally be required for the verifier to compute H. See [AB09, Chapter 8] or [Tha22, §4.1] for details.

SNARKs We adapt the definition provided in [KST22].

Definition 2.1. *Consider a relation $\mathcal{R}$ over public parameters, structure, instance, and witness tuples. A non-interactive argument of knowledge for $\mathcal{R}$ consists of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$, denoting the generator, the prover, the verifier and the encoder respectively with the following interface.*

- $\mathcal{G}(1^\lambda) \to \mathsf{pp}$*: On input security parameter λ, samples public parameters* pp.
- $\mathcal{K}(\mathsf{pp}, \mathsf{s}) \to (pk, \mathsf{vk})$*: On input structure* s*, representing common structure among instances, outputs the prover key pk and verifier key* vk.
- $\mathcal{P}(pk, u, w) \to \pi$*: On input instance u and witness w, outputs a proof π proving that* $(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}$.
- $\mathcal{V}(\mathsf{vk}, u, \pi) \to \{0,1\}$*: On input the verifier key* vk*, instance u, and a proof π, outputs 1 if the instance is accepting and 0 otherwise.*

A non-interactive argument of knowledge satisfies completeness if for any PPT adversary $\mathcal{A}$

$$\Pr\left[\mathcal{V}(\mathsf{vk}, u, \pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ (pk, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}), \\ \pi \leftarrow \mathcal{P}(pk, u, w) \end{array}\right] = 1.$$

A non-interactive argument of knowledge satisfies knowledge soundness if for all PPT adversaries $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that for all randomness ρ

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{vk}, u, \pi) = 1, \\ (\mathsf{pp}, \mathsf{s}, u, w) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{s}, u, \pi) \leftarrow \mathcal{A}(\mathsf{pp}; \rho), \\ (pk, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}), \\ w \leftarrow \mathcal{E}(\mathsf{pp}, \rho) \end{array}\right] = negl\lambda.$$

A non-interactive argument of knowledge is succinct if the size of the proof π is polylogarithmic in the size of the statement proven.

Polynomial commitment scheme We adapt the definition from [BFS20]. A polynomial commitment scheme for multilinear polynomials is a tuple of four protocols $\mathsf{PC} = (\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$:

- $pp \leftarrow \mathsf{Gen}(1^\lambda, \mu)$: takes as input μ (the number of variables in a multilinear polynomial); produces public parameters pp.
- $\mathcal{C} \leftarrow \mathsf{Commit}(pp, \mathcal{G})$: takes as input a μ-variate multilinear polynomial over a finite field $\mathcal{G} \in \mathbb{F}[\mu]$; produces a commitment $\mathcal{C}$.
- $b \leftarrow \mathsf{Open}(pp, \mathcal{C}, \mathcal{G})$: verifies the opening of commitment $\mathcal{C}$ to the μ-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\mu]$; outputs $b \in \{0, 1\}$.
- $b \leftarrow \mathsf{Eval}(pp, \mathcal{C}, r, v, \mu, \mathcal{G})$ is a protocol between a PPT prover $\mathcal{P}$ and verifier $\mathcal{V}$. Both $\mathcal{V}$ and $\mathcal{P}$ hold a commitment $\mathcal{C}$, the number of variables μ, a scalar $v \in \mathbb{F}$, and $r \in \mathbb{F}^\mu$. $\mathcal{P}$ additionally knows a μ-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\mu]$. $\mathcal{P}$ attempts to convince $\mathcal{V}$ that $\mathcal{G}(r) = v$. At the end of the protocol, $\mathcal{V}$ outputs $b \in \{0, 1\}$.

Definition 2.2. *A tuple of four protocols $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ is an extractable polynomial commitment scheme for multilinear polynomials over a finite field $\mathbb{F}$ if the following conditions hold.*

- ***Completeness.*** *For any μ-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\mu]$,*

$$\Pr\left\{\begin{array}{c} pp \leftarrow \mathsf{Gen}(1^\lambda, \mu);\ \mathcal{C} \leftarrow \mathsf{Commit}(pp, \mathcal{G}): \\ \mathsf{Eval}(pp, \mathcal{C}, r, v, \mu, \mathcal{G}) = 1 \wedge v = \mathcal{G}(r) \end{array}\right\} \geq 1 - negl\lambda$$

- ***Binding.*** *For any PPT adversary $\mathcal{A}$, size parameter $\mu \geq 1$,*

$$\Pr\left\{\begin{array}{c} pp \leftarrow \mathsf{Gen}(1^\lambda, m);\ (\mathcal{C}, \mathcal{G}_0, \mathcal{G}_1) = \mathcal{A}(pp); \\ b_0 \leftarrow \mathsf{Open}(pp, \mathcal{C}, \mathcal{G}_0);\ b_1 \leftarrow \mathsf{Open}(pp, \mathcal{C}, \mathcal{G}_1): \\ b_0 = b_1 \neq 0 \wedge \mathcal{G}_0 \neq \mathcal{G}_1 \end{array}\right\} \leq negl\lambda$$

- ***Knowledge soundness.*** *Eval is a succinct argument of knowledge for the following NP relation given $pp \leftarrow \mathsf{Gen}(1^\lambda, \mu)$.*

$$\mathcal{R}_{\mathsf{Eval}}(pp) = \{\langle(\mathcal{C}, r, v), (\mathcal{G})\rangle : \mathcal{G} \in \mathbb{F}[\mu] \wedge \mathcal{G}(r) = v \wedge \mathsf{Open}(pp, \mathcal{C}, \mathcal{G}) = 1\}$$

Scheme	Commit Size	Proof Size	$\mathcal{V}$ time	Commit time	$\mathcal{P}$ time
Brakedown-commit	$1\ \lvert\mathbb{H}\rvert$	$O(\sqrt{N \cdot \lambda})\ \lvert\mathbb{F}\rvert$	$O(\sqrt{N \cdot \lambda})\ \mathbb{F}$	$O(N)\ \mathbb{F}, \mathbb{H}$	$O(N)\ \mathbb{F}, \mathbb{H}$
Orion-commit	$1\ \lvert\mathbb{H}\rvert$	$O(\lambda \log^2 N)\ \lvert\mathbb{H}\rvert$	$O(\lambda \log^2 N)\ \mathbb{H}$	$O(N)\ \mathbb{F}, \mathbb{H}$	$O(N)\ \mathbb{F}, \mathbb{H}$
Hyrax-commit	$O(\sqrt{N})\ \lvert\mathbb{G}\rvert$	$O(\sqrt{N})\ \lvert\mathbb{G}\rvert$	$O(\sqrt{N})\ \mathbb{G}$	$O(N)\ \mathbb{G}$	$O(N)\ \mathbb{F}$
Dory	$1\ \lvert\mathbb{G}_T\rvert$	$O(\log N)\ \lvert\mathbb{G}_T\rvert$	$O(\log N)\ \mathbb{G}_T$	$O(N)\ \mathbb{G}_1$	$O(N)\ \mathbb{F}$
Sona (this work)	$1\ \lvert\mathbb{H}\rvert$	$O(1)\ \lvert\mathbb{G}\rvert$	$O(\sqrt{N})\ \mathbb{G}$	$O(1)\ \mathbb{G}$	$O(N)\ \mathbb{F}$, $O(\sqrt{N})\mathbb{G}$

Figure 2: Approximate costs of polynomial commitment schemes most relevant to this work (i.e., schemes where the cost of proving an evaluation incurs sublinear cryptographic operations), when committing to a multilinear ℓ-variate polynomial over $\mathbb{F}$, with $N = 2^\ell$. All are transparent. $\mathcal{P}$ time refers to the time to compute evaluation proofs. In addition to the reported $O(N)$ field operations, Hyrax and Dory require roughly $O(N^{1/2})$ cryptographic work to compute evaluation proofs. $\mathbb{F}$ refers to a finite field, $\mathbb{H}$ refers to a collision-resistant hash, $\mathbb{G}$ refers to a cryptographic group where DLOG is hard, and $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ refer to pairing-friendly groups. Columns with a suffix of "size" depict to the number of elements of a particular type, and columns with a suffix of "time" depict the number of operations (e.g., field multiplications or the size of multiexponentiations). Orion also requires $O(\sqrt{N})$ pre-processing time for the verifier.

2.2 Polynomial IOPs and polynomial commitments

Most SNARKs work by combining a type of interactive protocol called a *polynomial IOP* [BFS20] with a cryptographic primitive called a *polynomial commitment scheme* [KZG10]. The combination yields succinct *interactive* argument, which can then be rendered non-interactive via the Fiat-Shamir transformation [FS86], yielding a SNARK. Roughly, a polynomial IOP is an interactive protocol where, in one or more rounds, the prover may "send" to the verifier a very large polynomial q. Because q is so large, one does not wish for the verifier to read a complete description of q. Instead, in any efficient polynomial IOP, the verifier only "queries" q at one point (or a handful of points). This means that the only information the verifier needs about q to check that the prover is behaving honestly is one (or a few) evaluations of q.

In turn, a polynomial commitment scheme enables an untrusted prover to succinctly *commit* to a polynomial q, and later provide to the verifier any evaluation $q(r)$ for a point r chosen by the verifier, along with a proof that the returned value is indeed consistent with the committed polynomial. Essentially, a polynomial commitment scheme is exactly the cryptographic primitive that one needs to obtain a succinct argument from a polynomial IOP. Rather than having the prover send a large polynomial q to the verifier as in the polynomial IOP, the argument system prover instead cryptographically commits to q and later reveals any evaluations of q required by the verifier to perform its checks.

Whether or not a SNARK requires a so-called trusted setup, as well as whether or not it is plausibly post-quantum secure, is determined by the polynomial commitment scheme used. If the polynomial commitment scheme does not require a trusted setup, neither does the resulting SNARK, and similarly if the polynomial commitment scheme is plausibly binding against quantum adversaries, then the SNARK is plausibly post-quantum sound.

Our SNARKs can make use of any commitment schemes for *multilinear* polynomials q.[13] Here an ℓ-variate multilinear polynomial $q \colon \mathbb{F}^\ell \to \mathbb{F}$ is a polynomial of degree at most one in each variable. A brief summary of the multilinear polynomial commitment schemes that are most relevant to this work is provided in Figure 2.2. All of the schemes in the figure, except for KZG-based scheme, are transparent; Brakedown-commit and Orion-commit are plausibly post-quantum secure.

3 Technical overview

In Lasso, the prover commits to a natural *witness* M for its claim, and uses the sum-check protocol to prove that the committed object is indeed a valid witness. The witness simply lists, for each looked-up item a_i, the index $j(i)$ such that $a_i = t_{j(i)}$. A direct application of our sparse-dense sum-check protocol allows the verifier

[13] Any univariate polynomial commitment scheme can be transformed into a multilinear one, though the transformations introduce some overhead (e.g.,[CBBZ23, BCHO22, ZXZS20]).

to "check" that the committed object M is indeed a valid witness. We detail the sparse-dense sum-check protocol later in this technical overview.

3.1 Sparse polynomial commitment: Spark

To ensure that the sparse-dense sum-check protocol can be used to check the validity of M, the indices $j(i)$ described above must specified in a "sparse format". Specifically, M is an $m \times N$ matrix whose ith row is the unit vector $e_{j(i)} \in \{0,1\}^N$ (the vector with a 1 at index $j(i)$ and all other entries equal to 0).

This turns out to mean that the prover must use a *sparse* polynomial commitment scheme to commit to M. More precisely, the commitment is to the so-called *multilinear extension polynomial* of M, which is a multilinear polynomial in $\log m + \log N$ variables. When we say that the multilinear extension of M is a sparse polynomial, we mean that only m out of the $m \cdot N$ entries of M are non-zero.

To this end, we show that a protocol from Spartan [Set20] called Spark in fact yields a sparse polynomial commitment scheme (as discussed earlier, Spartan itself established security of the scheme under the assumption that the commitment is computed honestly, which sufficed for its application in the context of Spartan).

The Spark sparse polynomial commitment scheme works as follows. The prover commits to a *densified* representation of the sparse polynomial p, using any polynomial commitment scheme for "dense" (multilinear) polynomials. The densified representation of p is effectively a list of all of the monomials of p with a non-zero coefficient (and the corresponding coefficient). More precisely, the list specifies all *multilinear Lagrange basis polynomials* with non-zero coefficient. Details as to what are the multilinear Lagrange basis polynomials are not relevant to this overview (but can be found in Section 2.1).

When the verifier requests an evaluation $p(r)$ of the committed polynomial p, the prover returns the claimed evaluation v and needs to prove that v is indeed equal to the committed polynomial evaluated at r. Let c be such that $N = m^c$. As explained below, there is a simple and natural algorithm that takes as input the densified representation of p, and outputs $p(r)$ in $O(cm)$ time. Spark amounts to bespoke SNARK establishing that the prover correctly ran this sparse-polynomial-evaluation algorithm on the committed description of p. (This perspective on Spartan's sparse polynomial commitment scheme is somewhat novel, though it is partially implicit in the scheme itself and in an exposition of [Tha22, Section 16.2]).

The $O(c \cdot m)$-time algorithm for evaluating a multilinear polynomial of sparsity m. The algorithm simply iterates over each Lagrange basis polynomials specified in the committed densified representation, evaluates that basis polynomial at r, multiplies by the corresponding coefficient, and adds the result to the evaluation. Unfortunately, naively evaluating a $(\log N)$-variate Lagrange basis polynomial at r would take $O(\log N)$ time, resulting in a total runtime of $O(m \log N)$. The key to achieving time $O(cm)$ is to ensure that each Lagrange basis polynomial can be evaluated in $O(c)$ time. This is done via the following procedure, which is extremely reminiscent of Pippenger's algorithm for multiexponentiation (with m the size of the multiexponentiation, and Lagrange basis polynomials with non-zero coefficients corresponding to exponents).

Let us break the $\log N = c \log m$ variables of r into c blocks, each of size $\log m$, writing $r = (r_1, \dots, r_c) \in \left(\mathbb{F}^{\log m}\right)^c$. Then any $(\log N)$-variate Lagrange basis polynomial evaluated at r can be expressed as a product of c "smaller" Lagrange basis polynomials, each defined over only $\log m$ variables, with the i'th such polynomial evaluated at r_i.

There are only $2^{\log m} = m$ multilinear Lagrange basis polynomials over $\log m$ variables. Moreover, there are known algorithms that, for any input $r_i \in \mathbb{F}^{\log m}$, run in time m and evaluate all m of the $(\log m)$-variate Lagrange basis polynomials at r_i. Hence, in $O(cm)$ total time, the algorithm can evaluate *all* m of these basis polynomials at each r_i, storing the results in a (write-once) memory.

Given the contents of this memory, the algorithm can evaluate *any* given $\log(N)$-variate Lagrange basis polynomial at r by performing c lookups into memory, one for each block r_i, and multiplying together the results.[14]

[14] This is also closely analogous to the behavior of tabulation hashing discussed earlier in the introduction, which is why we

In this algorithm, we chose to break the $\log N$ variables into c blocks of length $\log m$ (rather than more, smaller blocks, or fewer, bigger blocks) to balance the runtime of the two phases of the algorithm, namely:

- The time required to "write to memory" the evaluations of all $(\log m)$-variate Lagrange basis polynomials at $r_1, \ldots, r_c$.
- The time required to evaluate $p(r)$ given the contents of memory.

In general, if we break the variables into $\log(N)/\ell$ blocks of size ℓ, the first phase will require time $(\log(N)/\ell)\cdot 2^\ell$, and the second will require time $O(m\ell)$.

How the prover proves it correctly ran the above $O(cm)$-time algorithm on p. To enable the untrusted prover to efficiently prove that it correctly ran this algorithm to compute $p(r)$, and in particular that it correctly "implemented" the write-once memory of size $c \cdot m$ over the course of the algorithm's execution, Spark uses simple fingerprinting techniques from the *offline memory checking* literature [BEG+91]. This effectively forces the prover to commit to the "execution trace" of the algorithm (which has size roughly $c \cdot m$, because the algorithm runs in time $O(c)$ for each of the m Lagrange basis polynomials with non-zero coefficient) plus $c \cdot N^{1/c} = O(cm)$ (because the offline memory-checking techniques require the prover to "replay" the entire contents of memory at the very end of the algorithm's execution, and this memory has size $c \cdot N^{1/c}$ if the algorithm breaks the $\log N$ variables into c blocks of size $\log(N)/c$). This is why Lasso's prover's winds up cryptographically committing to $3 \cdot c \cdot m + c \cdot N^{1/c}$ field elements.

Remark 4. *The cost incurred by Spark's prover to "replay" the entire contents of memory at the very end of the algorithm's execution can be amortized over multiple sparse polynomial evaluations. In particular, if the prover proves an evaluation of k sparse polynomials in the same number of variables, the cost incurred in the offline memory checking is reused across all k sparse polynomials.*

Accordingly, one can think of Lasso as a reduction from proving m lookups into an arbitrary table T of size N to *the same problem, but on c smaller tables, each of size $N^{1/c}$, that moreover are guaranteed to be structured even if T is not.* Following this reduction, the core of the protocol is a bespoke SNARK for proving correct execution of the particularly simple algorithm described above—the main work of the algorithm is to perform m lookups into c tables of size $N^{1/c}$. Moreover, these c tables are each structured in the sense that their multilinear extensions can be evaluated at any desired point in logarithmic time. The structured nature of these tables is crucial to Spark, as it enables the verifier to evaluate the multilinear extensions of the tables itself, thereby forcing the prover to use the "correct" tables.

In summary, Lasso can be viewed as using our new sparse-dense sum-check protocol and the commitment phase of Spark to reduce m lookups into an arbitrary table of size N to the problem of performing $c \cdot m$ lookups into c structured tables, each of size $N^{1/c}$. Spark evaluation proofs then offer a solution to this latter problem.

3.2 Sparse-dense sum-check protocol

A simple observation in prior works [ZBK+22, ZGK+22] shows that the matrix M described in Section 3.1 is a valid witness for a lookup argument if and only if $M \cdot t = a$. This turns out to be equivalent, up to negligible soundness error, to confirming that

$$\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) \cdot \tilde{t}(y) = \widetilde{a}(r), \tag{8}$$

for an r chosen at random by the verifier. Here, $\widetilde{M}$, $\widetilde{a}$ and $\widetilde{t}$ are the so-called *multilinear extension polynomials* (MLEs) of M, t, and a (see Section 2.1 for details).

Equation (8) can be computed with the sum-check protocol of Lund, Fortnow, Karloff, and Nisan [LFKN90], so long as the verifier can evaluate each of the three aforementioned polynomials at a random point. The key

chose to highlight this example from algorithm design.

question for this technical overview is: how fast can the prover be implemented in this application of the sum-check protocol?

The challenge is that this protocol is computing the inner product between two vectors in $u, t \in \mathbb{F}^N$, and we are unsatisfied with a prover time of $O(N)$ field operations. Here, entries of u are indexed by vectors $y \in \{0,1\}^{\log N}$ and the y'th entry of u equals $\widetilde{M}(r, y)$. Fortunately, we are *guaranteed* that at most m entries of u are non-zero. We leverage this guarantee to show how to implement the prover in this sum-check protocol with only $O(cm)$ field operations where c is such that $N = M^c$, so long as t is structured.

Brief overview of existing linear-time sum-check provers. In each round j of the sum-check protocol, the jth input to the polynomials $\widetilde{u}(y_1, \ldots, y_{\log N})$ and $\widetilde{t}(y_1, \ldots, y_{\log N})$ gets "bound" to a random field element r_j of the verifier's choosing. Existing linear-time sum-check protocols [CTY11, Tha13] applied to compute $\langle u, t\rangle$ achieve a prover that runs in $O(N)$ time by treating two or more entries of u (and of t) as a single entity once all their "bit-differences" are bound. That is, if $y, y' \in \{0,1\}^{\log N}$ agree on their last ℓ entries, then existing prover implementations treat u_y and $u_{y'}$ as a "single entity" for the final ℓ rounds of the protocol. This ensures that in each round j, the prover only needs to process $N/2^j$ entries, yielding total runtime of $O\left(\sum_{j=1}^{\log N} N/2^j\right) = O(N)$.

Earlier techniques [CMT12] can reduce the prover time instead to $O(m \cdot \text{polylog} N)$. However, achieving $O(m)$ prover time is substantially more challenging.

Brief overview of our sparse-dense sum-check prover. We reduce the prover time to $O(cm)$ as follows. Whereas prior works on the linear-time sum-check provers treat two indices $y, y' \in \{0,1\}^{\log N}$ of u as a single entity *after their last* "bit-difference" gets bound, our key idea (and fundamentally new technique) is to give a way treat any two indices y, y' as a single entity *until their first* bit-difference gets bound. For example, in round 1, the indices are split into just two entities: those with high-order bit equal to 0 and those with high-order bit equal to 1. In round 2, they are split into four entities, based on their highest-order two bits. And so forth.

This observation lets the prover handle each round $j = 1, \ldots, \log m$ in time $O(2^j)$. $\mathcal{P}$ begins to run into a problem once the protocol passes round $\log m$, since then $O(2^j)$ time starts to become larger than the $O(m)$ time bound we wish to satisfy. We think of this phenomenon, of the number of "relevant entities" to be tracked by the prover potentially doubling in each round, as *expansion*.

The idea then is to apply the techniques from the existing linear-time sum-check protocols, spending $O(m)$ time to "make the first $\log m$ bits of the index of each non-zero entry u_y of u no longer relevant to the protocol", by updating the entries of u to "incorporate" the binding of the first $\log m$ variables of $\widetilde{u}$ to the values $r_1, \ldots, r_{\log m}$. We call this procedure *consolidation*, as it reduces the number of "relevant entities" tracked by the prover from m down to 1.

The above procedure can be repeated every $\log m$ rounds. That is, for each contiguous "chunk" of input variables to $\widetilde{u}$ of size $\log m$, $\mathcal{P}$ spends $O(m)$ field work processing the rounds that bind variables in that chunk. This is $O(c \cdot m)$ field work total if $N = m^c$, i.e., if there are $\log N = c \cdot \log m$ variables.

In the above procedure, there's a tension whereby the more rounds that go by without consolidating, the more time $\mathcal{P}$ is paying each round. But consolidating costs $O(m)$ work. Hence, $\mathcal{P}$ does not want to consolidate too frequently. The optimal approach is to let expansion occur unchecked for $\log m$ rounds at a time—this balances the cost of consolidating vs. deferring consolidation.

3.3 Details on what constitutes a "structured table" for sparse-dense sum-check

Recall (Section 1.1.2) that our *sparse-dense sum-check protocol* exploits structure in the table in two ways: to implement the prover in the sum-check protocol in only $O(m)$ field operations, and to ensure that the verifier in the protocol, on its own, can quickly compute the information it needs about the table. Details follow, starting with how the verifier computes the information about the table that it needs to check the proof.

3.3.1 Ensuring the verifier can quickly compute the information it needs about the table

For many natural lookup tables, the multilinear polynomial $\widetilde{t}$ that "captures" the table in our protocol can be directly evaluated by the verifier at any desired point $r \in \mathbb{F}^{\log N}$ in $O(\log N)$ time. In fact, often the evaluation procedure only involves finite field additions and multiplication by powers of 2, rather than general finite field multiplications.

Hence, the verifier's work is inexpensive even if $\widetilde{t}$ is not cryptographically committed by the prover. In contrast, all prior lookup arguments that we are aware of require the prover to cryptographically commit to some polynomial "encoding" the table, which requires cryptographic costs at least linear in the table size.

Examples of lookup tables that satisfy this property include:

- All integers between 0 and $N-1$ (as described earlier in the section, this table enables range checks).
- All even (or all odd) integers between 0 and $2N$.
- All integers between 0 and N^2 whose natural binary representations have all even-indexed (or odd-indexed) bits set to 0. This table is useful for representing bitwise operations within constraint systems defined over large prime-order fields [BCG+18] including bitwise XOR, OR, and AND.

The key technical property that all of the above tables have in common is that the i'th table entry is a specific linear combination of the individual bits of the binary representation of i. Moreover, lookup tables that are the union of $O(\log N)$ many tables of the form above also fall into the class. That is, for such tables, the polynomial $\widetilde{t}$ used in Lasso can be evaluated by the verifier itself in $O(\log N)$ time.

As an example, for the table consisting of all finite field elements between 0 and $N-1$, if we index the table entries by $i \in \{0,1\}^{\log N}$, then the i'th table element is simply the field element $\sum_{j=0}^{\log(N)-1} 2^i \cdot i_j$. As we explain later, this means that for arbitrary finite field elements $(r_1, \ldots, r_{\log N}) \in \mathbb{F}$,

$$\tilde{t}(r_1, \ldots, r_{\log n}) = \sum_{j=0}^{\log N} 2^i \cdot r_j. \tag{9}$$

Clearly, Equation (9) can be evaluated in $O(\log N)$ time, and in fact only requires multiplications-by-powers-of-two and finite field additions.

Two more examples. An illustrative example that is somewhat more complicated than any of the above, is a lookup table introduced in our companion paper, JOLT, to handle bitwise operations more efficiently than prior works. For bitwise AND over b-bit inputs $x, y \in \{0,1\}^b$, the (x,y)'th entry of the appropriate (ordered) table is $\sum_{i=1}^{b} 2^{i-1} \cdot x_i \cdot y_i$. This turns out to imply that (after applying the transformation of Remark 1), for this table,

$$\widetilde{t}(x,y) = 2^{2b} \cdot \sum_{i=1}^{b} 2^{i-1} \cdot x_i \cdot y_i + \sum_{i=1}^{b} 2^{i-1} x_i + 2^b \sum_{i=1}^{b} 2^{i-1} y_i. \tag{10}$$

Here, the latter two sums and the 2^{2b} leading factor in front of the first sum are introduced by the transformation of Remark 1, from "unordered" to "ordered" lookups. Meanwhile, the first sum is derived by observing that for any two bits $a, b \in \{0,1\}$, $\mathsf{AND}(a,b) = a \cdot b$, and then taking the appropriate weighted sum of the bitwise AND of x and y to transform it into the associated integer. The result is that any $(x,y) \in \{0,1\}^b \times \{0,1\}^b$ gets mapped to the field element with binary representation $(x,y,z) \in \{0,1\}^{3b}$ where z is the bitwise AND of x and y.

Observe that the (x,y)'th entry of the evaluation table for the bitwise AND operation is *not* a weighted sum of the bits of x and y, because the function has total degree 2 owing to the first sum in Equation (10). Nonetheless, the multilinear extension $\widetilde{t}$ of this table can be evaluated by the verifier in logarithmic time, and the prover in the sparse-dense sum-check protocol applied to this table (Theorem 7) runs in $O(m)$ time.

As a final, more complicated example, JOLT also considers the lookup table associated with the integer comparison instruction LT (short for "less than"). This instruction takes two 64-bit inputs x and y, interprets them as (say, unsigned) integers, and outputs 1 if and only if $x \geq y$. The appropriate (ordered as per Remark 1) lookup table has (x, y)'th entry equal to the output of the LT instruction when run on x and y. As with the table AND above, we show that the multilinear extension of this table can be evaluated by the verifier in logarithmic time, and the prover in the sparse-dense sum-check protocol applied to this table (Theorem 7) runs in $O(cm)$ time when the table size is at most $O(m^{1/c})$.

3.3.2 Ensuring the sum-check prover runs in time close to m

Depending on just how "structured" the table y is, we prove two results regarding how fast the prover can be implemented in the sparse-dense sum-check protocol. First, using prior techniques [CMT12], we bound the prover time in the sparse-dense sum-check protocol by

$$O\left(m \cdot \log N \cdot \mathsf{evaltime}(\widetilde{t})\right),$$

where $\mathsf{evaltime}(\widetilde{t})$ denotes the time required to evaluate $\widetilde{t}(r')$ for any point $r' \in \mathbb{F}^{\log N}$. In particular, if $\mathsf{evaltime}(\widetilde{t}) = O(\log N)$, then this means $O(m \cdot \log^2 N)$ field operations for the prover.

Theorem 1. *The prover in the sparse-dense sum-check protocol applied to compute $\langle u, t \rangle$ can be implemented in $O\left(m \cdot \log N \cdot \mathsf{evaltime}(\widetilde{t})\right)$ field operations.*

Using the same techniques, we are in fact able to reduce the prover runtime for all tables considered in Section 3.3.1 to $O(m \log N)$ field operations (see Section 6.4).

Reducing the prover's runtime to the optimal $O(m)$ field operations is a substantially more challenging task. Intuitively, this requires the prover to spend a *constant* amount of work per non-zero entry of u, in total across *all* $\log N$ rounds of the protocol. This is a far more exacting task then achieving a constant amount of work per non-zero entry of u in *each* of the $\log N$ rounds, which is what the previous paragraph achieved.[15]

Hence, fundamentally new algorithmic techniques (outlined in Section 3.2) are required to reduce the prover's runtime to the optimal $O(m)$ field operations. We achieve this in Theorem 2 below for a large class of tables that captures all of those considered in Section 3.3.1 Informally, the key property that we require to achieve $O(m)$ prover time is that, given any evaluation $\widetilde{t}(r_1, \ldots, r_{\log N})$ of $\widetilde{t}$, changing the value of one variable from r_j to r'_j has a "simple" effect on the evaluation of $\widetilde{t}$.

Theorem 2 (Informal version of Theorem 7 in Section 6.5.2)**.** *Suppose there is some constant $C > 0$ such that $m \leq O(N^C)$. Suppose that $\widetilde{t}\colon \mathbb{F}^{\log N} \to \mathbb{F}$ satisfies the following property. For any $(r_1, \ldots, r_{\log N}) \in \mathbb{F}^{\log N}$ and any $r' \in \mathbb{F}^{\log N}$,*

$$\widetilde{t}(r_1, \ldots, r_{j-1}, r'_j, r_{j+1}, \ldots, r_{\log N}) = \mathsf{m} \cdot \widetilde{t}(r_1, \ldots, r_{j-1}, r_j, r_{j+1}, \ldots, r_{\log N}) + \mathsf{a}$$

where m and a are field elements that depend only on j, $r_1, \ldots, r_j, r_{j+1}$, and r'_j and can be computed in $O(1)$ time. Then the sparse-dense sum-check protocol prover can be implemented in $O(m)$ field operations.

In Lasso, the $O(Cm)$ field operations incurred by the prover in the sparse-dense sum-check protocol will certainly not be a bottleneck for the prover relative to the task of applying a sparse polynomial commitment scheme to commit to $O(cm)$ field elements (Section 1.1.1).

When the sparse-dense sum-check protocol is applied to compute $\langle u, t \rangle$ where u is m-sparse, the prover ultimately has to provide the verifier with the value $\widetilde{u}(r)$ for a randomly chosen $r \in \mathbb{F}^{\log N}$, where $\widetilde{u}$ is the multilinear extension polynomial of u. The fastest known algorithm for evaluating $\widetilde{u}(r)$ requires $O(Cm)$ field work. In fact this algorithm underlies Spark, our sparse polynomial commitment scheme (it is described in Section 3.1).

Hence, if this algorithm for evaluating sparse multilinear polynomials is optimal, then so is the $O(Cm)$-time algorithm of implementing the sparse-dense sum-check prover. Another implication of this connection is that, even though computing its messages in the sum-check protocol is not the bottleneck for the Lasso prover, any

[15] Because cryptographic operations are orders of magnitude more expensive than field operations, we believe that even $O(m \log N)$ field work would not be a bottleneck for the Lasso prover, compared to the work of committing to $O(m)$ field elements, unless $\log N$ is in the thousands or larger.

further improvement to the sparse-dense sum-check protocol is likely to lead to improvements in Spark as well (which *is* the bottleneck for the Lasso prover).

3.3.3 Expanding the set of structured tables: beyond multilinear extensions

In fact, for Lasso to avoid any party ever materializing the full table, is not necessary that the *multilinear* extension polynomial $\widetilde{t}$ of the table vector t be evaluatable in time logarithmic in the table size. It is enough for *any* low-degree extension polynomial $\hat{t}$ of t to be efficiently evaluatable. In general, if $\hat{t}$ can be evaluated in time $\mathsf{evaltime}(\hat{t})$ and has degree at most d in each of its $\log N$ variables, then the verification time for the sparse-dense sum-check protocol will be $O(d \log N + \mathsf{evaltime}(\hat{t}))$. The same techniques underlying the proof of Theorem 1 imply that the prover will perform at most $O(m \cdot d \log N \cdot \mathsf{evaltime}(\hat{t}))$ field operations.

While this is theoretically a major expansion of the class of structured tables and may be relevant in some practical applications, we believe that for a "critical mass" of tables of interest in practice, the multilinear extension of the table is sufficiently structured for the sparse-dense-sum-check verifier to run in logarithmic time and for the prover to perform $O(m)$ field operations (Theorem 2). Indeed, all tables considered in the companion work JOLT satisfy this property.

3.4 Surge: A Generalization of Spark

A re-imagining of Spark. A sparse polynomial commitment scheme can be viewed as having the prover commit to an m-sparse vector u of length N, where m is the number of non-zero coefficients of the polynomial, and N is the number of elements in a suitable basis. For univariate polynomials in the standard monomial basis, N is the degree, m is the number of non-zero coefficients, and u is the vector of coefficients. For an ℓ-variate multilinear polynomial p over the Lagrange basis, $N = 2^{\ell}$, and m is the number of $x \in \{0,1\}^{\ell}$ such that $p(x) \neq 0$, while u is the vector of evaluations of p at all inputs in $\{0,1\}^{\ell}$.

An evaluation query to p at input r returns the inner product of the sparse vector u with the dense vector t consisting of the evaluations of all basis polynomials at r. In the multilinear case, for each $S \in \{0,1\}^{\ell}$, the S'th entry of t is $\chi_S(r)$. In this sense, *any* sparse polynomial commitment scheme achieves the same effect as the sparse-dense sum-check protocol: it allows the prover to establish the value of the inner product $\langle u, t\rangle$ of a sparse (committed) vector u with a dense, structured vector t.

To obtain Surge, we critically examine the type of structure in t that is exploited by Spark, and introduce Surge as the natural generalization of Spark that supports any table t with this structure. Finally, we observe that many lookup tables critically important in practice exhibit this structure. For all such tables, we can modify Lasso to "skip" the sparse-dense sum-check protocol, as Surge itself instantiates the necessary functionality.

In Surge, the prover essentially establishes that it correctly ran a natural $O(cm)$-time algorithm for computing $\langle u, t\rangle$. The algorithm is a natural analog of the sparse polynomial evaluation algorithm described in Section 3.1: it iterates over every non-zero entry u_i of u, quickly computes $t_i = T[i]$ by performing one lookup into each of $O(c)$ "subtables" of size $N^{1/c}$, and quickly "combines" the result of each lookup to obtain t_i and hence $u_i \cdot t_i$. In this way, the the algorithm takes just $O(cm)$ time to compute the inner product $\sum_{i\colon u_i \neq 0} u_i \cdot t_i$.

Details of the structure needed to apply Surge. In the case of Spark itself, the dense vector t is simply the *tensor product* of smaller vectors, $t_1, \ldots, t_c$, each of size $N^{1/c}$. Specifically, Spark breaks r into c "chunks" $r = (r_1, \ldots, r_c) \in \left(\mathbb{F}^{(\log N)/c}\right)^c$, where r is the point at which the Spark verifier wants to evaluate the committed polynomial. Then t_i contains the evaluations of all $((\log N)/c)$-variate Lagrange basis polynomials evaluated at r_i. And for each $S = (S_1, \ldots, S_c) \in \left(\{0,1\}^{(\log N)/c}\right)^c$, the S'th entry of t is:

$$\prod_{i=1}^{c} t_i(r_i).$$

In general, Spark applies to any table vector t that is "decomposable" in a manner similar to the above. Specifically, suppose that $k \geq 1$ is an integer and there are $\alpha = k \cdot c$ tables $T_1, \ldots, T_\alpha$ of size $N^{1/c}$ and an α-variate multilinear polynomial g such that the following holds. For any $r \in \{0,1\}^{\log N}$, write $r = (r_1, \dots, r_c) \in \left(\{0,1\}^{\log(N)/c}\right)^c$, i.e., break r into c pieces of equal size. Suppose that for every $r \in \{0,1\}^{\log N}$,

$$T[r] = g\left(T_1[r_1], \dots, T_k[r_1], T_{k+1}[r_2], \dots, T_{2k}[r_2], \dots, T_{\alpha-k+1}[r_c], \dots, T_\alpha[r_c]\right). \tag{11}$$

Simplifying slightly, Surge allows the prover to commit to a m-sparse vector $u \in \mathbb{F}^N$ and prove that the inner product of u and the table T (or more precisely the associated vector t) equals some claimed value. And the cost for the prover is dominated by the following operations.

- Committing to $3\alpha m + \alpha \cdot N^{1/c}$ field elements, where $2\alpha m + \alpha N^{1/c}$ of the committed elements are in the set
$$\{0, 1, \dots, \max\{m, N^{1/c}\} - 1\},$$
and the remaining αm of them are elements of the subtables $T_1, \dots, T_\alpha$. For many lookup tables T, these elements are themselves in the set $\{0, 1, \dots, N^{1/c} - 1\}$.
- Let b be the number of monomials in g. Then the Surge prover performs $O(k \cdot \alpha N^{1/c}) = O(bcN^{1/c})$ field operations. In many cases, the factor of b in the number of prover field operations can be removed (see Section **??** for details).

We refer to tables that can be decomposed into subtables of size $N^{1/c}$ as per Equation (67) as having *Spark-only structure* (SOS).

4 Lasso: A highly performative lookup argument

This section describes Lasso, a lookup argument (Definition 1.1), with a cost profile summarized earlier (§1). We describe Lasso by depicting a polynomial IOP, in which the prover sends a single multilinear polynomial that we denote by $\widetilde{M}$. The polynomial IOP can be transformed into a SNARK by combining it with any multilinear polynomial commitment scheme.

More specifically, recall from Section 2.1 that, given a vector $v \in \mathbb{F}^N$, $\widetilde{v}$ denotes the multilinear polynomial obtained as follows: interpret v as the evaluation table of a function mapping $\{0,1\}^{\log N} \to \mathbb{F}$, and define $\widetilde{v}$ to be the multilinear extension of that function. In Lasso, the commitments cm_a and cm_t to the vectors a and t that are inputs to the verifier are commitments to the multilinear extensions $\widetilde{a}$ and $\widetilde{t}$ (if t is structured, then the commitment cm_t is omitted). Likewise, the Lasso prover sends a commitment to $\widetilde{M}$ in place of sending a complete description of $\widetilde{M}$. The prover's runtime in Lasso is primarily determined by the time to commit to $\widetilde{M}$, and to compute one evaluation proof for each of the three committed polynomials $\widetilde{a}$, $\widetilde{t}$, and $\widetilde{M}$.

4.1 Key tool: Sparse polynomial commitments

Consider a function $f \colon \{0,1\}^\ell \to \mathbb{F}$. Let m denote the number of inputs x such that $f(x) \neq 0$. If $m \ll n := 2^\ell$, then we say that f is *sparse*. In particular, this means that the MLE $\widetilde{f}$ of f has only m non-zero coefficients in the Lagrange basis (§2.1). Hence, if $m \ll n$, we say that $\widetilde{f}$ is *sparse in the Lagrange basis*.

Directly applying one of the polynomial commitment schemes discussed in Section 2.1 results in costs that grow with 2^ℓ rather than with m. To give an example, in Hyrax's polynomial commitment [WTS$^+$18], the commitment can be computed with $O(m)$ group exponentiations, but the size of the commitment remains $O(\sqrt{n})$ regardless of how sparse is the polynomial being committed. We refer to the commitment schemes of Section 2.1 as *dense polynomial commitment schemes*, because their costs their costs are appropriate for dense (i.e., non-sparse) polynomials, but much larger than necessary for sparse polynomials.

Spartan [Set20] describes a transformation to turn any multilinear polynomial commitment scheme with costs that grow with n into one where n is replaced by $O(m \log n)$ ([GLS$^+$21, §6], [Tha22, §16.2], and Appendix B provide an alternate exposition). Spartan [Set20] further reduces $O(m \log n)$ to the optimal value of $O(m)$, but under the assumption that certain metadata associated with the sparse polynomial is committed honestly (i.e., by the verifier itself). In Section 5, we prove that the scheme remains secure even if that metadata is committed by the prover, thereby achieving the first "standard" sparse polynomial commitment scheme with optimal prover costs.

4.2 The lookup argument

We identify a simple and clean witness for the lookup arguments underlying prior works [ZBK+22, PK22, ZGK+22], but have the prover commit to the witness and check its validity via totally different techniques.

Satisfying "witness" for the lookup argument. A satisfying witness is simply a matrix $M \in \mathbb{F}^{m \times N}$ such that

- the rows of M are unit vectors, i.e., each row has one entry equal to 1 and all other entries equal to 0.
- $M \cdot t = a$.

If M satisfies both bullets above, then the i'th row of M is the unit vector corresponding to the *index* in the table "containing" a_i. That is, suppose that the ith row of M equals the unit vector $e_j \in \{0,1\}^N$, i.e., $e_{j,j} = 1$ and $e_{j,k} = 0$ for all $k \neq j$. Then $a_i = t_j$.

Clearly such a matrix M exists if and only if for each a_i there is some table index j such that $a_i = t_j$. View M as a function $M \colon \{0,1\}^{\log m} \times \{0,1\}^{\log N} \to \mathbb{F}$, and let $\widetilde{M}$ be the multilinear extension. Note that if the prover is honest, then $\widetilde{M}$ has sparsity exactly m in the Lagrange basis.

Tools to check if the witness is satisfying. To check if the prover's witness satisfies the two bullet points above, we use two new tools: (1) we specialize Spartan's sparse polynomial commitment scheme so the prover can only commit to sparse polynomials that extend matrices where each row is a unit vector (this not only optimizes costs in the sparse commitment commitment scheme, but also avoids having to check M has the desired structure via other protocols); and (2) we introduce the *sparse-dense sum-check protocol* to compute the inner product of two vectors, one of which has at most m non-zero entries, and show that the prover can be implemented with $O(m)$ field operations regardless of the length n of the vectors.

The polynomial IOP. In the polynomial IOP, the prover sends $\widetilde{M}$ to the verifier as the first message in the protocol. (In the succinct arguments resulting from this polynomial IOP, the prover will commit to $\widetilde{M}$ using our specialized version of Spartan's sparse polynomial commitment scheme, which additionally ensures that each row of M is a unit vector.) Below, we check that $M \cdot t = a$ using the linear-time sum-check protocol.

Define $b = M \cdot t$, and let $\widetilde{b}$ denote the multilinear extension of b. Then it is easy to see that:

$$\widetilde{b}(r) = \sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r,j) \cdot \widetilde{t}(j). \tag{12}$$

Indeed, the RHS is a multilinear polynomial in the variables of r, and by the definition of matrix-vector multiplication, it agrees with b at all inputs in $\{0,1\}^{\log m}$. Hence, the RHS is the unique multilinear polynomial extending b.

Accordingly, to confirm that $M \cdot t = a$, it suffices to confirm that $\widetilde{b}$ and $\widetilde{a}$ are the same polynomial. To do this, it suffices for the verifier to pick a random input $r \in \mathbb{F}^{\log m}$ and confirm that $\widetilde{b}(r) = \widetilde{a}(r)$ (up to soundness error $\log(m)/|\mathbb{F}|$, by the Schwartz-Zippel lemma). The verifier can learn $\widetilde{a}(r)$ with one evaluation query to $\widetilde{a}$.

To learn $\widetilde{b}(r)$, the verifier applies the sumcheck protocol to the $(\log N)$-variate polynomial $g(j) \coloneqq \widetilde{M}(r,j) \cdot \widetilde{t}(j)$, in order to compute the right hand side of Equation (12). At the end of the sum-check protocol, the verifier needs to evaluate $\widetilde{M}(r,r')$ and $\widetilde{t}(r')$ for a randomly chosen point $r' \in \mathbb{F}^{\log N}$. This can be done with one evaluation query to $\widetilde{M}$ and one to $\widetilde{t}$.

As explained later (Section 6), $\widetilde{M}(r,j) = 0$ for all but at most m values of $j \in \{0,1\}^{\log N}$. Hence, standard techniques [CMT12] suffice to implement the prover in the application of the sum-check protocol to $g(j) \coloneqq \widetilde{M}(r,j) \cdot \widetilde{t}(j)$ with a number of field operations that is *quasilinear* in m. However, we wish to lower this to $O(m)$, especially considering that we would like to apply Lasso to (structured) tables so large that $\log N$ is well over one hundred. We call this $O(m)$-time prover algorithm the *sparse-dense sum-check protocol.* We defer a detailed description of the algorithm to Section 6. The consequence of this result is captured in Theorem 3.

- Polynomial commitments to the multilinear polynomials $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$ and $\widetilde{t}\colon \mathbb{F}^{\log N} \to \mathbb{F}$ are given to the verifier as input. The commitment to $\widetilde{t}$ is omitted if $\widetilde{t}(r)$ can be evaluated at any point $r \in \mathbb{F}^{\log N}$ in logarithmic time.
- The prover $\mathcal{P}$ sends a polynomial commitment to the MLE $\widetilde{M}$ of a matrix $M \in \{0,1\}^{m \times N}$ using our specialized version of Spartan's sparse polynomial commitment scheme.
- The verifier $\mathcal{V}$ picks a random $r \in \mathbb{F}^{\log m}$ and sends r to $\mathcal{P}$. The verifier makes one evaluation query to $\widetilde{a}$, to learn $\widetilde{a}(r)$.
- $\mathcal{P}$ and $\mathcal{V}$ apply the sum-check protocol to the $(\log N)$-variate polynomial $g(j) \coloneqq \widetilde{M}(r,j) \cdot \widetilde{t}(j)$, to confirm that
$$\widetilde{a}(r) = \sum_{j \in \{0,1\}^{\log N}} g(j).$$
- At the end of the sum-check protocol, the verifier needs to evaluate $\widetilde{M}(r,r')$ and $\widetilde{t}(r')$ for a random point $r' \in \mathbb{F}^{\log N}$ that is chosen entry-by-entry over the course of the sum-check protocol. This costs one evaluation query to $\widetilde{M}$ and one to $\widetilde{t}$.

Figure 3: Description of our lookup argument. Here, a denotes the vector of lookups and t the vector capturing the lookup table (Definition 1.1). Polynomial commitments to the multilinear extension polynomials $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$ and $\widetilde{t}\colon \mathbb{F}^{\log N} \to \mathbb{F}$ are given to the verifier as input (the commitment to $\widetilde{t}$ can be omitted if $\widetilde{t}$ can be evaluated at any point in logarithmic time).

Theorem 3. *There is a polynomial IOP that can be combined with an appropriate commitment scheme for sparse polynomials to obtain a lookup argument for m lookups into a table of size N. The polynomial IOP requires that the characteristic of the field $\mathbb{F}$ over which the lookup argument is defined is at least* $\max\{m, N\}$. *The polynomial IOP has soundness error at most*

$$(\log m + 2\log N)/|\mathbb{F}|.$$

The honest prover sends one polynomial $\widetilde{M}$, which is the multilinear extension of a matrix in $\{0,1\}^{m \times N}$ in which each row is a unit vector. The verifier queries the polynomials $\widetilde{a}$, $\widetilde{t}$, and $\widetilde{M}$ once each, to obtain the values $\widetilde{a}(r)$, $\widetilde{t}(r')$, and $\widetilde{M}(r,r')$. The proof length and verifier time is $O(\log m + \log N)$ field elements and operations respectively, plus the time to query the aforementioned polynomials. The prover time is $O(m)$ field operations if the table satisfies the properties of Theorem 7, plus the time to answer the above queries to the polynomials $\widetilde{a}$, $\widetilde{t}$, and $\widetilde{M}$.

Proof. Completeness holds because, if the prover is honest, then $Mt = a$, and the verifier's checks pass with probability 1.

Soundness holds by the following reasoning. If the prover's claim is false, then $Mt \neq a$. By the Schwartz-Zippel lemma, with probability at least $1 - \log(m)/|\mathbb{F}|$, $\widetilde{b}(r) \neq \widetilde{a}(r)$. The sum-check protocol (bulletpoint four of Figure 3) forces the prover to provide $\widetilde{b}(r)$, and it has soundness error $2\log(N)/|\mathbb{F}|$. By a union bound, the total soundness error is at most
$$\frac{\log(m) + 2\log(N)}{|\mathbb{F}|}.$$

The prover runtime assertion is immediate from Theorem 7, which is stated in proved in Section 6.5.2. □

We remark that while the soundness error of the polynomial IOP in Figure 3 is $O(\log(N)/|\mathbb{F}|)$, actual SNARKs derived from the polynomial IOP will use the sparse polynomial commitment scheme of Section 5 (Spark) to commit to $\widetilde{M}$. And this sparse polynomial commitment scheme is itself based on a polynomial IOP with soundness error $O(N/|\mathbb{F}|)$. So the resulting lookup argument will need to work over fields of size substantially larger than N to ensure adequate soundness error.

4.3 Eliminating the commitment to $\tilde{t}$ for structured tables

In the lookup argument above (Figure 3), the commitment to the multilinear extension $\widetilde{t}$ of the table vector t can be eliminated entirely if the verifier can evaluate $\widetilde{t}(r')$ on its own with $O(\log N)$ field operations (note that the verifier in the protocol has to perform $O(\log N)$ field operations simply to run the various invocations of the sum-check protocol). This turns out to be possible for many, if not most, lookup tables used in practice. Examples are given below.

- Suppose the table contains integers between 0 and $N-1$. As discussed in Section 1, this table is important for range checks. Then

$$\widetilde{t}(x_1, \ldots, x_{\log N}) = \sum_{i=1}^{\log N} 2^{i-1} \cdot x_i. \tag{13}$$

  Indeed, the right hand side of Equation (13) is a multilinear polynomial in the variables $(x_1, \ldots, x_{\log N})$ and maps the binary representation of the integer $j \in \{0, 1, \ldots, N-1\}$ to j. Hence, it is the unique multilinear polynomial that extends the function $t \colon \{0,1\}^{\log N} \to \mathbb{F}$ that maps the binary representation of j to j. Equation (13) can be evaluated at any input $r \in \mathbb{F}^{\log N}$ with $O(\log N)$ field operations (in fact, using only multiplications by 2 and field additions).

- Suppose the table contains all even integers between 0 and $2N-1$. Then it is straightforward to check that

$$\widetilde{t}(x_1, \ldots, x_{\log N}) = \sum_{i=1}^{\log N} 2^{i} \cdot x_i. \tag{14}$$

  If the table contains all odd integers between 0 and $2N-1$, then:

$$\widetilde{t}(x_1, \ldots, x_{\log N}) = 1 + \sum_{i=1}^{\log N} 2^{i} \cdot x_i. \tag{15}$$

  Equations (14) and (15) can clearly be evaluated at any input $r \in \mathbb{F}^{\log N}$ with $O(\log N)$ field operations (in fact, only multiplications by 2 and field additions are required).

- Suppose the table contains all integers between 0 and N^2 whose natural binary representation have all even-indexed bits set to 0. Then it is straightforward to check that:

$$\widetilde{t}(x_1, \ldots, x_{\log N}) = \sum_{i=1}^{\log N} 2^{2i-1} \cdot x_i. \tag{16}$$

  Similarly, if the table contains all integers between 0 and N^2 whose natural binary representation have all odd-indexed bits set to 0, then

$$\widetilde{t}(x_1, \ldots, x_{\log N}) = \sum_{i=1}^{\log N} 2^{2(i-1)} \cdot x_i. \tag{17}$$

  These tables table is important for representing bitwise operations within constraint systems defined over large prime-order fields [BCG+18]. This includes bitwise XOR, OR, and AND.

The key technical property that all of the tables above satisfy is that the i'th table entry is a specific linear combination of the individual bits of the binary representation of i. Other examples that arise in practice and fall into this class include bit-rotations and more generally arbitrary permutations of the binary representation of field elements. By this, we mean tables in which the i'th entry is obtained via a procedure of the following form: "Take the binary representation of i, apply some fixed permutation π to its bits, and turn the result back into an integer (interpreted as a field element)". Formally, if to-field denotes the canonical injection from $\{0,1\}^{\log N}$ to $\{0, 1, \ldots, N-1\}$ and to-bits its inverse, then this means $t_i = \text{to-field}(\pi(\text{to-bits}(i)))$.

Unions of structured tables. Moreover, lookup tables that are the union of $O(\log N)$ many tables of the form above also fall into the class. That is, for such tables, the polynomial $\widetilde{t}$ used in our lookup argument can be evaluated by the verifier itself in $O(\log N)$ time.

Indeed, suppose that the lookup table is the union of ℓ different tables, where assume for simplicity that each table has size N. For $i = 1, \ldots, \ell$, let $t_i \in \mathbb{F}^N$ denote the vector representing table i as per Definition 1.1, and let $t \in \mathbb{F}^{\ell \cdot N}$ denote the concatenation of these vectors. Then viewing t as a function mapping $\{0,1\}^{\log \ell} \times \{0,1\}^{\log N} \to \mathbb{F}$, in the natural way, the following equation holds, with $x \in \{0,1\}^{\log \ell}$ and $y \in \{0,1\}^{\log N}$:

$$\widetilde{t}(x,y) = \sum_{z \in \{0,1\}^{\log \ell}} \widetilde{\mathsf{eq}}(z,x) \cdot \widetilde{t}_{\mathsf{to\text{-}int}(z)}(y), \tag{18}$$

where $\mathsf{to\text{-}int}(z) = \sum_{i=1}^{\ell-1} 2^{i-1} \cdot z_i$ denotes the integer of which z is the binary representation. This is because the right hand side of Equation (18) is a multilinear polynomial in the variables of x and y and agrees with t at all inputs in $\{0,1\}^{\log \ell} \times \{0,1\}^{\log N}$. Hence, it is the unique multilinear polynomial extending t. Moreover, Equation (18) can be evaluated at any point $(r_1, r_2) \in \mathbb{F}^{\log \ell} \times \mathbb{F}^{\log N}$ in time $O(\ell)$ plus the amount of time required to evaluate each of $t_1, \ldots, t_\ell$ at r_2.

We remark that if tables t_1 and t_2 are structured as above, meaning their multilinear extensions $\tilde{t}_1$ and $\tilde{t}_2$ can each be evaluated in logarithmic time, then so is the table $t_1 + \alpha t_2$ for any $\alpha \in \mathbb{F}$. As noted in [EFG22], this is important for vector lookups (see [GW20b, Section 4]).

4.4 Lookups for ordered tables

Per Remark 1, in some settings one may which to view the lookup table as an *ordered* list of values rather than an unordered set. Accordingly, suppose that the lookups are for (index, value) pairs, and the prover must prove that $t_{i_j} = a_j$ for each pair (i_j, a_j) in the list of lookups. Let R be such that all table elements are in $\{0, 1, \ldots, R-1\}$, and assume that $m \cdot N$ is less than the field characteristic. Replace each table element t_i with $t_i \cdot R + i$ to obtain a modified table t', and replace each lookup pair (i_j, a_j) with the field element $a'_j = a_j \cdot R + i_j$. Apply a range check to confirm that $i_j \in \{0, 1, \ldots, R-1\}$ for all lookups. Clearly, $t_{i_j} = a_j$ implies that $a'_j = t'_{i_j}$. Conversely, under the guarantee that each i_j is in $\{0, \ldots, R-1\}$, if $t_{i_j} \neq a_j$ then a'_j is not in the set represented by t'. Hence, one can apply a lookup argument satisfying Definition 1.1 to a' and t' to confirm that $t_{i_j} = a_j$ for all pairs (i_j, a_j) in the list of lookups.

If R is chosen to be the smallest power of 2 bounding all the table elements, then the range check can be implemented via a lookup into the structured table $\{0, 1, \ldots, R-1\}$, and moreover t' is structured if and only if t is structured.

The range check on i_j can be omitted if the value i_j is provided by a party that is guaranteed to be honest. This is the case in our companion paper, a front-end called JOLT, where the indices i_j being "looked up" are in fact input(s) to a computer instruction, and those input(s) are computed "by the circuit" output by the front-end. This circuit is "trusted", i.e., correctness of the front-end *means* that the circuit applies the appropriate instruction to the correct input(s) at each step of the computation.

4.5 Beyond multilinear extensions, and a generic speedup over bit-decomposition

In our lookup argument (Figure 3) there is nothing special about using the *multilinear* extension $\widetilde{t}$ of t. We can replace $\widetilde{t}$ with *any* extension polynomial $\hat{t}$ of t (recall from Section 2.1 that $\hat{t}$ extends t if $\hat{t}(i) = t(i)$ for all $i \in \{0,1\}^{\log N}$).

Indeed, the key equality (Equation (12)) that for $b = M \cdot t$ that

$$\widetilde{b}(r) = \sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r,j) \cdot \widetilde{t}(j).$$

holds with $\widetilde{t}$ replaced by *any* extension $\hat{t}$ of t. This is because the right hand side of the equation is multilinear in r regardless of whether or not $\widetilde{t}(j)$ is multilinear in j.

This means that if the multilinear extension $\widetilde{t}$ of t cannot be evaluated by the verifier sufficiently quickly (say, in polylogarithmic time), we can replace $\widetilde{t}$ with another extension $\hat{t}$ that can be. This does potentially increase the costs of the sum-check protocol applied to compute

$$\sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r, j) \cdot \widetilde{t}(j)$$

(see Figure 4). In particular, the length of each message j from prover to verifier across the $\log N$ rounds of the sum-check protocol grows from 3 field elements (specifying a degree-2 univariate polynomial) to $1 + d_j$ where d_j is the degree of $\hat{t}$ in its j'th variable. Assuming that $d_j \leq \text{polylog}(N)$ for each variable $j = 1, \ldots, \log N$, and applying standard techniques to implement the sum-check protocol prover [CMT12], we obtain a prover performing $O(m \cdot \text{polylog}(N))$ field operations.

This result can be viewed as formalizing the following intuitive statement: any operation that can be "efficiently performed via bit-decomposition" (meaning there is an arithmetic or Boolean formula of size polynomial in the number of bits in the bit-decomposition that outputs the result of the operation) can be solved by Lasso with $\mathcal{P}$ only needing to cryptographically commit to $3c$ many field elements per lookup ($\mathcal{P}$ also performs polylogarithmic field operations per lookup). In contrast, as explained in Section 1, naive bit-decomposition of integers in the range $\{0, 1, \ldots, R-1\}$ requires the prover to commit to $\log R$ many field elements.

- Polynomial commitment to the multilinear polynomial $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$.
- The prover $\mathcal{P}$ sends a polynomial commitment to the MLE $\widetilde{M}$ of a matrix $M \in \{0,1\}^{m \times N}$ using our specialized version of Spartan's sparse polynomial commitment scheme.
- The verifier $\mathcal{V}$ picks a random $r \in \mathbb{F}^{\log m}$ and sends r to $\mathcal{P}$. The verifier makes one evaluation query to $\widetilde{a}$, to learn $\widetilde{a}(r)$.
- $\mathcal{P}$ and $\mathcal{V}$ apply the sum-check protocol to the $(\log N)$-variate polynomial $g(j) := \widetilde{M}(r, j) \cdot \hat{t}(j)$, to confirm that
$$\widetilde{a}(r) = \sum_{j \in \{0,1\}^{\log N}} g(j).$$
- At the end of the sum-check protocol, the verifier needs to evaluate $\widetilde{M}(r, r')$ and $\hat{t}(r')$ for a random point $r' \in \mathbb{F}^{\log N}$ that is chosen entry-by-entry over the course of the sum-check protocol. This costs one evaluation query to $\widetilde{M}$ and one to $\hat{t}$.

Figure 4: Description of our lookup argument using an extension polynomial $\hat{t}$ of the table vector t, where $\hat{t}$ may not be multilinear. A polynomial commitment to the multilinear polynomial $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$ is given to the verifier as input.

5 Spark: Sparse multilinear polynomial commitment

We prove a substantial strengthening of a result from Spartan [Set20, Lemma 7.6]. In particular, we show that in Spartan's sparse polynomial commitment scheme, Spark, one does not need to assume that certain metadata associated with a sparse polynomial is committed honestly. We thereby obtain the first "standard" polynomial commitment scheme (i.e., meeting Definition 2.2) with prover costs *linear* in the number of non-zero coefficients.

We prove this result without any substantive changes to Spark. For simplicity of presentation, we do make a minor change that does not affect costs nor analysis: we have the prover commit to metadata associated with the sparse polynomial at the time of proving an evaluation rather than when the prover commits to the sparse polynomial (the metadata depends only on the sparse polynomial, and in particular, it is independent of the point at which the sparse polynomial is evaluation, so the metadata can be committed either in the commit phase or when proving an evaluation). Our text below is adapted from an exposition of Spartan's result by Golovnev et al. [GLS+21]. The reader should conceptualize the sparse polynomial commitment scheme below as a bespoke SNARK allowing the prover to prove it correctly ran the sparse $(\log N)$-variate multilinear polynomial evaluation algorithm sketched in Section 3.1 using c memories of size $N^{1/c}$.

A (slightly) simpler result: $c = 2$. We begin by stating and proving a special case of our final result, the proof of which exhibits all of the ideas and techniques. This special case (Theorem 4) describes a transformation from any commitment scheme for dense polynomials defined over $\log m$ variables to one for sparse multilinear polynomials defined over $\log N = 2 \log m$ variables. It is the bespoke SNARK mentioned above when using $c = 2$ memories of size $N^{1/2}$.

The dominant costs for the prover in the sparse polynomial commitment scheme is committing to 7 dense multilinear polynomials over $\log(m)$-many variables, and 2 dense multilinear polynomials over $\log(N^{1/c})$-many variables.

In dense ℓ-variate multilinear polynomial commitment schemes, the prover time is roughly linear in 2^{ℓ}. Hence, so long as $m \geq N^{1/c}$, the prover time is dominated by the commitments to the 7 dense polynomials over $\log(m)$-many variables. This ensures that the prover time is linear in the sparsity of the committed polynomial as desired (rather than linear in $2^{2 \log m} = m^2$, which would be the runtime of applying a dense polynomial commitment scheme directly to the sparse polynomial over $2 \log m$ variables).

The full result. If we wish to commit to a sparse multilinear polynomial over ℓ variables, let $N := 2^{\ell}$ denote the dimensionality of the space of ℓ-variate multilinear polynomials. For any desired integer $c \geq 2$, our final, general, result replaces these two memories (each of size equal to $N^{1/2}$) with c memories of size equal to $N^{1/c}$. Ultimately, the prover must commit to $(3c + 1)$ many dense $(\log m)$-variate multilinear polynomials, and c many dense $(\log(N^{1/c}))$-variate polynomials.

We begin with the simpler result where c equals 2 before stating and proving the full result.

Theorem 4 (Special case of Theorem 5 with $c = 2$)**.** *Let $\mathsf{M} = N^{1/2}$. Given a polynomial commitment scheme for $(\log \mathsf{M})$-variate multilinear polynomials with the following parameters (where M is a positive integer and WLOG a power of 2):*

- *the size of the commitment is $\mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $\mathsf{tc}(\mathsf{M})$;*
- *the running time of the prover to prove a polynomial evaluation is $\mathsf{tp}(\mathsf{M})$;*
- *the running time of the verifier to verify a polynomial evaluation is $\mathsf{tv}(\mathsf{M})$;*
- *the proof size is $\mathsf{p}(\mathsf{M})$,*

there exists a polynomial commitment scheme for multilinear polynomials over $2 \log \mathsf{M} = \log N$ variables that evaluate to a non-zero value at at most m locations over the Boolean hypercube $\{0,1\}^{2 \log \mathsf{M}}$, with the following parameters:

- *the size of the commitment is $7\mathsf{c}(m) + 2\mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $O(\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))$;*
- *the running time of the prover to prove a polynomial evaluation is $O(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M}))$;*
- *the running time of the verifier to verify a polynomial evaluation is $O(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M}))$; and*
- *the proof size is $O(\mathsf{p}(m) + \mathsf{p}(\mathsf{M}))$.*

Representing sparse polynomials with dense polynomials. Let D denote a $(2\log \mathsf{M})$-variate multilinear polynomial that evaluates to a non-zero value at at most m locations over $\{0,1\}^{2\log \mathsf{M}}$. For any $r \in \mathbb{F}^{2\log \mathsf{M}}$, we can express the evaluation of $D(r)$ as follows. Interpret $r \in \mathbb{F}^{2\log \mathsf{M}}$ as a tuple (r_x, r_y) in a natural manner, where $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$. Then by multilinear Lagrange interpolation (Lemma 1), we can write

$$D(r_x, r_y) = \sum_{(i,j)\in\{0,1\}^{\log \mathsf{M}}\times\{0,1\}^{\log \mathsf{M}}:\ D(i,j)\neq 0} D(i,j)\cdot \widetilde{eq}(i, r_x)\cdot \widetilde{eq}(j, r_y). \tag{19}$$

Claim 1. *Let* to-field *be the canonical injection from* $\{0,1\}^{\log \mathsf{M}}$ *to* $\mathbb{F}$ *and* to-bits *be its inverse. Given a* $2\log \mathsf{M}$*-variate multilinear polynomial* D *that evaluates to a non-zero value at at most* m *locations over* $\{0,1\}^{2\log \mathsf{M}}$*, there exist three* $(\log m)$*-variate multilinear polynomials* $\mathsf{row}, \mathsf{col}, \mathsf{val}$ *such that the following holds for all* $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$*.*

$$D(r_x, r_y) = \sum_{k\in\{0,1\}^{\log m}} \mathsf{val}(k)\cdot \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)\cdot \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y). \tag{20}$$

Moreover, the polynomials' coefficients in the Lagrange basis can be computed in $O(m)$ time.

Proof. Since D evaluates to a non-zero value at at most m locations over $\{0,1\}^{2\log \mathsf{M}}$, D can be represented uniquely with m tuples of the form $(i, j, D(i,j)) \in (\{0,1\}^{\log \mathsf{M}}, \{0,1\}^{\log \mathsf{M}}, \mathbb{F})$. By using the natural injection to-field from $\{0,1\}^{\log \mathsf{M}}$ to $\mathbb{F}$, we can view the first two entries in each of these tuples as elements of $\mathbb{F}$ (let to-bits denote its inverse). Furthermore, these tuples can be represented with three m-sized vectors $R, C, V \in \mathbb{F}^m$, where tuple k (for all $k \in [m]$) is stored across the three vectors at the kth location in the vector, i.e., the first entry in the tuple is stored in R, the second entry in C, and the third entry in V. Take row as the unique MLE of R viewed as a function $\{0,1\}^{\log m} \to \mathbb{F}$. Similarly, col is the unique MLE of C, and val is the unique MLE of V. The claim holds by inspection since Equations (19) and (20) are both multilinear polynomials in r_x and r_y and agree with each other at every pair $r_x, r_y \in \{0,1\}^{\log \mathsf{M}}$. □

Conceptually, the sum in Equation (20) is *exactly* what the sparse polynomial evaluation algorithm described in Section 3.1 computes term-by-term. Specifically, that algorithm (using $c = 2$ memories) filled up one memory with the quantities $\widetilde{eq}(i, r_x)$ as i ranges over $\{0,1\}^{\log \mathsf{M}}$ (see Equation (19), and the other memory with the quantities $\widetilde{eq}(j, r_x)$, and then computed each term of Equation (20) via one lookup into each memory, to the respective memory cells with (binary) indices $\mathsf{to\text{-}bits}(\mathsf{row}(k))$ and $\mathsf{to\text{-}bits}(\mathsf{col}(k))$, followed by two field multiplications.

Commit phase. To commit to D, the committer can send commitments to the three $(\log m)$-variate multilinear polynomials $\mathsf{row}, \mathsf{col}, \mathsf{val}$ from Claim 1. Using the provided polynomial commitment scheme, this costs $O(m)$ finite field operations, and the size of the commitment to D is $O_\lambda(\mathsf{c}(m))$.

Intuitively, the commit phase commits to a "densified" representation of the sparse polynomial, which simply lists all the Lagrange basis polynomial with non-zero coefficients (each specified as an element in $\{0, \ldots, \mathsf{M}-1\}^2$), along with the associated coefficient. This is exactly the input to the sparse polynomial evaluation algorithm described in Section 3.1.

In the evaluation phase described below, the prover proves that it correctly ran the sparse polynomial evaluation algorithm sketched in Section 3.1 on the committed polynomial in order to evaluate it at the requested evaluation point $(r_x, r_y) \in \mathbb{F}^{2\log \mathsf{M}}$.

A first attempt at the evaluation phase. Given $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$, to prove an evaluation of a committed polynomial, i.e., to prove that $D(r_x, r_y) = v$ for a purported evaluation $v \in \mathbb{F}$, consider the polynomial IOP in Figure 5, where the polynomial IOP assumes that the verifier has oracle access to the three $(\log m)$-variate multilinear polynomial oracles that encode D (namely $\mathsf{row}, \mathsf{col}, \mathsf{val}$).

1. $\mathcal{P} \to \mathcal{V}$: two $(\log m)$-variate multilinear polynomials E_{rx} and E_{ry} as oracles. These polynomials are purported to respectively equal the multilinear extensions of the functions mapping $k \in \{0,1\}^{\log m}$ to $\widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$ and $\widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.
2. $\mathcal{V} \leftrightarrow \mathcal{P}$: run the sum-check reduction to reduce the check that

   $$v = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$$

   to checking if the following hold, where $r_z \in \mathbb{F}^{\log m}$ is chosen at random by the verifier over the course of the sum-check protocol:
   - $\mathsf{val}(r_z) \stackrel{?}{=} v_{\mathsf{val}}$;
   - $E_{\mathsf{rx}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$ and $E_{\mathsf{ry}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$. Here, v_{val}, $v_{E_{\mathsf{rx}}}$, and $v_{E_{\mathsf{ry}}}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the three equalities hold with an oracle query to each of $\mathsf{val}, E_{\mathsf{rx}}, E_{\mathsf{ry}}$.

Figure 5: A first attempt at a polynomial IOP for revealing a requested evaluation of a $(2\log(\mathsf{M}))$-variate multilinear polynomial p over $\mathbb{F}$ such that $p(x) \neq 0$ for at most m values of $x \in \{0,1\}^{2\log(\mathsf{M})}$.

Here, the oracles E_{rx} and E_{ry} should be thought of as the (purported) multilinear extensions of the values returned by each memory reads that the algorithm of Section 3.1 performed into each of its two memories, step-by-step over the course of its execution.

If the prover is honest, it is easy to see that it can convince the verifier about the correct of evaluations of D. Unfortunately, the two oracles that the prover sends in the first step of the depicted polynomial IOP can be completely arbitrary. To fix, this, $\mathcal{V}$ must *additionally* check that the following two conditions hold.

- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$; and
- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.

A core insight of Spartan [Set20] is to check these two conditions using memory-checking techniques [BEG$^+$91]. These techniques amount to an efficient randomized procedure to confirm that every memory read over the course of an algorithm's execution returns the value last written to that location.

We take a detour to introduce new results that we rely on here.

Detour: A new variant of offline memory checking. Recall that in the offline memory checking algorithm of [BEG$^+$91], a *trusted checker* issues operations to an untrusted memory. For our purposes, it suffices to consider only operation sequences in which each memory address is initialized to a certain value, and all subsequent operations are read operations. To enable efficient checking using multiset-fingerprinting techniques, the memory is modified so that in addition to storing a value at each address, the memory also stores a timestamp with each address. Moreover, each read operation is followed by a write operation that updates the timestamp associated with that address (but not the value stored there).

In prior descriptions of offline memory checking [BEG$^+$91, CDD$^+$03, SAGL18], the trusted checker maintains a single timestamp counter and uses it to compute write timestamps, whereas in the description below, the trusted checker does not use any local timestamp counter; rather, each memory cell maintains its own counter, which is incremented by the checker every time the cell is read.[16] For this reason, we depart from the standard terminology in the memory-checking literature and henceforth refer to these quantities as *counters* rather than timestamps.

[16] The same timestamp update procedure was used in Spartan [Set20, §7.2.3], but to achieve a concrete efficiency benefit. In particular, Spartan used a separate timestamp counter for each cell and considered the case where all read timestamps were guaranteed to be computed honestly. In this case, the write timestamp is the result of incrementing an honestly returned read timestamp, which allows Spartan to not explicitly materialize write timestamps. Here, we are interested in the case where read timestamps themselves are not computed honestly.

The memory-checking procedure is captured in the codebox below.

Local state of the checker: Two sets: RS and WS, which are initialized as follows.[17] $\mathsf{RS} = \{\}$, and for an M-sized memory, WS is initialized to the following set of tuples: for all $i \in [N^{1/c}]$, the tuple $(i, v_i, 0)$ is included in WS, where v_i is the value stored at address i, and the third entry in the tuple, 0, is an "initial count" associated with the value (intuitively capturing the notion that when v_i was written to address i, it was the first time that address was accessed). Here, $[\mathsf{M}]$ denotes the set $\{0, 1, \ldots, \mathsf{M} - 1\}$.

Read operations and an invariant. For a read operation at address a, suppose that the untrusted memory responds with a value-count pair (v, t). Then the checker updates its local state as follows:

1. $\mathsf{RS} \leftarrow \mathsf{RS} \cup \{(a, v, t)\}$;
2. store $(v, t + 1)$ at address a in the untrusted memory; and
3. $\mathsf{WS} \leftarrow \mathsf{WS} \cup \{(a, v, t + 1)\}$.

The following claim captures the invariant maintained on the sets of the checker:

Claim 2. *Let $\mathbb{F}$ be a prime order field. Assuming that the domain of counts is $\mathbb{F}$ and that m (the number of reads issued) is smaller than $|\mathbb{F}|$. Let WS and RS denote the multisets maintained by the checker in the above algorithm at the conclusion of m read operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set S with cardinality M consisting of tuples of the form (k, v_k, t_k) for all $k \in [\mathsf{M}]$ such that $\mathsf{WS} = \mathsf{RS} \cup S$. Moreover, S is computable in time linear in M.*

Conversely, if the untrusted memory ever returns a value v for a memory call $k \in [\mathsf{M}]$ such v does not equal the value initially written to cell k, then there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.

Proof. If for every read operation, the untrusted memory returns the tuple last written to that location, then it is easy to see the existence of the desired set S. It is simply the current state of the untrusted memory viewed as the set of address-value-count tuples.

We now prove the other direction in the claim. For notational convenience, let WS_i and RS_i $(0 \leq i \leq m)$ denote the multisets maintained by the trusted checker at the conclusion of the ith read operation (i.e., WS_0 and RS_0 denote the multisets before any read operation is issued). Suppose that there is some read operation i that reads from address k, and the untrusted memory responds with a tuple (v, t) such that v differs from the value initially written to address k. This ensures that $(k, v, t) \in \mathsf{RS}_j$ for all $j \geq i$, and in particular that $(k, v, t) \in \mathsf{RS}$, where recall that RS is the read set at the conclusion of the m read operations. Hence, to ensure that there exists a set S such that $\mathsf{RS} \cup S = \mathsf{WS}$ at the conclusion of the procedure (i.e., to ensure that $\mathsf{RS} \subseteq \mathsf{WS}$), there must be some other read operation during which address k is read, and the untrusted memory returns tuple $(k, v, t - 1)$.[18] This is because the only way that the checker writes (k, v, t) to memory is if a read to address k returns tuple $(v, t - 1)$.

Accordingly, the same reasoning as above applies to tuple $(k, v, t - 1)$. That is, to ensure that $\mathsf{RS} = \mathsf{WS}$ at the conclusion of the procedure, there must be some other read operation at which address k is read, and the untrusted memory returns tuple $(k, v, t - 2)$. And so on. We conclude that for *every* field element t' in $\mathbb{F}$, there is some read operation that returns (k, v, t'). Since there are m many read operations and the size of field is greater than m, we obtain a contradiction. □

Remark 5. *The proof of Claim 2 implies that, if the checker ever performs a read to an "invalid" memory cell k, meaning a cell indexed by $k \notin [\mathsf{M}]$, then regardless of the value and timestamp returned by the untrusted prover in response to that read, there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.*

[17] The checker in [BEG$^+$91] maintains a fingerprint of these sets, but for our exposition, we let the checker maintain full sets.

[18] Recall here that counter arithmetic is done over $\mathbb{F}$, i.e., t and $t - 1$ are in $\mathbb{F}$.

Counter polynomials. To aid the polynomial evaluation proof of the sparse polynomial the prover commits to additional multilinear polynomials beyond E_{rx} and E_{ry}. We now describe these additional polynomials and how they are constructed.

Observe that given the size M of memory and a list of m addresses involved in read operations, one can compute two vectors $C_r \in \mathbb{F}^m, C_f \in \mathbb{F}^{\mathsf{M}}$ defined as follows. For $k \in [m]$, $C_r[k]$ stores the count that would have been returned by the untrusted memory if it were honest during the kth read operation. Similarly, for $j \in [\mathsf{M}]$, let $C_f[j]$ store the final count stored at memory location j of the untrusted memory (if the untrusted memory were honest) at the termination of the m read operations. Computing these three vectors requires computation comparable to $O(m)$ operations over $\mathbb{F}$.

Let $\mathsf{read\_counts} = \widetilde{C_r}, \mathsf{write\_counts} = \widetilde{C_r} + 1, \mathsf{final\_counts} = \widetilde{C_f}$. We refer to these polynomials as *counter polynomials*, which are unique for a given memory size M and a list of m addresses involved in read operations.

The actual evaluation proof. To prove the evaluation of a given a $(2\log \mathsf{M})$-variate multilinear polynomial D that evaluates to a non-zero value at at most m locations over $\{0,1\}^{2\log \mathsf{M}}$, the prover sends the following polynomials in addition to E_{rx} and E_{ry}: two $(\log m)$-variate multilinear polynomials as oracles $(\mathsf{read\_counts}_{\mathsf{row}}, \mathsf{read\_counts}_{\mathsf{col}})$, and two $(\log \mathsf{M})$-variate multilinear polynomials $(\mathsf{final\_counts}_{\mathsf{row}}, \mathsf{final\_counts}_{\mathsf{col}})$, where $(\mathsf{read\_counts}_{\mathsf{row}}, \mathsf{final\_counts}_{\mathsf{row}})$ and $(\mathsf{read\_counts}_{\mathsf{col}}, \mathsf{final\_counts}_{\mathsf{col}})$ are respectively the counter polynomials for the m addresses specified by row and col over a memory of size M.

After that, in addition to performing the polynomial IOP depicted earlier in the proof (Figure 5), the core idea is to check if the two oracles sent by the prover satisfy the conditions identified earlier using Claim 2.

Claim 3. *Given a* $(2\log \mathsf{M})$*-variate multilinear polynomial, suppose that* $(\mathsf{row}, \mathsf{col}, \mathsf{val})$ *denote multilinear polynomials committed by the commit algorithm. Furthermore, suppose that*

$$(E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_counts}_{\mathsf{row}}, \mathsf{final\_counts}_{\mathsf{row}}, \mathsf{read\_counts}_{\mathsf{col}}, \mathsf{final\_counts}_{\mathsf{col}})$$

denote the additional polynomials sent by the prover at the beginning of the evaluation proof.

For any $r_x \in \mathbb{F}^{\log \mathsf{M}}$*, suppose that*

$$\forall k \in \{0,1\}^{\log m},\ E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x). \tag{21}$$

Then the following holds: $\mathsf{WS} = \mathsf{RS} \cup S$*, where*

- $\mathsf{WS} = \{(\mathsf{to\text{-}field}(i), \widetilde{eq}(i, r_x), 0) \colon i \in \{0,1\}^{\log(\mathsf{M})}\} \cup \{(\mathsf{row}(k), E_{\mathsf{rx}}(k), \mathsf{write\_counts}_{\mathsf{row}}(k) = \mathsf{read\_counts}_{\mathsf{row}}(k) + 1) \colon k \in \{0,1\}^{\log m}\}$*;*
- $\mathsf{RS} = \{(\mathsf{row}(k), E_{\mathsf{rx}}(k), \mathsf{read\_counts}_{\mathsf{row}}(k)) \colon k \in \{0,1\}^{\log m}\}$*; and*
- $S = \{(\mathsf{to\text{-}field}(i), \widetilde{eq}(i, r_x), \mathsf{final\_counts}_{\mathsf{row}}(i)) \colon i \in \{0,1\}^{\log(\mathsf{M})}\}$.

Meanwhile, if Equation (21) *does not hold, then there is no set* S *such that* $\mathsf{WS} = \mathsf{RS} \cup S$*, where* WS *and* RS *are defined as above.*

Similarly, for any $r_y \in \mathbb{F}^{\log \mathsf{M}}$*, checking that* $\forall k \in \{0,1\}^{\log m}$*,* $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$ *is equivalent (in the sense above) to checking that* $\mathsf{WS}' = \mathsf{RS}' \cup S'$*, where*

- $\mathsf{WS}' = \{(\mathsf{to\text{-}field}(j), \widetilde{eq}(j, r_y), 0) \colon j \in \{0,1\}^{\log(\mathsf{M})}\} \cup \{(\mathsf{col}(k), E_{\mathsf{ry}}(k), \mathsf{write\_counts}_{\mathsf{col}}(k) = \mathsf{read\_counts}_{\mathsf{col}}(k) + 1) \colon k \in \{0,1\}^{\log m}\}$*;*
- $\mathsf{RS}' = \{(\mathsf{col}(k), E_{\mathsf{ry}}(k), \mathsf{read\_counts}_{\mathsf{col}}(k)) \colon k \in \{0,1\}^{\log m}\}$*; and*
- $S' = \{(\mathsf{to\text{-}field}(j), \widetilde{eq}(j, r_y), \mathsf{final\_counts}_{\mathsf{col}}(i)) \colon j \in \{0,1\}^{\log(\mathsf{M})}\}$.

Proof. The result follows from an application of the invariant in Claim 2.

Here, we clarify the following subtlety. The expression $\mathsf{to\text{-}bits}(\mathsf{row}(k))$ appearing in Equation (21) is not defined if $\mathsf{row}(k)$ is outside of $[\mathsf{M}]$ for any $k \in \{0,1\}^{\log m}$. But in this event, Remark 5 nonetheless implies the conclusion of the theorem, namely that there is no set S such that $\mathsf{WS} = \mathsf{RS} \cup S$. The analogous conclusion holds by the same reasoning if $\mathsf{col}(k)$ is outside of $[\mathsf{M}]$ for any $k \in \{0,1\}^{\log m}$. □

There is no direct way to prove that the checks on sets in Claim 3 hold. Instead, we rely on public-coin, multiset hash functions to compress RS, WS, and S into a single element of $\mathbb{F}$ each. Specifically:

Claim 4 ([Set20]). *Given two multisets A, B where each element is from $\mathbb{F}^3$, checking that $A = B$ is equivalent to checking the following, except for a soundness error of $O(|A| + |B|)/|\mathbb{F}|)$ over the choice of γ, τ: $\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$, where $\mathcal{H}_{\tau,\gamma}(A) = \prod_{(a,v,t)\in A} (h_\gamma(a, v, t) - \tau)$, and $h_\gamma(a, v, t) = a \cdot \gamma^2 + v \cdot \gamma + t$. That is, if $A = B$, $\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$ with probability 1 over randomly chosen values τ and γ in $\mathbb{F}$, while if $A \neq B$, then $\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$ with probability at most $O(|A| + |B|)/|\mathbb{F}|)$.*

Intuitively, Claim 4 gives an efficient randomized procedure for checking whether two sequences of tuples are permutations of each other. First, the procedure Reed-Solomon fingerprints each tuple (see [Tha22, Section 2.1] for an exposition). This is captured by the function h_γ and intuitively replaces each tuple with a single field element, such that distinct tuples are unlikely to collide. Second, the procedure applies a permutation-independent fingerprinting procedure $H_{r,\gamma}$ to confirm that the resulting two sequences of fingerprints are permutations of each other.

We are now ready to depict a polynomial IOP for proving evaluations of a committed sparse multilinear polynomial. Given $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$, to prove that $D(r_x, r_y) = v$ for a purported evaluation $v \in \mathbb{F}$, consider the polynomial IOP given in Figure 6, which assumes that the verifier has an oracle access to multilinear polynomial oracles that encode D (namely, $\mathsf{row}, \mathsf{col}, \mathsf{val}$)

Completeness. Perfect completeness follows from perfect completeness of the sum-check protocol and the fact that the multiset equality checks using their fingerprints hold with probability 1 over the choice of τ, γ if the prover is honest.

Soundness. Applying a standard union bound to the soundness error introduced by probabilistic multiset equality checks with the soundness error of the sum-check protocol [LFKN90], we conclude that the soundness error for the depicted polynomial IOP as at most $O(m)/|\mathbb{F}|$.

Round and communication complexity. There are three invocations of the sum-check protocol. First, the sum-check protocol is applied on a polynomial with $\log m$ variables where the degree is at most 3 in each variable, so the round complexity is $O(\log m)$ and the communication cost is $O(\log m)$ field elements. Second, four sum-check-based "grand product" protocols are computed in parallel. Two of the grand products are over vectors of size M and the remaining two are over vectors of size m. Third, the depicted IOP runs four additional "grand products", which incurs the same costs as above. In total, with the protocol of [SL20, Section 6] for grand products, the round complexity of the depicted IOP is $\tilde{O}(\log m + \log(N))$ and the communication cost is $\tilde{O}(\log m + \log N)$ field elements, where the $\tilde{O}$ notation hides doubly-logarithmic factors. The prover commits to an extra $O(m/\log^3 m)$ field elements.

Verifier time. The verifier's runtime is dominated by its runtime in the grand product sum-check reductions, which is $\tilde{O}(\log m)$ field operations.

Prover Time. Using linear-time sum-checks [Tha13] in all three sum-check reductions (and using the linear-time prover in the grand product protocol [Tha13, SL20]), the prover's time is $O(N)$ finite field operations for unstructured tables. For appropriately structured tables, we explain in Section 6 how to implement the prover in $O(cm)$ finite field operations.

Finally, to prove Theorem 4, applying the compiler of [BFS20] to the depicted polynomial IOP with the given polynomial commitment primitive, followed by the Fiat-Shamir transformation [FS86], provides the desired non-interactive argument of knowledge for proving evaluations of committed sparse multilinear polynomials, with efficiency claimed in the theorem statement.

Appendix C provides additional details of the grand product argument.

//During the commit phase, $\mathcal{P}$ has committed to three $(\log m)$-variate multilinear polynomials $\mathsf{row}, \mathsf{col}, \mathsf{val}$.

1. $\mathcal{P} \to \mathcal{V}$: four $(\log m)$-variate multilinear polynomials $E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_counts}_{\mathsf{row}}, \mathsf{read\_counts}_{\mathsf{col}}$ and two $(\log \mathsf{M})$-variate multilinear polynomials $\mathsf{final\_counts}_{\mathsf{row}}$, $\mathsf{final\_counts}_{\mathsf{col}}$.
2. Recall that Claim 1 (see Equation (20)) shows that $D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$ assuming that
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$; and
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.

   Hence, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $\mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $\mathsf{val}(r_z) \overset{?}{=} v_{\mathsf{val}}$; and
   - $E_{\mathsf{rx}}(r_z) \overset{?}{=} v_{E_{\mathsf{rx}}}$ and $E_{\mathsf{ry}}(r_z) \overset{?}{=} v_{E_{\mathsf{ry}}}$. Here, v_{val}, $v_{E_{\mathsf{rx}}}$ and $v_{E_{\mathsf{ry}}}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the three equalities above hold with one oracle query each to each of $\mathsf{val}, E_{\mathsf{rx}}, E_{\mathsf{ry}}$.
4. // The following checks if E_{rx} is well-formed as per the first bullet in Step 2 above.
5. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
6. $\mathcal{V} \leftrightarrow \mathcal{P}$: run a sum-check-based protocol for "grand products" ([Tha13, Proposition 2] or [SL20, Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r_{\mathsf{M}} \in \mathbb{F}^{\log \mathsf{M}}, r_m \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $\widetilde{eq}(r_{\mathsf{M}}, r_x) \overset{?}{=} v_{eq}$
   - $E_{\mathsf{rx}}(r_m) \overset{?}{=} v_{E_{\mathsf{rx}}}$
   - $\mathsf{row}(r_m) \overset{?}{=} v_{\mathsf{row}}$; $\mathsf{read\_counts}_{\mathsf{row}}(r_m) \overset{?}{=} v_{\mathsf{read\_counts}_{\mathsf{row}}}$; and $\mathsf{final\_counts}_{\mathsf{row}}(r_{\mathsf{M}}) \overset{?}{=} v_{\mathsf{final\_counts}_{\mathsf{row}}}$
7. $\mathcal{V}$: directly check if the first equality holds, which can be done with $O(\log \mathsf{M})$ field operations; check the remaining equations hold with an oracle query to each of $E_{\mathsf{rx}}, \mathsf{row}, \mathsf{read\_counts}_{\mathsf{row}}, \mathsf{final\_counts}_{\mathsf{row}}$.
8. // The following steps check if E_{ry} is well-formed as per the second bullet in Step 2 above.
9. $\mathcal{V} \to \mathcal{P}$: $\tau', \gamma' \in_R \mathbb{F}$.
10. $\mathcal{V} \leftrightarrow \mathcal{P}$: run a sum-check-based reduction for "grand products" ([Tha13, Proposition2] or [SL20, Sections 5 and 6]) to reduce the check that $\mathcal{H}_{\tau',\gamma'}(\mathsf{WS}') = \mathcal{H}_{\tau',\gamma'}(\mathsf{RS}') \cdot \mathcal{H}_{\tau',\gamma'}(S')$, where $\mathsf{RS}', \mathsf{WS}', S'$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r'_{\mathsf{M}} \in \mathbb{F}^{\log \mathsf{M}}, r'_m \in \mathbb{F}^{\log m}$ are chosen at random by the verifier in the sum-check protocol:
    - $\widetilde{eq}(r'_{\mathsf{M}}, r_y) \overset{?}{=} v'_{eq}$
    - $E_{\mathsf{ry}}(r'_m) \overset{?}{=} v_{E_{\mathsf{ry}}}$
    - $\mathsf{col}(r'_m) \overset{?}{=} v_{\mathsf{col}}$; $\mathsf{read\_counts}_{\mathsf{col}}(r'_m) \overset{?}{=} v_{\mathsf{read\_counts}_{\mathsf{col}}}$; and $\mathsf{final\_counts}_{\mathsf{col}}(r'_{\mathsf{M}}) \overset{?}{=} v_{\mathsf{final\_counts}_{\mathsf{col}}}$
11. $\mathcal{V}$: directly check if the first equality holds, which can be done with $O(\log \mathsf{M})$ field operations; check the remaining equations hold with an oracle query to each of $E_{\mathsf{ry}}, \mathsf{col}, \mathsf{read\_counts}_{\mathsf{col}}, \mathsf{final\_counts}_{\mathsf{col}}$.

Figure 6: Evaluation procedure of our sparse polynomial commitment scheme.

Additional discussion and intuition. As previously discussed, the protocol in Figure 6 allows the prover to prove that it correctly ran the sparse polynomial evaluation algorithm described in Section 3.1 on the committed representation of the sparse polynomial. The core of the protocol lies in the memory-checking procedure, which enables the untrusted prover to establish that it produced the correct value upon every one of the algorithm's reads into the $c = 2$ memories of size $\mathsf{M} = N^{1/2}$. Intuitively, the values that the prover cryptographically commits to in the protocol are simply the values and counters returned by the aforementioned read operations (including a final "read pass" over both memories, which is required by the memory-checking procedure).

A key and subtle aspect of the above is that the prover does *not* have to cryptographically commit to the values written to memory in the algorithm's first phase, when it initializes the two memories (aka lookup tables, albeit dynamically determined by the evaluation point (r_x, r_y)), of size $\mathsf{M} = N^{1/2}$. This is because these lookup tables are structured, meaning that the verifier can evaluate the multilinear extension of these tables on its own. The whole point of cryptographically committing to these values is to let the verifier evaluate the multilinear extension thereof at a randomly chosen point in the grand product argument. Since the verifier can perform this evaluation quickly on its own, there is no need for the prover in the protocol of Figure 6 to commit to these values.

5.1 The general result

Theorem 4 gives a commitment scheme for m-sparse multilinear polynomials over $\log N = 2\log(\mathsf{M})$ many variables, in which the prover commits to 7 dense multilinear polynomials over $\log m$ many variables, and 2 dense polynomials over $\log(\mathsf{M})$ many variables.

Suppose we want to support sparse polynomials over $c\log(\mathsf{M})$ variables for constant $c > 2$, while ensuring that the prover still only commits to $3c + 1$ many dense multilinear polynomials over $\log m$ many variables, and c many over $\log(N^{1/c})$ many variables. We can proceed as follows.

The function eq and its tensor structure. Recall that $\mathsf{eq}_s \colon \{0,1\}^s \times \{0,1\}^s \to \{0,1\}$ takes as input two vectors of length s and outputs 1 if and only if the vectors are equal. (In this section, we find it convenient to make explicit the number of variables over which eq is defined by including a subscript s.) Recall from Equation (4) that $\widetilde{\mathsf{eq}}_s(x, e) = \prod_{i=1}^{s} \left(x_i e_i + (1 - x_i)(1 - e_i)\right)$.

Equation (19) expressed the evaluation $\widetilde{D}(r_x, r_y)$ of a sparse $2\log(\mathsf{M})$-variate multilinear polynomial $\widetilde{D}$ as

$$\widetilde{D}(r_x, r_y) = \sum_{(i,j)\in\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}} D(i,j)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y). \tag{22}$$

The last two factors on the right hand side above have effectively factored $\widetilde{\mathsf{eq}}_{2\log(\mathsf{M})}\left((i,j),(r_x,r_y)\right)$ as the product of two terms that each test equality over $\log(\mathsf{M})$ many variables, namely:

$$\widetilde{\mathsf{eq}}_{2\log(\mathsf{M})}\left((i,j),(r_x,r_y)\right) = \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y).$$

Within the sparse polynomial commitment scheme, this ultimately led to checking two different memories, each of size M, one of which we referred to as the "row" memory, and one as the "column" memory. For each memory checked, the prover had to commit to three $(\log m)$-variate polynomials, e.g., E_{rx}, row, $\mathsf{read\_counts}_{\mathsf{row}}$, and one $\log(\mathsf{M})$-variate polynomial, e.g., $\mathsf{final\_counts}_{\mathsf{row}}$.

Supporting $\log N = c\log M$ variables rather than $2\log M$. If we want to support polynomials over $c\log(\mathsf{M})$ variables for $c > 2$, we simply factor $\widetilde{\mathsf{eq}}_{c\log(\mathsf{M})}$ into a product of c terms that test equality over $\log(\mathsf{M})$ variables each. For example, if $c = 3$, then we can write:

$$\widetilde{\mathsf{eq}}_{3\log(\mathsf{M})}\left((i,j,k),(r_x,r_y,r_z)\right) = \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k, r_z).$$

Hence, if D is a $(3\log M)$-variate polynomial, we obtain the following analog of Equation (22):

$$\widetilde{D}(r_x, r_y, r_z) = \sum_{(i,j,k) \in \{0,1\}^{\log(\mathsf{M})} \times \{0,1\}^{\log(\mathsf{M})} \times \{0,1\}^{\log(\mathsf{M})}} D(i,j,k) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k, r_z). \tag{23}$$

Based on the above equation, straightforward modifications to the sparse polynomial commitment scheme lead to checking c different untrusted memories, each of size M, rather than two. For example, when $c = 3$, the first memory stores all evaluations of $\widetilde{eq}_{\log(\mathsf{M})}(i, r_x)$ as i ranges over $\{0,1\}^{\log m}$, the second stores $\widetilde{eq}_{\log(\mathsf{M})}(j, r_y)$ as j ranges over $\{0,1\}^{\log m}$, and the third stores $\widetilde{eq}_{\log(\mathsf{M})}(k, r_z)$ as k ranges over $\{0,1\}^{\log m}$. These are exactly the contents of the three lookup tables of size $N^{1/c}$ used by the sparse polynomial evaluation algorithm of Section 3.1 when $c = 3$.

For each memory checked, the prover has to commit to three multilinear polynomials defined over $\log(m)$-many variables, and one defined over $\log(\mathsf{M}) = \log(N)/c$ variables. We obtain the following theorem.

Theorem 5. *Given a polynomial commitment scheme for $(\log \mathsf{M})$-variate multilinear polynomials with the following parameters (where M is a positive integer and WLOG a power of 2):*

- *the size of the commitment is $\mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $\mathsf{tc}(\mathsf{M})$;*
- *the running time of the prover to prove a polynomial evaluation is $\mathsf{tp}(\mathsf{M})$;*
- *the running time of the verifier to verify a polynomial evaluation is $\mathsf{tv}(\mathsf{M})$;*
- *the proof size is $\mathsf{p}(\mathsf{M})$,*

there exists a polynomial commitment scheme for $(c \log \mathsf{M})$-variate multilinear polynomials that evaluate to a non-zero value at at most m locations over the Boolean hypercube $\{0,1\}^{c \log \mathsf{M}}$, with the following parameters:

- *the size of the commitment is $(3c+1)\mathsf{c}(m) + c \cdot \mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $O\left(c \cdot (\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))\right)$;*
- *the running time of the prover to prove a polynomial evaluation is $O\left(c\left(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M})\right)\right)$;*
- *the running time of the verifier to verify a polynomial evaluation is $O\left(c\left(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M})\right)\right)$;*
- *the proof size is $O\left(c\left(\mathsf{p}(m) + \mathsf{p}(\mathsf{M})\right)\right)$.*

Many polynomial commitment schemes have efficient batching properties for evaluation proofs. For such schemes, the factor c can be omitted in the final three bullet points of Theorem 5 (i.e., prover and verifier costs for verifying polynomial evaluation do not grow with c).

5.2 Specializing the sparse commitment scheme to Lasso

Recall that in Lasso, if the prover is honest then the sparse polynomial commitment scheme is applied to the multilinear extension of a matrix M with m rows and N columns, where m is the number of lookups and N is the size of the table. If the prover is honest then each row of M is a unit vector.

In fact, we require the commitment scheme to enforce these properties even when the prover is potentially malicious. Achieving this simplifies the commitment scheme and provides concrete efficiency benefits. It also keeps Lasso's polynomial IOP simple as it does not need additional invocations of the sum-check protocol to prove that M satisfies these properties.

First, the multilinear polynomial $\mathsf{val}(k)$ is fixed to 1, and it is not committed by the prover. Recall from Claim 1 that $\mathsf{val}(k)$ extends the function that maps a bit-vector $k \in \{0,1\}^{\log m}$ to the value of the k'th non-zero evaluation of the sparse function. Since M is a $\{0,1\}$-valued matrix, $\mathsf{val}(k)$ is just the constant polynomial that evaluates to 1 at all inputs.

Second, for any $k = (k_1, \ldots, k_{\log m}) \in \{0,1\}^{\log m}$, the k'th non-zero entry of M is in row $\textsf{to-field}(k) = \sum_{j=1}^{\log m} 2^{j-1} \cdot k_j$. Hence, in Equation (20) of Claim 1, $\textsf{to-bits}(\textsf{row}(k))$ is simply k.[19] This means that $E_{\textsf{rx}}(k) = \widetilde{\textsf{eq}}(k, r_x)$, which the verifier can evaluate on its own in logarithmic time. With this fact in hand, the prover does not commit to $E_{\textsf{rx}}$ nor prove that it is well-formed.

In terms of costs in the resulting sparse polynomial commitment scheme applied to $\widetilde{M}$, this effectively removes the contribution of the first $\log m$ variables of $\widetilde{M}$ to the costs. Hence, the costs are that of applying the commitment scheme to an m-sparse $\log(N)$-variate polynomial (with $\textsf{val}$ fixed to 1).

This means that, setting $c = 2$ for illustration, the prover commits to 6 multilinear polynomials with $\log(m)$ variables each and to two multilinear polynomials with $(1/2)\log N$ variables each.

In general, the sparse polynomial commitment scheme used in Lasso to commit to $\widetilde{M}$ is described in Figure 7. The prover commits to $3c$ dense $(\log(m))$-variate multilinear polynomials, called $\dim_1, \ldots, \dim_c$ (the analogs of the row and col polynomials of Section 5), $E_1, \ldots, E_c$, and $\textsf{read\_counts}_1, \ldots, \textsf{read\_counts}_c$, as well as c dense multilinear polynomials in $\log(N^{1/c}) = \log(N)/c$ variables, called $\textsf{final\_counts}_1, \ldots, \textsf{final\_counts}_c$. Each $\dim_i$ is purported to be the memory cell from the i'th memory that the sparse polynomial evaluation algorithm of Section 3.1 reads at each of its m timesteps, $E_1, \ldots, E_c$ the values returned by those reads, and $\textsf{read\_counts}_1, \ldots, \textsf{read\_counts}_c$ the associated counts. $\textsf{final\_counts}_1, \ldots, \textsf{final\_counts}_c$ are purported to be to counts returned by the memory checking procedures final pass over each of the c memories.

If the prover is honest, then $\dim_1, \ldots, \dim_c$ each map $\{0,1\}^{\log m}$ to $\{0, \ldots, N^{1/c}-1\}$, and $\textsf{read\_counts}_1, \ldots, \textsf{read\_counts}_c$ each map $\{0,1\}^{\log m}$ to $\{0, \ldots, m-1\}$ and $\textsf{final\_counts}_1, \ldots, \textsf{final\_counts}_c$ each map $\{0,1\}^{\log m}$ to $\{0, \ldots, m-1\}$. In fact, for any integer $j > 0$, at most m/j out of the m evaluations of each counter polynomial $\textsf{read\_counts}_i$ and $\textsf{final\_counts}_i$ can be larger than j.

6 The sparse-dense sum-check protocol

Let $u \in \mathbb{F}^N$ be a vector with at most m non-zero entries and $t \in \mathbb{F}^N$ be another vector (which may have N non-zero entries). Let $\widetilde{u}$ and $\widetilde{t}$ denote their multilinear extensions. Throughout, we assume that there is some constant $C > 0$ such that $N \leq m^C$ (put another way, in our $O(m)$ prover time results, the Big-Oh notation hides a multiplicative factor that is linear in C).

6.1 Establishing that for any $r \in \mathbb{F}^{\log m}$, $\widetilde{M}(r, y)$ is m-sparse

In Lasso, $t \in \mathbb{F}^N$ will be the table (Definition 1.1), while $\widetilde{u}$ will be $\widetilde{M}(r, y)$. If the prover is honest, each row of M has exactly one non-zero entry, and accordingly $\widetilde{M}(r, y) \neq 0$ for at most m values of $y \in \{0,1\}^{\log N}$ by the following reasoning. Let

$$\chi_i(r) = \prod_{k=1}^{\log m} \left(r_k i_k + (1 - r_k)(1 - i_k)\right)$$

denote the i'th Lagrange basis polynomial, which maps i to 1 and maps all other inputs in $\{0,1\}^{\log m}$ to zero. Standard Lagrange interpolation for multilinear polynomials (see [Tha22, Chapter 3]) states that

$$\widetilde{M}(r, y) = \sum_{(i,j) \in \{0,1\}^{\log m + \log N}} M_{i,j} \cdot \chi_i(r) \cdot \chi_j(y). \tag{24}$$

Since for any i, $M_{i,j} \neq 0$ for exactly one j, the right hand side of Equation (24) can be non-zero for at most m values of $y \in \{0,1\}^{\log N}$, namely those y's indexing columns of M with at least one non-zero entry.

[19] More precisely, this holds if we define r_x to be in $\mathbb{F}^{\log m}$ and r_y to be in $\mathbb{F}^{\log N}$, rather than defining them both to be in $\mathbb{F}^{\log M} = \mathbb{F}^{(1/2)(\log m + \log n)}$.

//During the commit phase applied to the multilinear extension $\widetilde{M}$ of $m \times N$ matrix M with each row a unit vector, $\mathcal{P}$ has committed to c different ℓ-variate multilinear polynomials $\dim_1, \ldots, \dim_c$, where $\ell = \log(N^{1/c})$. These are analogs of the polynomials row and col from Figure 6. $\dim_i$ is purported to provide the indices of the cells of the i'th memory that are read by the sparse polynomial evaluation algorithm of Section 3.1. Note that these indices depend only on the the locations of the non-zero entries of $\widetilde{M}$.

//If $\mathcal{P}$ is honest, then each $\dim_i$ maps $\{0,1\}^{\log m}$ to $\{0, \ldots, N^{1/c} - 1\}$. For each $j \in \{0,1\}^{\log m}$, $(\dim_1(j), \ldots, \dim_c(j))$ is interpreted as specifying the identity of the unique non-zero entry of row j of M.

//$\mathcal{V}$ requests to evaluate $\widetilde{M}$ at input (r, r') where $r' = (r'_1, \ldots, r'_c) \in \left(\mathbb{F}^{\ell}\right)^c$.

1. $\mathcal{P} \to \mathcal{V}$: $2c$ different $(\log m)$-variate multilinear polynomials $E_1, \ldots, E_c$, $\mathsf{read\_counts}_1, \ldots \mathsf{read\_counts}_c$ and c different ℓ-variate multilinear polynomials $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_c$.
   //If $\mathcal{P}$ is honest, then $\mathsf{read\_counts}_1, \ldots \mathsf{read\_counts}_c$ and $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_c$ map $\{0,1\}^{\log m}$ to $\{0, \ldots, m-1\}$, as these are "counter polynomials" for each of the c memories.
   //If $\mathcal{P}$ is honest, then $E_1, \ldots, E_c$ contain the values returned by each read operation that the sparse polynomial evaluation algorithm of Section 3.1 makes to each of the c memories.
2. Recall (Equation 23) that $\widetilde{M}(r, r') = \sum_{k \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, k) \cdot \prod_{i=1}^{c} E_i(k)$, assuming that
   - $\forall k \in \{0,1\}^{\log m}$, $E_i(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\dim_i(k)), r'_i)$.

   Hence, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $g(k) := \widetilde{\mathsf{eq}}(r, k) \cdot \prod_{i=1}^{c} E_i(k)$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, k) \prod_{i=1}^{c} E_i(k)$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_z) \overset{?}{=} v_{E_i}$ for $i = 1, \ldots, c$. Here, $v_{E_1}, \ldots, v_{E_c}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the above equalities hold with one oracle query to each E_i.
   // The following checks if E_i is well-formed as per the first bullet in Step 2 above.
4. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
   //In practice, one would apply a single sum-check protocol to a random linear combination of the below polynomials. For brevity, we describe the protocol as invoking c independent instances of sum-check.
5. $\mathcal{V} \leftrightarrow \mathcal{P}$: For $i = 1, \ldots, c$, run a sum-check-based protocol for "grand products" ([Tha13, Proposition2] or [SL20, Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r''_i \in \mathbb{F}^{\ell}, r'''_i \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r'''_i) \overset{?}{=} v_{E_i}$
   - $\dim_i(r'''_i) \overset{?}{=} v_i$; $\mathsf{read\_counts}_i(r'''_i) \overset{?}{=} v_{\mathsf{read\_counts}_i}$; and $\mathsf{final\_counts}_i(r''_i) \overset{?}{=} v_{\mathsf{final\_counts}_{\mathsf{row}}}$
6. $\mathcal{V}$: check that the remaining equations hold with an oracle query to each of $E_i, \dim_i, \mathsf{read\_counts}_i, \mathsf{final\_counts}_i$.

Figure 7: Evaluation procedure of our sparse polynomial commitment scheme, optimized for its application to M in Lasso.

6.2 Background on the sum-check protocol

For each round $j = 1, \ldots, \log N$ of the sum-check protocol, the prescribed prover message is the degree-2 univariate polynomial s_j where

$$s_j(c) = \sum_{(b_{j+1}, b_{j+2}, \ldots, b_{\log N}) \in \{0,1\}^{\log(N)-j}} \widetilde{u}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) \cdot \widetilde{t}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}). \tag{25}$$

Here, $r_1, \ldots, r_{j-1}$ are random field elements chosen by the verifier in rounds $1, 2, \ldots, j-1$. The prover will specify s_j by sending its evaluations at 3 inputs, say, $s_j(0)$, $s_j(1)$, and $s_j(-1)$.

6.3 Proof of Theorem 1

Proof of Theorem 1. Observe that

$$\widetilde{u}(r_1, b_2, \ldots, b_{\log N}) = (1 - r_1) \cdot \widetilde{u}(0, b_2, \ldots, b_{\log N}) + r_1 \cdot \widetilde{u}(1, b_2, \ldots, b_{\log N})$$

This holds because the left hand size and right hand sides are both multilinear polynomials that agree on all inputs $(r_1, b_2, \ldots, b_{\log N}) \in \{0,1\}^{\log N}$.

Let $S_u = \{i = (i_1, \ldots, i_{\log N}) \in \{0,1\}^{\log N} \colon u_i \neq 0\}$ denote the non-zero entries of u. For every $i \in S_u$, and every round j of the sum-check protocol, $\mathcal{P}$ will store the value

$$\chi_i(r_1, \ldots, r_{j-1}, i_j, i_{j+1}, \ldots, i_{\log N}) = \prod_{k=1}^{j-1} \left(i_k \cdot r_k + (1 - i_k) \cdot (1 - r_k) \right). \tag{26}$$

Note that given all such values for round j, the prover can compute all the relevant values for round $j+1$ in time $O(m)$. This is because there are m elements in $i \in S_u$ and computing $\chi_i(r_1, \ldots, r_{j-1}, r_j, i_{j+1}, \ldots, i_{\log N})$ given $\chi_i(r_1, \ldots, r_{j-1}, i_j, i_{j+1}, \ldots, i_{\log N})$ can be done with just one field multiplication. Hence, maintaining all such values across all rounds j takes time $O(m \log N)$ in total for the prover.

By standard multilinear Lagrange interpolation (see [Tha22, Chapter 3]),

$$\widetilde{u}(r_1, \ldots, r_{\log N}) = \sum_{i \in S_u} u_i \cdot \chi_i(r_1, \ldots, r_{\log N}), \tag{27}$$

where u_i denotes the i'th entry of u when its entries are indexed by bit-vectors $\{0,1\}^{\log N}$. Hence, for any $c \in \mathbb{F}$,

$$\begin{aligned} s_j(c) &= \sum_{(b_{j+1}, b_{j+2}, \ldots, b_{\log N}) \in \{0,1\}^{\log(N)-j}} \widetilde{u}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) \cdot \widetilde{t}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) \\ &= \sum_{(b_{j+1}, b_{j+2}, \ldots, b_{\log N}) \in \{0,1\}^{\log(N)-j}} \left(\sum_{i \in S_u} u_i \cdot \chi_i(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) \right) \cdot \widetilde{t}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) \\ &= \sum_{i \in S_u} u_i \cdot \chi_i(r_1, \ldots, r_{j-1}, c, i_{j+1}, \ldots, i_{\log N}) \cdot \widetilde{t}(r_1, \ldots, r_{j-1}, c, i_{j+1}, \ldots, i_{\log N}). \end{aligned} \tag{28}$$

The final equality above exploits the fact that for any $(b_{j+1}, \ldots, b_{\log N}) \in \{0,1\}^{\log(N)-j}$, if

$$(i_{j+1}, \ldots, i_{\log N}) \neq (b_{j+1}, \ldots, b_{\log N}),$$

then

$$\chi_i(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) = 0.$$

To see this, observe that

$$\chi_i(x_1, \ldots, x_{\log N}) = \prod_{k=1}^{\log N} \left(i_k x_k + (1 - i_k)(1 - x_k) \right),$$

and if $i_k = 1$ and $x_k = 0$ or vice versa then the k'th term of this product is zero.

So to compute $s_j(c)$ for $c \in \{0, 1, -1\}$, the prover directly computes Expression (28). Given that the prover maintains in each round j the values in Expression (26), in round j computing $s_j(c)$ takes time

$$O\left(m \cdot \mathsf{evaltime}\left(\widetilde{t}\right)\right).$$

Across all $\log N$ rounds of the sum-check protocol, this entails total prover time

$$O\left(m \cdot \mathsf{evaltime}\left(\widetilde{t}\right)\right).$$

□

6.4 Improving the runtime to $O(m \log N)$ if $\widetilde{t}$ has additional structure

In the proof of Theorem 1, the prover time is bottlenecked by evaluating $\widetilde{t}$ at all points of the form

$$(r_1, \ldots, r_{j-1}, c, i_{j+1}, \ldots, i_{\log N}),$$

where $i = (i_1, \ldots, i_{\log N})$ ranges over $i \in S_u$ and $c \in \{0, 1, -1\}$. For all the example tables in Section 4.3, given

$$\widetilde{t}(r_1, \ldots, r_{j-1}, i_j, i_{j+1}, \ldots, i_{\log N}),$$

it takes only constant time (in most cases, just one multiplication by a power of two and one field addition) to compute

$$\widetilde{t}(r_1, \ldots, r_{j-1}, r_j, i_{j+1}, \ldots, i_{\log N}).$$

This reduces the prover time for all such tables from $O(m \log^2 N)$ of Theorem 1 down to $O(m \log N)$.

6.5 Improving the runtime to $O(cm)$

Fundamentally different techniques are required to reduce the prover's runtime to $O(m)$.

We begin by explaining how to implement the prover in $O(m)$ time under the assumption that $\widetilde{t}$ has total degree one, i.e.,

$$\widetilde{t}(r_1, \ldots, r_{\log N}) = \sum_{j=1}^{\log N} d_j r_j.$$

This is sufficient to capture most of the example tables considered in Section 3.3.1. Later (Section 6.5.2) we explain how to implement the prover in $O(m)$ time for a larger class of tables.

6.5.1 Handling tables for which $\widetilde{t}$ has total degree 1

Theorem 6. *Suppose that the multilinear extension $\widetilde{t}$ of t has total degree 1, i.e., can be written as $\widetilde{t}(r_1, \ldots, r_{\log N}) = \sum_{k=1}^{\log N} d_k \cdot r_k$ for some field elements $d_1, \ldots, d_{\log N} \in \mathbb{F}$. Then the prover in the sparse-dense sum-check protocol can be implemented in $O(m)$ field operations.*[20]

Proof. We begin by showing that $\widetilde{t}$ having total degree 1 ensures that altering the value of a single variable leads to "simple" changes in the output of $\widetilde{t}$.

[20] To clarify, Theorem 6 also assumes that for each index b, the b'th table entry, t_b, can be computed in constant time (otherwise, it is impossible to even compute the correct answer $\langle u, t \rangle$ in the sparse-dense sum-check protocol in $O(m)$ time).

Understanding the effect on an evaluation of $\widetilde{t}$ of altering the j'th variable. Let

$$\widetilde{t}(r_1, \dots, r_{\log N}) = \sum_{j=1}^{\log N} d_j r_j.$$

This ensures that for any $c \in \mathbb{F}$, and any $(r_1, \dots, r_{j-1}, b_j, b_{j+1}, \dots, b_{\log N}) \in \mathbb{F}^j \times \{0,1\}^{\log(N)-j}$,

$$\widetilde{t}(r_1, \dots, r_{j-1}, c, b_{j+1}, \dots, b_{\log N}) = \widetilde{t}(r_1, \dots, r_{j-1}, b_j, b_{j+1}, \dots, b_{\log N}) + (c - b_j) \cdot d_j. \tag{29}$$

The important implication of Equation (29) is that "revising' the j'th variable value from b_j to c affects the evaluation of $\widetilde{t}$ by an additive term $(c - b_j) \cdot d_j$. The key here is that this term can be computed in $O(1)$ time, and does *not* depend on the variables $b_{\log(m)+1}, \dots, b_{\log N}$.

In Section 6.5.2, we extend our techniques to achieve $O(m)$ prover time whenever $\widetilde{t}$ can be decomposed into a sum of $\eta = O(1)$ polynomials $\widetilde{t}_1, \dots, \widetilde{t}_\eta$ such that the following holds. There exist values $\mathsf{a}_\ell(c, j, b_j)$, $\mathsf{m}_\ell(c, j, b_j)$ such that "revising" the j'th variable from b_j to c affects the evaluation of $\widetilde{t}_\ell$ by a multiplicative factor of $\mathsf{m}_\ell(c, j, b_j)$ and an additive factor of $\mathsf{a}_\ell(c, j, b_j)$. Moreover, these factors $\mathsf{a}_\ell(c, j, b_j)$ and $\mathsf{m}_\ell(c, j, b_j)$ can be evaluated in $O(1)$ time and do *not* depend on the variables $b_{\log(m)+1}, \dots, b_{\log N}$. The special case of $\widetilde{t}$ having total degree 1, which we consider first, corresponds to $\eta = 1$, $\mathsf{a}_\ell(c, j, b_j) = (c - b_j) \cdot d_j$ and $\mathsf{m}_\ell(c, j, b_j) = 1$.

Values computed by the prover at the start of the protocol. For every $k \in \{0,1\}^{\log m}$, let $\mathsf{extend}_\ell(k)$ denote the set of all vectors in $\{0,1\}^\ell$ whose first $\log m$ entries equal k. At the start of the protocol, the prover computes the following two values q_k and z_k for every $k \in \{0,1\}^{\log m}$:

$$q_k \coloneqq \sum_{y \in \mathsf{extend}_{\log N}(k)} \widetilde{u}(y) \cdot \widetilde{t}(y) \tag{30}$$

$$z_k \coloneqq \sum_{y \in \mathsf{extend}_{\log N}(k)} \widetilde{u}(y). \tag{31}$$

Since u is m-sparse, all m quantities q_k and z_k can be computed in $O(m)$ total time.

The prover also computes "a binary tree of aggregations" of the above values. Specifically, let us think of the m different q_k (respectively, z_k) values as the roots of a binary tree Q (respectively, Z), and assign each node in Q and Z the value equal to the sum of its leaves. These values can be computed in $O(m)$ time in total.

For example, the roots of Q and Z respectively store

$$\sum_{k \in \{0,1\}^{\log m}} q_k = \sum_{y \in \{0,1\}^{\log N}} \widetilde{u}(y) \cdot \widetilde{t}(y),$$

and

$$\sum_{k \in \{0,1\}^{\log m}} z_k = \sum_{y \in \{0,1\}^{\log N}} \widetilde{u}(y).$$

Likewise, the two children of the root in Q store:

$$\sum_{k=(k_1, \dots, k_{\log m}) \in \{0,1\}^{\log m} \colon k_1 = 0} q_k, \tag{32}$$

and

$$\sum_{k=(k_1, \dots, k_{\log m}) \in \{0,1\}^{\log m} \colon k_1 = 1} q_k, \tag{33}$$

and similarly for Z. In general, let $Q^{(j)} \in \mathbb{F}^{2^j} \in \mathbb{F}^{2^j}$ is the vector of values assigned to nodes at at depth j of Q, and similarly let $Z^{(j)}$ denote the corresponding vector of values for Z. For example, $Q^{(1)}$ is the length-2 vector whose two entries are given in Equations (32) and (33).

The prover's workflow in the first $\log m$ rounds. Recall from Equation (25) that

$$s_j(c) = \sum_{(b_{j+1}, b_{j+2}, \ldots, b_{\log N}) \in \{0,1\}^{\log(N)-j}} \widetilde{u}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) \cdot \widetilde{t}(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}).$$

And recall from Equation (27) that

$$\widetilde{u}(r_1, \ldots, r_{\log N}) = \sum_{i \in S_u} u_i \cdot \chi_i(r_1, \ldots, r_{\log N}).$$

Moreover, for any $i = (i_1, \ldots, i_{\log N}) \in \{0,1\}^{\log N}$, if $i_j = 1$, then

$$\chi_i(r_1, \ldots, r_{j-1}, c, i_{j+1}, \ldots, i_{\log N}) = c \cdot \chi_i(r_1, \ldots, r_{j-1}, i_j, i_{j+1}, \ldots, i_{\log N}),$$

and if $i_j = 0$ then

$$\chi_i(r_1, \ldots, r_{j-1}, c, i_{j+1}, \ldots, i_{\log N}) = (1-c) \cdot \chi_i(r_1, \ldots, r_{j-1}, i_j, i_{j+1}, \ldots, i_{\log N}).$$

Based on the above equations, it can be derived that

$$\begin{aligned}
s_1(c) &= \sum_{y \in \{0,1\}^{\log N}} \widetilde{u}(c, y_2, \ldots, y_{\log N}) \cdot \widetilde{t}(c, y_2, \ldots, y_{\log N}) \\
&= \sum_{y \in \{0,1\}^{\log N}} \left(\sum_{i=(i_1, \ldots, i_{\log N}) \in S_u} u_i \cdot \chi_i(c, y_2, \ldots, y_{\log N}) \right) \cdot \widetilde{t}(c, y_2, \ldots, y_{\log N}) && (34) \\
&= \sum_{i=(i_1, \ldots, i_{\log N}) \in S_u} u_i \cdot \chi_i(c, i_2, \ldots, i_{\log N}) \cdot \widetilde{t}(c, i_2, \ldots, i_{\log N}) && (35) \\
&= \sum_{i=(i_1, \ldots, i_{\log N}) \in S_u} \chi_{i_1}(c) \cdot u_i \cdot \chi_i(i_1, i_2, \ldots, i_{\log N}) \cdot \left(\widetilde{t}(i_1, i_2, \ldots, i_{\log N}) + (c - i_1)\, d_1 \right) && (36) \\
&= \sum_{i=(i_1, \ldots, i_{\log N}) \in S_u \colon i_1 = 0} (1-c) \cdot u_i \cdot \left(\widetilde{t}(i_1, i_2, \ldots, i_{\log N}) + c d_1 \right) && (37) \\
&+ \sum_{i=(i_1, \ldots, i_{\log N}) \in S_u \colon i_1 = 1} c \cdot u_i \cdot \left(\widetilde{t}(i_1, i_2, \ldots, i_{\log N}) + (c-1)\, d_1 \right). && (38)
\end{aligned}$$

Here, Equation (34) invokes Equation (27). Equation (35) holds because $\chi_i(j) = 0$ whenever there is even a single index $\ell \in 2, 3, \ldots, \log N$ such that $i_\ell, j_\ell \in \{0,1\}$ and $i_\ell \neq j_\ell$. Equation (36) holds by Equation (29). Equation (37) holds because $\chi_i(i) = 1$ for all $i \in \{0,1\}^{\log N}$. Equation (38) holds by definition of χ_{i_1} (Equation (6)).

Round 1 computation. Expression (38) above equals:

$$\begin{aligned}
&\sum_{k=(k_1, \ldots, k_{\log m}) \in \{0,1\}^{\log m} \colon k_1 = 0} \left((1-c) \cdot q_k + (1-c) \cdot c \cdot d_1 \cdot z_k \right) \\
&+ \sum_{k=(k_1, \ldots, k_{\log m}) \in \{0,1\}^{\log m} \colon k_1 = 1} \left(c \cdot q_k + c \cdot (c-1) \cdot d_1 \cdot z_k \right)
\end{aligned}$$

For each $c \in \{-1, 0, 1\}$, computing this expression requires just a constant amount of work given the values of the two children of the root vertex of the trees Q (Equations (32) and (33)) and Z.

Round $j > 1$ computation. A similar calculation to the above reveals that $s_j(c)$ equals:

$$\begin{aligned}
&\sum_{(y_{j+1},\dots,y_{\log N})\in\{0,1\}^{\log(N)-j}} \widetilde{u}(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\cdot\widetilde{t}(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\\
=&\sum_{(y_{j+1},\dots,y_{\log N})\in\{0,1\}^{\log(N)-j}} \left(\sum_{i=(i_1,\dots,i_{\log N})\in S_u} u_i\cdot\chi_i(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\right)\cdot\widetilde{t}(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\\
=&\sum_{i=(i_1,\dots,i_{\log N})\in S_u} u_i\cdot\chi_i(r_1,\dots,r_{j-1},c,i_{j+1},\dots,i_{\log N})\cdot\widetilde{t}(r_1,\dots,r_{j-1},c,i_{j+1},\dots,i_{\log N})\\
=&\sum_{i=(i_1,\dots,i_{\log N})\in S_u} \chi_{(i_1,\dots,i_j)}(r_1,\dots,r_{j-1},c)\cdot u_i\cdot\left(\widetilde{t}(i_1,i_2,\dots,i_{\log N})+\sum_{k=1}^{j-1}(r_k-i_k)\,d_k+(c-i_j)d_j\right)\\
=&\sum_{(b_1,\dots,b_j)\in\{0,1\}^j}\ \sum_{i=(i_1,\dots,i_{\log N})\in S_u\,:\,(i_1,\dots,i_j)=(b_1,\dots,b_j)} u_i\cdot\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)\cdot\left(\widetilde{t}(i_1,i_2,\dots,i_{\log N})+\sum_{k=1}^{j-1}(r_k-i_k)\,d_k+(c-i_j)d_j\right)\\
=&\sum_{(b_1,\dots,b_j)\in\{0,1\}^j}\ \sum_{i=(i_1,\dots,i_{\log N})\in S_u\,:\,(i_1,\dots,i_j)=(b_1,\dots,b_j)} u_i\cdot\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)\cdot\widetilde{t}(i_1,i_2,\dots,i_{\log N}) \qquad (39)\\
+&\sum_{(b_1,\dots,b_j)\in\{0,1\}^j}\ \sum_{i=(i_1,\dots,i_{\log N})\in S_u\,:\,(i_1,\dots,i_j)=(b_1,\dots,b_j)} u_i\cdot\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)\cdot\left(\sum_{k=1}^{j-1}(r_k-i_k)\,d_k+(c-i_j)d_j\right) \qquad (40)
\end{aligned}$$

Recall that $Q^{(j)}\in\mathbb{F}^{2^j}$ (respectively $Z^{(j)}$) is the vector of values assigned to nodes at at depth j of Q. Let $v^{(j)}$ be the length-2^j vector with entries indexed by $(b_1,\dots,b_j)\in\{0,1\}^j$, with $(b_1,\dots,b_j)$'th entry given by

$$\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_j). \qquad (41)$$

Let $(v')^{(j)}$ be the vector with $(b_1,\dots,b_j)$'th entry given by

$$\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)=v^{(j-1)}[b_1,\dots,b_{j-1}]\cdot(b_jc+(1-b_j)(1-c)).$$

Note that $(v')^{(j)}$ can be computed in $O(2^j)$ time given $v^{(j-1)}$. And Expression (39) equals

$$\langle(v')^{(j)},Q^{(j)}\rangle.$$

Similarly, let $w^{(j)}$ be the length-2^j vector with entries indexed by $(b_1,\dots,b_j)\in\{0,1\}^j$, with $(b_1,\dots,b_j)$'th entry given by

$$\sum_{k=1}^{j-1}(r_k-i_k)\,d_k.$$

Let $(w')^{(j)}$ be the vector with $(b_1,\dots,b_j)$'th entry given by $w^{(j-1)}[b_1,\dots,b_j]+(c-b_j)\cdot d_j$. Then Expression (40) equals

$$\langle(v')^{(j)}\circ(w')^{(j)},Z^{(j)}\rangle,$$

where $(v')^{(j)}\circ(w')^{(j)}$ denotes the Hadamard (i.e., entry-wise) product of $(v')^{(j)}$ and $(w')^{(j)}$.

In summary, we have shown that for $j=1,\dots,m$, the sum-check prover's j'th message s_j can be computed in time $O(2^j)$ so long as the prover can compute $v^{(j)}$ and $w^{(j)}$ in this time bound. And indeed this is the case. For $v^{(j)}$, observe that, given all entries of $v^{(j-1)}$, one can compute $v^{(j)}$ in time $O(2^j)$. This is because

$$v^{(j)}[b_1,\dots,b_j]=v^{(j-1)}[b_1,\dots,b_{j-1}]\cdot(r_jb_j+(1-r_j)(1-b_j)).$$

Similarly, for $w^{(j)}$, observe that, given all entries of $w^{(j-1)}$, one can compute $w^{(j)}$ in time $O(2^j)$. This is because

$$w^{(j)}[b_1,\dots,b_j]=w^{(j-1)}[b_1,\dots,b_{j-1}]+(r_j-b_j)\cdot d_j.$$

Rounds $\log(m)+1,\ldots,2\log(m)$. Because the prover's runtime in round j takes time $O(2^j)$, by the time we reach round $\log m$, the prover is requiring $O(m)$ time per round. Hence, before the prover proceeds to round $m+1$, the prover needs to perform a "condensation" operation so that round $\log(m)+1$ behaves like round 1 in terms of prover complexity.

Round $j=\log(m)+1$ of the sparse-dense sum-check protocol is equivalent to round 1 of the sparse-dense sum-check protocol with the $(\log(N))$-variate polynomials $\widetilde{u}$ and $\widetilde{t}$ replaced by the following $(\log(N)-\log(m))$-variate polynomials $\widetilde{u}'$ and $\widetilde{t}'$:

$$\widetilde{u}'(b_{\log(m)+1},\ldots,b_{\log N}) \coloneqq \widetilde{u}(r_1,\ldots,r_{\log m},b_{\log(m)+1},\ldots,b_{\log N}),$$
$$\widetilde{t}'(b_{\log(m)+1},\ldots,b_{\log N}) \coloneqq \widetilde{t}(r_1,\ldots,r_{\log m},b_{\log(m)+1},\ldots,b_{\log N}).$$

So we merely need to show that in round $m+1$ of the sparse-dense sum-check protocol, the prover can in $O(m)$ time compute the necessary data structures about $\widetilde{u}'$ and $\widetilde{t}'$, namely (per Equations (30) and (31)) the following quantities:

$$q'_k \coloneqq \sum_{y\in\mathsf{extend}_{\log(N)-\log(m)}(k)} \widetilde{u}'(y)\cdot\widetilde{t}'(y) \tag{42}$$

$$z'_k \coloneqq \sum_{y\in\mathsf{extend}_{\log(N)-\log(m)}(k)} \widetilde{u}'(y). \tag{43}$$

All m of the z'_k can be computed in $O(m)$ time, as $z'_k = z_k\cdot\chi_k(r_1,\ldots,r_{\log m})$ and the values

$$\{\chi_k(r_1,\ldots,r_{\log m})\colon k\in\{0,1\}^{\log m}\} \tag{44}$$

can all be computed in $O(m)$ total time, and in fact are precisely the contents of the vector $v^{(\log m)}$ (see Equation (41)) computed by the prover anyway during the course of the first $\log m$ rounds of sum-check.

All m of the q'_k values can also be computed in $O(m)$ total time. To see this, recall that there are at most m values $y=(y_1,\ldots,y_{\log N})\in\{0,1\}^{\log N}$ such that $\widetilde{u}(y)\neq 0$. For each such y, let $y=(y',y'')\in\{0,1\}^{\log m}\times\{0,1\}^{\log(N)-\log(m)}$. Then $\widetilde{u}'(y')=\sum_{k\in\{0,1\}^{\log m}}\chi_k(r_1,\ldots,r_{\log m})\cdot\widetilde{u}(k,y')$. Since the $\chi_k(r_1,\ldots,r_{\log m})$ values (Equation (44)) can all be computed in $O(m)$ total time, this means that all non-zero $\widetilde{u}'(y')$ values can in turn all be computed in $O(m)$ time. Let S be the set

$$\{y'\in\{0,1\}^{\log(N)-\log(m)}\colon \widetilde{u}'(y)\neq 0\},$$

and recall that S has size at most m. For all $y'\in S$, we can also compute $\widetilde{t}'(y')$ in $O(m)$ total time, by the following reasoning.

First, recall from Equation (29) that for round $j=\log m$ and $(b_1,\ldots,b_{\log N})\in\{0,1\}^{\log N}$,

$$\widetilde{t}_\ell(r_1,\ldots,r_j,b_{j+1},\ldots,b_{\log N}) = \widetilde{t}_\ell(b_1,\ldots,\ldots,b_{\log N}) + \sum_{k=1}^{j}(r_k-b_k)\cdot d_k$$

Using dynamic programming, the following values can be computed in total time $O(2^j)=O(m)$ for *all* $(b_1,\ldots,b_j)\in\{0,1\}^j$:

$$\sum_{k=1}^{j}(r_k-b_k)\cdot d_k. \tag{45}$$

Indeed, let $H^{(j)}$ be the length 2^j-array with entries indexed by $(b_1,\ldots,b_j)\in\{0,1\}^j$ and such that $H^{(j)}[b_1,\ldots,b_j]=\sum_{k=1}^{j}(r_k-b_k)\cdot d_k$. Then $H^{(j+1)}$ can be computed from $H^{(j)}$ in $O(2^{j+1})$ time, since $H^{(j)}[b_1,\ldots,b_j,b_{j+1}] = H^{(j)}[b_1,\ldots,b_j]+(r_{j+1}-b_{j+1})\cdot d_{j+1}$. This means $H^{(\log m)}$ can be computed in $O(\sum_{j=1}^{\log m}2^j)=O(m)$ time in total.

The prover, having computed and stored $\widetilde{t}_\ell(y)$ for all $y \in S$ in $O(m)$ total time at the start of the protocol (see Footnote 20), can use these values to compute

$$\widetilde{t}(r_1, \ldots, r_j, b_{j+1}, \ldots, b_{\log N})$$

for all $y \in S$ in $O(m)$ total time.

Remaining rounds. The prover's computation in the remaining rounds $(2\log(m)+1, \ldots, \log N)$ proceeds analogously to rounds $\log m, \ldots, 2\log m$. Every $\log m$ rounds, the prover perform a "condensation" operation so that subsequent round behaves like round 1 in terms of prover complexity. This entails computing quantities $q'_{k,\ell}$ and z'_k for each $k \in \{0,1\}^{\log m}$, defined analogously to Equations (42) and (43). As above, these m quantities can all be computed in time $O(m)$ per condensation operation. In total, the prover performs $O(C)$ condensation operations, so $O(Cm)$ time is spent on condensation operations. Outside of the condensation operations, the prover implements each "chunk" of $\log m$ rounds in $O(m)$ time. Since there are $O(C)$ chunks, this means that the total prover time is $O(Cm)$.

□

6.5.2 An $O(cm)$-time prover for tables with $\widetilde{t}$ having total degree larger than one

Theorem 7. *Suppose that $\widetilde{t}$ can be decomposed into a sum of $\eta = O(1)$ polynomials $\widetilde{t}_1, \ldots, \widetilde{t}_\eta$, i.e.,*

$$\widetilde{t} = \sum_{\ell=1}^{\eta} \widetilde{t}_\ell \tag{46}$$

and such that the following holds. For any $(r_1, \ldots, r_{j-1}, c) \in \mathbb{F}^j$ and any $(b_j, \ldots, b_{\log N}) \in \{0,1\}^{\log(N)-j+1}$, there exist values $\mathsf{a}_\ell(c, j, b_j)$, $\mathsf{m}_\ell(c, j, b_j)$, each of which can be evaluated in $O(1)$ time, and which do not *depend on the variables $b_{j+1}, \ldots, b_{\log N}$, such that:*

$$\widetilde{t}_\ell(r_1, \ldots, r_{j-1}, c, b_{j+1}, \ldots, b_{\log N}) = \mathsf{m}_\ell(c, j, b_j) \cdot \widetilde{t}_\ell(r_1, \ldots, r_{j-1}, b_j, b_{j+1}, \ldots, b_{\log N}) + \mathsf{a}_\ell(c, j, b_j). \tag{47}$$

Moreover, assume that for each $y \in \{0,1\}^{\log N}$ such that $\widetilde{t}(y) \neq 0$, and each $\ell = 1, \ldots, \kappa$, it holds that $\widetilde{t}_\ell(y)$ can be computed by the prover in $O(1)$ time.[21] *Then the prover in the sparse-dense sum-check protocol can be implemented in $O(m)$ field operations.*

The theorem continues to hold if the values a_ℓ and m_ℓ depend on $(r_1, \ldots, r_{j-1})$ in addition to c, j, and b_j. It also holds if a_ℓ and m_ℓ depend on $b_{j+1}, \ldots, b_{j+\gamma}$ for some $\gamma = O(1)$ (in addition to c, j, b_j and $(r_1, \ldots, r_{j-1})$).

The last sentence of Theorem 7 is needed to capture the two final example tables in Section 3.3.1, namely those capturing bitwise AND and Less-Than (LT) operations (for both of these examples, it suffices to take $\gamma = 1$). For example, recall from Equation (10) that for the table t capturing bitwise AND evaluations on b-bit inputs, the following holds:

$$\widetilde{t}(x, y) = 2^{2b} \cdot \sum_{i=1}^{b} 2^{i-1} \cdot x_i \cdot y_i + \sum_{i=1}^{b} 2^{i-1} x_i + 2^b \sum_{i=1}^{b} 2^{i-1} y_i.$$

We can express $\widetilde{t}$ as the sum of three multilinear polynomials $\widetilde{t}_1$, $\widetilde{t}_2$ and $\widetilde{t}_3$ corresponding to the three sums above. $\widetilde{t}_2$ and $\widetilde{t}_3$ have total degree 1 and hence Theorem 1 applies to them. However, Theorem 1 does not apply to $\widetilde{t}_1$, which has total degree 2.

[21] As with Footnote 20, if $\widetilde{t}(y)$ itself cannot be computed in $O(1)$ time for all $y \in \{0,1\}^{\log N}$ then the correct answer $\langle u, t\rangle$ cannot necessarily be computed by the prover in $O(m)$ time.

Let us order the variables of (x, y) so that x_1 comes first and y_1 comes second, followed x_2 in third and y_2 in fourth, and so on. Then for even values of $j = 2k$, changing the value of the j'th variable from y_k to r_j leads to an additive update to $\widetilde{t}_1(x, y)$ of

$$2^{2b+k-1} \cdot r_{j-1} \cdot (r_j - y_k),$$

which depends only on $j = 2k$, r_{j-1}, r_j and y_k. However, if $j = 2k - 1$ is odd, then the additive "effect" on $\widetilde{t}_1$ when changing the value of the jth variable from x_j to r_j is

$$2^{2b+k-1} \cdot (r_j - x_k) \cdot y_k.$$

This depends on variable $j + 1$ (i.e., on y_k).

Conceptually, this means that the value of y_k cannot be "ignored" by the sum-check prover algorithm during round $j = 2k - 1$, which is the round in which variable x_k is "processed". However, the proof of Theorem 7 shows that this does not substantially affect prover time, essentially because there are only two possible values of $y_{k+1} \in \{0, 1\}$ that the algorithm needs to contemplate.

Proof of Theorem 7. We will prove the theorem assuming Equation (47) holds, and then explain what modifications are necessary if $\mathsf{a}_\ell(c, j, b_j)$ and $\mathsf{m}_\ell(c, j, b_j)$ have additional dependencies as per the last two sentences of the theorem statement.

A consequence of Equation (47) is that for any $j \in \{1, \ldots, \log N\}$, and any $(b_1, \ldots, b_{\log N}) \in \{0, 1\}^{\log N}$,

$$\widetilde{t}_\ell(r_1, \ldots, r_j, b_{j+1}, \ldots, b_{\log N}) = \left(\prod_{k=1}^{j} \mathsf{m}_\ell(r_k, k, b_k)\right) \cdot \widetilde{t}_\ell(b_1, \ldots, \ldots, b_{\log N}) + \sum_{k=1}^{j} \mathsf{a}_\ell(r_k, k, b_k). \tag{48}$$

Values computed by the prover at the start of the protocol. At the start of the protocol, *for each polynomial* $\widetilde{t}_\ell$ in the decomposition of $\widetilde{t}$ of Equation (46), the prover computes the exact same $q_{k,\ell}$ and z_k values as in Section 6.5.1 and builds binary trees Q_ℓ and Z over them exactly as in Section 6.5.1. That is, for each $\ell = 1, \ldots, \kappa$,

$$q_{k,\ell} \coloneqq \sum_{y \in \mathsf{extend}_{\log N}(k)} \widetilde{u}(y) \cdot \widetilde{t}_\ell(y) \tag{49}$$

and z_k is defined exactly as in Definition (31). Q_ℓ is a binary tree over the $q_{k,\ell}$ values, where each internal node stores the sum of its two children, and Z is a binary tree over the z_k values.

The prover's workflow in the first $\log m$ rounds. Following the derivation in Section 6.5.1, we calculate the following convenient expression for the prover's first message polynomial s_1:

$$\begin{aligned}
s_1(c) &= \sum_{y \in \{0,1\}^{\log N}} \widetilde{u}(c, y_2, \ldots, y_{\log N}) \cdot \widetilde{t}(c, y_2, \ldots, y_{\log N}) \\
&= \sum_{y \in \{0,1\}^{\log N}} \left(\sum_{i=(i_1,\ldots,i_{\log N}) \in S_u} u_i \cdot \chi_i(c, y_2, \ldots, y_{\log N}) \right) \cdot \widetilde{t}(c, y_2, \ldots, y_{\log N}) \qquad (50) \\
&= \sum_{i=(i_1,\ldots,i_{\log N}) \in S_u} u_i \cdot \chi_i(c, i_2, \ldots, i_{\log N}) \cdot \widetilde{t}(c, i_2, \ldots, i_{\log N}) \\
&= \sum_{i=(i_1,\ldots,i_{\log N}) \in S_u} \chi_{i_1}(c) \cdot u_i \cdot \chi_i(i_1, i_2, \ldots, i_{\log N}) \cdot \left(\sum_{\ell=1}^{\eta} \left(\mathsf{m}_\ell(c, 1, i_1) \cdot \widetilde{t}_\ell(i_1, i_2, \ldots, i_{\log N}) + \mathsf{a}_\ell(c, 1, i_1) \right) \right) \\
&\qquad (51)
\end{aligned}$$

Here, Equation (50) invokes Equation (27) and Equation (51) holds by Equations (46) and (47).

Round 1 computation. Expression (51) above equals:

$$\begin{aligned}
&\sum_{k=(k_1,\dots,k_{\log m})\in\{0,1\}^{\log m}:\ k_1=0}\ \sum_{\ell=1}^{\kappa}\left(\chi_0(c)\cdot\mathsf{m}_\ell(c,1,0)\cdot q_{k,\ell}+\chi_0(c)\cdot\mathsf{a}_\ell(c,1,0)\cdot z_k\right)\\
+&\sum_{k=(k_1,\dots,k_{\log m})\in\{0,1\}^{\log m}:\ k_1=1}\ \sum_{\ell=1}^{\kappa}\left(\chi_1(c)\cdot\mathsf{m}_\ell(c,1,1)\cdot q_{k,\ell}+\chi_1(c)\cdot\mathsf{a}_\ell(c,1,1)\cdot z_k\right) \qquad (52)
\end{aligned}$$

For each $c\in\{-1,0,1\}$, computing this expression requires just a constant amount of work given the values of the two children of the root vertex of the trees $Q_1,\dots,Q_\kappa$ (Equations (32) and (33)) and $Z_1,\dots,Z_\kappa$.

Round $j>1$ computation. A similar calculation to the above reveals that $s_j(c)$ equals:

$$\begin{aligned}
&\sum_{(y_{j+1},\dots,y_{\log N})\in\{0,1\}^{\log(N)-j}} \widetilde{u}(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\cdot\widetilde{t}(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\\
=&\sum_{(y_{j+1},\dots,y_{\log N})\in\{0,1\}^{\log(N)-j}}\left(\sum_{i=(i_1,\dots,i_{\log N})\in S_u} u_i\cdot\chi_i(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\right)\cdot\widetilde{t}(r_1,\dots,r_{j-1},c,y_{j+1},\dots,y_{\log N})\\
=&\sum_{i=(i_1,\dots,i_{\log N})\in S_u} u_i\cdot\chi_i(r_1,\dots,r_{j-1},c,i_{j+1},\dots,i_{\log N})\cdot\widetilde{t}(r_1,\dots,r_{j-1},c,i_{j+1},\dots,i_{\log N})\\
=.&\sum_{i=(i_1,\dots,i_{\log N})\in S_u}\chi_{(i_1,\dots,i_j)}(r_1,\dots,r_{j-1},c)\cdot u_i\cdot\left(\sum_{\ell=1}^{\kappa}\left(\prod_{k=1}^{j-1}\mathsf{m}_\ell(r_k,k,b_k)\right)\widetilde{t}_\ell(i_1,i_2,\dots,i_{\log N})+\sum_{k=1}^{j-1}\mathsf{a}_\ell(r_k,k,b_k)\right)\\
=&\sum_{(b_1,\dots,b_j)\in\{0,1\}^j}\ \sum_{i=(b_1,\dots,b_j,i_{j+1},\dots,i_{\log N})\in S_u} u_i\cdot\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)\cdot\sum_{\ell=1}^{\kappa}\left(\left(\prod_{k=1}^{j-1}\mathsf{m}_\ell(r_k,k,b_k)\right)\widetilde{t}_\ell(i_1,i_2,\dots,i_{\log N})+\sum_{k=1}^{j-1}\mathsf{a}_\ell(r_k,k,b_k)\right)\\
=&\sum_{(b_1,\dots,b_j)\in\{0,1\}^j}\ \sum_{i=(b_1,\dots,b_j,i_{j+1},\dots,i_{\log N})\in S_u} u_i\cdot\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)\left(\sum_{\ell=1}^{\kappa}\left(\prod_{k=1}^{j-1}\mathsf{m}_\ell(r_k,k,b_k)\right)\widetilde{t}_\ell(i_1,i_2,\dots,i_{\log N})\right)\\
&\qquad (53)\\
+&\sum_{(b_1,\dots,b_j)\in\{0,1\}^j}\ \sum_{i=(b_1,\dots,b_j,i_{j+1},\dots,i_{\log N})\in S_u} u_i\cdot\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)\left(\sum_{\ell=1}^{\kappa}\sum_{k=1}^{j-1}\mathsf{a}_\ell(r_k,k,b_k)\right). \qquad (54)
\end{aligned}$$

Recall that for each $\ell=1,\dots,\kappa$, $Q_\ell^{(j)}\in\mathbb{F}^{2^j}$ (respectively $Z^{(j)}$) is the vector of values assigned to nodes at at depth j of Q_ℓ. As in Section 6.5.1, let $v^{(j)}$ be the length-2^j vector with entries indexed by $(b_1,\dots,b_j)\in\{0,1\}^j$, with $(b_1,\dots,b_j)$'th entry given by

$$\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_j).$$

Let $(v')^{(j)}$ be the vector with $(b_1,\dots,b_j)$'th entry given by

$$\chi_{(b_1,\dots,b_j)}(r_1,\dots,r_{j-1},c)=v^{(j-1,\ell)}[b_1,\dots,b_{j-1}]\cdot(b_jc+(1-b_j)(1-c)).$$

Note that $(v')^{(j)}$ can be computed in $O(2^j)$ time given $v^{(j-1)}$. For each $\ell=1,\dots,k$, let $x^{(j,\ell)}$ denote the vector with entries indexed by $b=(b_1,\dots,b_j)\in\{0,1\}^j$ whose b'th entry is $\prod_{k=1}^{j}\mathsf{m}_\ell(r_k,k,b_k)$, and let $(x')^{(j,\ell)}$ equal

$$x^{(j-1,\ell)}[b_1,\dots,b_{j-1}]\cdot\mathsf{m}_\ell(c,j,b_j).$$

Then Expression (53) equals

$$\sum_{\ell=1}^{\kappa}\langle(v')^{(j)}\circ(x')^{(j,\ell)},Q_\ell^{(j)}\rangle,$$

where $(v')^{(j)} \circ (x')^{(j,\ell)}$ denotes the Hadamard (i.e., entry-wise) product of $(v')^{(j)}$ and $(x')^{(j,\ell)}$.

Similarly, let $w^{(j)}$ be the length-2^j vector with entries indexed by $(b_1, \ldots, b_j) \in \{0,1\}^j$, with $(b_1, \ldots, b_j)$'th entry given by

$$\sum_{\ell=1}^{\kappa} \sum_{k=1}^{j-1} \mathsf{a}_\ell(r_k, k, b_k)$$

Let $(w')^{(j)}$ be the vector with $(b_1, \ldots, b_j)$'th entry given by $w^{(j-1)}[b_1, \ldots, b_j] + \sum_{\ell=1}^{\kappa} \mathsf{a}_\ell(c, j, b_j)$. Then Expression (54) equals

$$\langle (v')^{(j)} \circ (w')^{(j)}, Z^{(j)} \rangle.$$

In summary, we have shown that for $j = 1, \ldots, m$, the sum-check prover's j'th message s_j can be computed in time $O(\kappa \cdot 2^j)$ so long as the prover can compute $v^{(j)}$, $w^{(j)}$, and $x^{(j,\ell)}$ for $\ell = 1, \ldots, \kappa$ in this time bound. And indeed this is the case. This was explained for $v^{(j)}$ in Section 6.5.1. For $x^{(j,\ell)}$, observe that, given all entries of $x^{(j-1,\ell)}$, one can compute $x^{(j,\ell)}$ in time $O(2^j)$. This is because

$$x^{(j,\ell)}[b_1, \ldots, b_j] = v^{(j-1)}[b_1, \ldots, b_{j-1}] \cdot \mathsf{m}_\ell(r_j, j, b_j),$$

and we have assumed that $\mathsf{m}_\ell(r_j, j, b_j)$ can be computed in constant time. The case of $w^{(j)}$ is similar.

Rounds $\log(m) + 1, \ldots, 2\log(m)$. As in Section 6.5.1, round $j = \log(m) + 1$ of the sparse-dense sum-check protocol is equivalent to round 1 of the sparse-dense sum-check protocol with the $(\log(N))$-variate polynomials $\widetilde{u}$ and $\widetilde{t}$ replaced by the following $(\log(N) - \log(m))$-variate polynomials $\widetilde{u}'$ and $\widetilde{t}'$:

$$\widetilde{u}'(b_{\log(m)+1}, \ldots, b_{\log N}) := \widetilde{u}(r_1, \ldots, r_{\log m}, b_{\log(m)+1}, \ldots, b_{\log N}),$$
$$\widetilde{t}'(b_{\log(m)+1}, \ldots, b_{\log N}) := \widetilde{t}(r_1, \ldots, r_{\log m}, b_{\log(m)+1}, \ldots, b_{\log N}).$$

For $\ell = 1, \ldots, \kappa$, let

$$\widetilde{t}_\ell(b_{\log(m)+1}, \ldots, b_{\log N}) = \widetilde{t}_\ell(r_1, \ldots, r_{\log m}, b_{\log(m)+1}, \ldots, b_{\log N})$$

So we merely need to show that in round $m + 1$ of the sparse-dense sum-check protocol, the prover can in $O(m)$ time compute the necessary data structures about $\widetilde{u}'$ and $\widetilde{t}'_1, \ldots, \widetilde{t}'_\ell$, namely (per Equations (49) and (31)) the following quantities:

$$(q')_{k,\ell} := \sum_{y \in \mathsf{extend}_{\log(N)-\log(m)}(k)} \widetilde{u}'(y) \cdot \widetilde{t}'_\ell(y) \tag{55}$$

$$z'_k := \sum_{y \in \mathsf{extend}_{\log(N)-\log(m)}(k)} \widetilde{u}'(y). \tag{56}$$

All m of the z'_k can be computed in $O(m)$ time, as $z'_k = z_k \cdot \chi_k(r_1, \ldots, r_{\log m})$ exactly as per Section 6.5.1. All $m \cdot \kappa$ of the $q'_{k,\ell}$ values can also be computed in $O(m)$ total time. To see this, recall that there are at most m values $y = (y_1, \ldots, y_{\log N}) \in \{0,1\}^{\log N}$ such that $\widetilde{u}(y) \neq 0$. For each such y, let $y = (y', y'') \in \{0,1\}^{\log m} \times \{0,1\}^{\log(N)-\log(m)}$. Then $\widetilde{u}'(y') = \sum_{k \in \{0,1\}^{\log m}} \chi_k(r_1, \ldots, r_{\log m}) \cdot \widetilde{u}(k, y')$. Since the $\chi_k(r_1, \ldots, r_{\log m})$ values (Equation (44)) can all be computed in $O(m)$ total time, this means that all non-zero $\widetilde{u}'(y')$ values can in turn all be computed in $O(m)$ time. Let S be the set

$$\{y' \in \{0,1\}^{\log(N)-\log(m)} \colon \widetilde{u}'(y) \neq 0\},$$

and recall that S has size at most m. For all $y' \in S$, we can also compute $\widetilde{t}'_\ell(y')$ and $\ell = 1, \ldots, \kappa$ in $O(\kappa \cdot m)$ total time, by the following reasoning. First, recall from Equation (48) that for round $j = \log m$ and $(b_1, \ldots, b_{\log N}) \in \{0,1\}^{\log N}$,

$$\widetilde{t}_\ell(r_1, \dots, r_j, b_{j+1}, \dots, b_{\log N}) = \left(\prod_{k=1}^{j} \mathsf{m}_\ell(r_k, k, b_k)\right) \cdot \widetilde{t}_\ell(b_1, \dots, \dots, b_{\log N}) + \sum_{k=1}^{j} \mathsf{a}_\ell(r_k, k, b_k).$$

Using dynamic programming, the following values can be computed in total time $O(2^j) = O(m)$ for *all* $(b_1, \dots, b_j) \in \{0,1\}^j$:

$$\prod_{k=1}^{j} \mathsf{m}_\ell(r_k, k, b_k) \tag{57}$$

and

$$\sum_{k=1}^{j} \mathsf{a}_\ell(r_k, k, b_k). \tag{58}$$

The prover, having computed and stored $\widetilde{t}_\ell(y)$ for all $y \in S$ in $O(m)$ total time at the start of the protocol (see Footnote 21), can use these values to compute

$$\widetilde{t}_\ell(r_1, \dots, r_j, b_{j+1}, \dots, b_{\log N})$$

for all $y \in S$ in $O(m)$ total time.

Remaining rounds. The prover's computation in the remaining rounds $(2\log(m) + 1, \dots, \log N)$ proceeds analogously to rounds $\log m, \dots, 2\log m$. Every $\log m$ rounds, the prover perform a "condensation" operation so that subsequent round behaves like round 1 in terms of prover complexity. This entails computing quantities $q'_{k,\ell}$ and z'_k for each $k \in \{0,1\}^{\log m}$, The total prover runtime is $O(Cm)$ as in Section 6.5.1.

Modifications if $\mathsf{a}_\ell(c, j, b_j)$ and $\mathsf{m}_\ell(c, j, b_j)$ have additional dependencies. If $\mathsf{a}_\ell(c, j, b_j)$ and $\mathsf{m}_\ell(c, j, b_j)$ depend on $(r_1, \dots, r_{j-1})$, in addition to c, j, and b_j, nothing about the prover's computation needs to change because in each round j, there is only one vector $(r_1, \dots, r_{j-1})$ for the algorithm to consider.

If if a_ℓ and m_ℓ also depend on $b_{j+1}, \dots, b_{j+\gamma}$ for some $\gamma = O(1)$ (in addition to c, j, b_j and $(r_1, \dots, r_{j-1})$), then some modifications are required. Conceptually, in each round j, the prover computation groups those $y \in \{0,1\}^{\log N}$ such that $\widetilde{u}(y) \neq 0$ so that elements of the same group all have a_ℓ and m_ℓ equal to the same quantities. These groups correspond to the internal nodes at level j of the binary trees Q_ℓ and Z. If a_ℓ and m_ℓ depend on $b_{j+1}, \dots, b_{j+\gamma}$, then the grouping needs to incorporate $b_{j+1}, \dots, b_{j+\gamma}$ as well. Fortunately, the number of groups under consideration in each round j grows by at most a factor of 2^γ because $(b_{j+1}, \dots, b_{j+\gamma}) \in \{0,1\}^\gamma$ can only take 2^γ values.

Specifically, Equations (52) (capturing the prover's round-one message) is updated to have $2^{1+\gamma}$ sums rather than 2 sums. Associating each sum with a bit-vector in $\{0,1\}^\gamma$, the i'th sum is over terms $k \in \{0,1\}^{\log m}$ that have with $(k_1, \dots, k_\gamma) = i$ (Equation (52) itself corresponds to the case $\gamma = 0$). Explicitly, Equation (52) becomes:

$$\sum_{i=(i_1,\dots,i_\gamma)\in\{0,1\}^\gamma} \; \sum_{k=(k_1,\dots,k_{\log m})\in\{0,1\}^{\log m}\,:\,(k_1,\dots,k_\gamma)=i} \; \sum_{\ell=1}^{\kappa} \left(\chi_i(c, k_2, \dots, k_\gamma) \cdot \mathsf{m}_\ell \cdot q_{k,\ell} + \chi_i(c, k_2, \dots, k_\gamma) \cdot \mathsf{a}_\ell \cdot z_k\right),$$

where the quantities a_ℓ and m_ℓ may depend on c, $k_1, \dots, k_\gamma$.

Similarly (53), and (54) are updated to become:

$$\sum_{(b_1,\dots,b_j)\in\{0,1\}^j} \; \sum_{i=(b_1,\dots,b_j,i_{j+1},\dots,i_{\log N})\in S_u} u_i \cdot \chi_{(b_1,\dots,b_{j+\gamma})}(r_1, \dots, r_{j-1}, c, b_{j+1}, \dots, b_{j+\gamma}) \left(\sum_{\ell=1}^{\kappa} \left(\prod_{k=1}^{j-1} \mathsf{m}_\ell(r_k, k, b_k)\right) \widetilde{t}_\ell(i_1, i_2, \dots, i_{\log N})\right)$$

$$+ \sum_{(b_1,\dots,b_{j+\gamma})\in\{0,1\}^{j+\gamma}} \; \sum_{i=(b_1,\dots,b_{j+\gamma},i_{j+\gamma},\dots,i_{\log N})\in S_u} u_i \cdot \chi_{(b_1,\dots,b_{j+\gamma})}(r_1, \dots, r_{j-1}, c, b_{j+1}, \dots, b_{j+\gamma}) \left(\sum_{\ell=1}^{\kappa} \sum_{k=1}^{j-1} \mathsf{a}_\ell(r_k, k, b_k)\right).$$

Here, we write $\mathsf{a}_\ell(r_k, k, b_k)$ and $\mathsf{m}_\ell(r_k, k, b_k)$ for simplicity and consistency with Equations (53) and (54), but these quantities may in general depend on $k, r_1, \ldots, r_k$ and $b_{k+1}, \ldots, b_{k+\gamma}$. □

Applications to the AND and LT tables. The discussion following the statement of Theorem 7 explained that the theorem applied to the lookup table for the AND instruction, whose (x, y)'th entry is $\sum_{i=1}^{b} 2^{i-1} x_i \cdot y_i$.

Recall from Section 3.3.1 that LT denotes the function that takes two b-bit inputs x and y as input and outputs 1 if $x < y$ and 0 otherwise. The appropriate lookup table to capture this function (after applying the transformation of Section 4.4) has (x, y)'th entry equal to $2^{2b} \cdot \mathsf{LT}(x, y) + 2^b \cdot y + x$. Let $\widetilde{\mathsf{LT}}$ denote the multilinear extension of the function $\mathsf{LT}\colon \{0,1\}^b \times \{0,1\}^b \to \mathbb{F}$. Then

$$\widetilde{t}(x, y) = \widetilde{\mathsf{LT}}(x, y) + \sum_{i=1}^{b} 2^{i-1} \cdot x_i + \sum_{i=1}^{b} 2^{b+i-1} \cdot y_i.$$

So long as we show that $\widetilde{\mathsf{LT}}(x, y)$ itself satisfies the requirements of Theorem 7, then so does $\widetilde{t}$ itself.

Deriving an expression for $\widetilde{\mathsf{LT}}$. Let $x = (x_1, \ldots, x_b)$ and $y = (y_1, \ldots, y_b)$.

For $i = 2, \ldots, b$, define $x_{>i} = (x_{i+1}, \ldots, x_b)$, and define

$$\widetilde{\mathsf{LT}}_i(x, y) = (1 - x_i) \cdot y_i \cdot \widetilde{\mathsf{eq}}(x_{>i}, y_{>i}),$$

where recall that $\widetilde{\mathsf{eq}}$ was defined in Equation (4).

We claim that

$$\widetilde{\mathsf{LT}}(x, y) = \sum_{i=1}^{b} \widetilde{\mathsf{LT}}_i(x, y). \tag{59}$$

Indeed, the right hand side of this equation is multilinear and agrees with $\mathsf{LT}(x, y)$ at all inputs $x, y \in \{0,1\}^b$. This is because, if j is the highest-order bit that differs between x and y, then $\widetilde{\mathsf{LT}}_{j'}(x, y) = 0$ for all $j' \neq j$, and $\widetilde{\mathsf{LT}}_j(x, y) = 1$ if and only if $x_j = 0$ and $y_j = 1$.

Showing that $\widetilde{\mathsf{LT}}(x, y)$ satisfies the requirement of Theorem 7 so long as $m \geq \log N$. Let us order the $2b$ variables of (x, y) (i.e., the variables of the polynomial $\widetilde{\mathsf{LT}}$) so that the low-order bits x_1 and y_1 get bound in the first two rounds of sparse-dense sum-check, x_2 and y_2 get bound in the next two rounds, and so on. For any $(r_1, \ldots, r_{j-1}, r_j) \in \mathbb{F}^j$ and any $(z_j, \ldots, z_{2b}) \in \{0,1\}^{2b-j+1}$, we must show that there exist values a_ℓ, m_ℓ, each of which can be evaluated in $O(1)$ time, and which do *not* depend on the variables $z_{j+2}, \ldots, z_{2b}$, such that:

$$\widetilde{\mathsf{LT}}(r_1, \ldots, r_{j-1}, r_j, z_{j+1}, \ldots, z_{2b}) = \mathsf{m} \cdot \widetilde{\mathsf{LT}}(r_1, \ldots, r_{j-1}, z_j, z_{j+1}, \ldots, z_{2b}) + \mathsf{a}. \tag{60}$$

Say that $j = 2k$ is even and let us write

$$x = (r_1, r_3, r_5, \ldots, r_{j-1}, z_{j+1}, \ldots, z_{2b-1})$$

and

$$y' = (r_2, r_4, r_6, \ldots, r_j, z_{j+2}, \ldots, z_{2b}).$$

Let

$$y = (r_2, r_4, r_6, \ldots, r_{j-1}, z_j, z_{j+2}, \ldots, z_{2b}).$$

Let k^* be the highest-order bit such that $x_{k^*} \neq y_{k^*}$.

- If $k < k^*$, then $\widetilde{t}(x,y) - \widetilde{t}(x,y') = 0$. This holds by the following reasoning. For all $i \le k$, $\widetilde{\mathsf{LT}}_i(x,y) = \widetilde{\mathsf{LT}}_i(x,y) = 0$ due to the factor $\widetilde{\mathsf{eq}}(x_{>i}, y_{>i})$ appearing in $\widetilde{\mathsf{LT}}_i$ and the fact that $x_{k^*} \neq y_{k^*}$. And for $i > k$, $\widetilde{\mathsf{LT}}_i(x,y) = \widetilde{\mathsf{LT}}_i(x,y')$ because y and y' differ only in the k'th coordinate, and for each $i > k$, $\widetilde{\mathsf{LT}}_i$ does not depend on inputs $1, \dots, i-1$.

- Suppose $k \ge k^*$. Then:

$$\widetilde{\mathsf{LT}}(x,y') = \sum_{i=1}^{b} \widetilde{\mathsf{LT}}_i(x,y') = \sum_{i=1}^{k} (1 - r_{2i-1}) \cdot r_{2i} \cdot \widetilde{\mathsf{eq}}(x_{>i}, y'_{>i}) + \sum_{i=k+1}^{b} \mathsf{LT}_i(x,y')$$
$$= \sum_{i=1}^{k} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \prod_{j=i+1}^{b} (x_i y'_i + (1-x_i)(1-y'_i)) \right) + \sum_{j=k+1}^{b} \mathsf{LT}_j(x,y')$$
$$= \sum_{i=1}^{k} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \prod_{j=i+1}^{b} (x_i y'_i + (1-x_i)(1-y'_i)) \right) \tag{61}$$

Here, Equation (61) holds because $x_i = y_i = y'_i$ for all $i > k$ by assumption that k^* is the most significant index at which x and y differ.

  - Suppose $k > k^*$. Then (assuming each $r_j \notin \{0,1\}$) Equation (61) equals

$$\mathsf{LT}(x,y) \cdot \frac{r_{j-1} z_j + (1 - r_{j-1})(1 - z_j)}{r_{j-1} r_j + (1 - r_{j-1})(1 - r_j)}.$$

  - Suppose $k = k^*$. Then Equation (61) equals

$$\sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \prod_{j=i+1}^{k} (x_i y'_i + (1-x_i)(1-y'_i)) \right) + (1 - r_{2k-1}) \cdot r_{2k}$$
$$= (1 - r_{j-1}) \cdot r_j + \sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (x_i y'_i + (1-x_i)(1-y'_i)) \right) \right) \cdot \left(r_{j-1} r_j + (1 - r_{j-1})(1 - r_j) \right).$$

Here, we have used that $x_i = y_i$ and $x_i, y_i \in \{0,1\}$ for all $i > k^*$. Meanwhile, $\mathsf{LT}(x,y)$ equals

$$(1 - r_{j-1}) \cdot z_j + \sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (x_i y'_i + (1-x_i)(1-y'_i)) \right) \right) \cdot \left(r_{j-1} z_j + (1 - r_{j-1})(1 - z_j) \right).$$

Let

$$\beta = \frac{r_{j-1} r_j + (1 - r_{j-1})(1 - r_j)}{r_{j-1} z_j + (1 - r_{j-1})(1 - z_j)}. \tag{62}$$

Hence,

$$\mathsf{LT}(x,y') = \mathsf{LT}(x,y) \cdot \beta + (1 - r_{2k-1}) \cdot r_{2k} - \beta(1 - r_{j-1}) \cdot z_j.$$

Summarizing, if we are in round $j = 2k$ with $k < k^*$ where k^* is the most significant where x and y differ, then $\mathsf{a} = 0$ and $\mathsf{m} = 1$. If $k > k^*$ then $\mathsf{a} = 0$ and[22]

$$\mathsf{m} = \frac{r_{j-1} r_j + (1 - r_{j-1})(1 - r_j)}{r_{j-1} z_j + (1 - r_{j-1})(1 - z_j)}.$$

[22] Note that while multiplicative inverses take super-constant time to compute, the denominator of the fraction only involves r_{j-1}, and z_j, and hence only takes two different values, as r_{j-1} is fixed by the verifier before starting round j and z_j is only ever 0 or 1. That is, the prover does not have to compute a different inverse for each tuple (x,y) with $\widetilde{u}(x,y) \neq 0$.

If $k = k^*$ then $\mathsf{m} = \beta$ (defined in Equation (62)) and

$$\mathsf{a} = (1 - r_{2k-1}) \cdot r_{2k} - \beta(1 - r_{j-1}) \cdot z_j.$$

The case of $j = 2k - 1$ is similar and we highlight the main differences. In this case, let us write

$$x' = (r_1, r_3, r_5, \ldots, r_j, z_{j+2}, \ldots, z_{2b-1})$$

and

$$x = (r_1, r_3, r_5, \ldots, z_j, z_{j+2}, \ldots, z_{2b-1}),$$

and

$$y = (r_2, r_4, r_6, \ldots, r_{j-1}, z_j, z_{j+2}, \ldots, z_{2b}).$$

Then we have the following analog of Equation (61):

$$\begin{aligned}
\widetilde{\mathsf{LT}}(x', y) = \sum_{i=1}^{b} \widetilde{\mathsf{LT}}_i(x', y) &= \sum_{i=1}^{k} (1 - r_{2i-1}) \cdot r_{2i} \cdot \widetilde{\mathsf{eq}}(x_{>i}, y'_{>i}) + \sum_{i=k+1}^{b} \mathsf{LT}_i(x', y) \\
&= \sum_{i=1}^{k} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \prod_{j=i+1}^{b} (x'_i y_i + (1 - x'_i)(1 - y_i)) \right) + \sum_{j=k+1}^{b} \mathsf{LT}_j(x', y) \\
&= \sum_{i=1}^{k} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \prod_{j=i+1}^{b} (x'_i y_i + (1 - x'_i)(1 - y_i)) \right) \qquad (63)
\end{aligned}$$

The main difference when $j = 2k - 1$ compared to the analysis for $j = 2k$ lies is in the case that $k = k^*$. In this case, Equation (63) equals

$$\begin{aligned}
&\sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \prod_{j=i+1}^{k} (x'_i y_i + (1 - x'_i)(1 - y_i)) \right) + (1 - r_{2k-1}) \cdot y_k \\
&= (1 - r_j) \cdot y_k + \sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (x'_i y_i + (1 - x'_i)(1 - y_i)) \right) \right) \cdot (r_j y_k + (1 - r_j)(1 - y_k))
\end{aligned}$$

Meanwhile, $\mathsf{LT}(x, y)$ equals

$$(1 - x_k) \cdot y_k + \sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (x'_i y_i + (1 - x'_i)(1 - y_i)) \right) \right) \cdot (x_k y_k + (1 - x_k)(1 - y_k)) = -0,$$

where we have used the fact that $k = k^*$ and $x_{k^*} \neq y_{k^*}$ by assumption.

Hence,

$$\mathsf{LT}(x', y) = \mathsf{LT}(x, y) + (1 - r_j) \cdot y_j - (1 - x_j) \cdot y_j + \eta$$

where η equals

$$\begin{aligned}
&= \sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (x'_i y_i + (1 - x'_i)(1 - y_i)) \right) \right) \cdot (r_j y_k + (1 - r_j)(1 - y_k)) \\
&\sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (r_{2i-1} r_{2i} + (1 - r_{2i-1})(1 - r_{2i})) \right) \right) \cdot (r_j y_k + (1 - r_j)(1 - y_k)). \qquad (64)
\end{aligned}$$

In this case, we can set $\mathsf{a} = (1 - r_j) \cdot y_j - (1 - x_j) \cdot y_j + \eta$ and $\mathsf{m} = 1$. The prover can devote $O(\log N)$ total time over the *entire course* of the sum-check protocol to ensure that η can always be computed in $O(1)$ time.

That is, at all odd rounds $j = 2k - 1$ of the sparse-dense sum-check protocol it will update the quantity

$$\sum_{i=1}^{k-1} \left((1 - r_{2i-1}) \cdot r_{2i} \cdot \left(\prod_{j=i+1}^{k-1} (r_{2i-1} r_{2i} + (1 - r_{2i-1})(1 - r_{2i})) \right) \right).$$

Note that when moving from round j to round $j + 2$ this quantity can be updated in $O(1)$ time. Moreover, given this quantity, for either of the two possible values of $y_k \in \{0, 1\}$, η can be computed in $O(1)$ time.

7 Surge: A Generalization of Spark

The technical core of the SuperLasso lookup argument is a generalization of Spark we call Surge. Recall (Section 5, Figure 7) that Spark allows the untrusted Lasso prover to commit to $\widetilde{M}$, purported to be the multilinear extension of an $m \times N$ matrix M, with each row equal to a unit vector, such that $M \cdot t = a$.

The commitment phase of Surge is analogous to that of Spark. Surge differs from Spark primarily in that the Surge prover is able to prove a much more general class of statements about the committed polynomial $\widetilde{M}$ than can the Spark prover (the Spark prover is only able to provide evaluations of $\widetilde{M}$).

In order to motivate Surge, it is helpful to first explain at a high level how SuperLasso works.

Overview of SuperLasso. In SuperLasso, as in Lasso (see Equation (12)), after committing to $\widetilde{M}$, the SuperLasso verifier picks a random $r \in \mathbb{F}^{\log m}$ and seeks to confirm that

$$\sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r, j) \cdot t(j) = \widetilde{a}(r). \tag{65}$$

In Lasso, the verifier obtains $\widetilde{a}(r)$ via the commitment to $\widetilde{a}$. The prover then establishes that the left hand size of Equation (65) equals this quantity by applying the sum-check protocol to the $(\log(N))$-variate polynomial $g(j) = \widetilde{M}(r, j) \cdot t(j)$. This effectively reduces the verifier's task of confirming that Equation (65) holds, to the task of evaluating $\widetilde{M}$ at a random point $(r, r') \in \mathbb{F}^{\log m} \times \mathbb{F}^{\log N}$ and the task of evaluating an extension polynomial $\hat{t}$ of t at $r' \in \mathbb{F}^{\log N}$.

In SuperLasso, the prover establishes Equation (65) *directly* using Surge. Specifically, Surge generalizes Spark's procedure for generating evaluation proofs, to directly produce a proof as to the value of the left hand side of Equation (65). Essentially, the proof *proves* that the prover correctly ran a (very efficient) algorithm for evaluating the left hand side of Equation (65).

A roughly $O(\alpha m)$-time algorithm for computing the left hand side of Equation (65). Recalling Equation (24),

$$\widetilde{M}(r, y) = \sum_{(i,j) \in \{0,1\}^{\log m + \log N}} M_{i,j} \cdot \chi_i(r) \cdot \chi_j(y).$$

Hence, letting $\mathsf{nz}(i)$ denote the unique column in row i of M that contains a non-zero value (namely, the value 1), the left hand side of Equation (65) equals

$$\sum_{i \in \{0,1\}^{\log m}} \chi_i(r) \cdot T[\mathsf{nz}(i)]. \tag{66}$$

Suppose that T is SOS. This means that there is an integer $k \geq 1$ and $\alpha = k \cdot c$ tables $T_1, \ldots, T_\alpha$ of size $N^{1/c}$, as well as an α-variate multilinear polynomial g such that the following holds. Suppose that for every

$r = (r_1, \ldots, r_c) \in \left(\{0,1\}^{\log(N)/c}\right)^c,$

$$T[r] = g\left(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c]\right). \tag{67}$$

For each $i \in \{0,1\}^{\log m}$, let us decompose $\mathsf{nz}(i)$ and $(\mathsf{nz}_1(i), \ldots, \mathsf{nz}_c(i)) \in [N^{1/c}]^c$. Then Expression (66) equals

$$\sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i,r) \cdot g\left(T_1[\mathsf{nz}_1(i)], \ldots, T_k[\mathsf{nz}_1(i)], T_{k+1}[\mathsf{nz}_2(i)], \ldots, T_{2k}[\mathsf{nz}_2(i)], \ldots, T_{\alpha-k+1}[\mathsf{nz}_c(i)], \ldots, T_\alpha[\mathsf{nz}_c(i)]\right). \tag{68}$$

The algorithm to compute Expression (68) simply initializes all tables $T_1, \ldots, T_\alpha$, then iterates over every $i \in \{0,1\}^m$ and computes the i'th term of the sum with a single lookup into each table (of course, the algorithm evaluates g at the results of the lookups into $T_1, \ldots, T_\alpha$, and multiplies the result by $\widetilde{\mathsf{eq}}(i,r)$).

Description of Surge. The commitment to $\widetilde{M}$ in Surge consists of commitments to c multilinear polynomials $\dim_1, \ldots, \dim_c$, each over $\log m$ variables. $\dim_i$ is purported to be the multilinear extension of nz_i.

The verifier chooses $r \in \{0,1\}^{\log m}$ at random and requests that the Surge prover prove that the committed polynomial $\widetilde{M}$ satisfy Equation (66). The prover does so by proving it ran the aforementioned algorithm for evaluating Expression (68). Following the memory-checking procedure in Section 5, with each table $T_i \colon i = 1, \ldots, \alpha$ viewed as a memory of size $N^{1/c}$), this entails committing for each i to $\log(m)$-variate multilinear polynomials E_i and $\mathsf{read\_counts}_i$ (purported to capture the value and count returned by each of the m lookups into T_i) and a $\log(N^{1/c})$-variate multilinear polynomial $\mathsf{final\_counts}_i$ (purported to capture the final count for each memory cell of T_i.

Let $\widetilde{t_i}$ be the mutlilinear extension of the vector t_i whose j'th entry is $T_i[j]$. The sum-check protocol is applied to compute

$$\sum_{j \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r,j) \cdot g\left(E_1(j), \ldots, E_\alpha(j)\right). \tag{69}$$

At the end of the sum-check protocol, the verifier needs to evaluate $\widetilde{\mathsf{eq}}(r,r') \cdot g(E_1(r'), \ldots, E_\alpha(r'))$ at a random point $r' \in \mathbb{F}^{\log m}$, which it can do with one evaluation query to each E_i (the verifier can compute $\widetilde{\mathsf{eq}}(r,r')$ on its own in $O(\log m)$ time).

The verifier must still check that each E_i is well-formed, in the sense that $E_i(j)$ equals $T_i[\dim_i(j)]$ for all $j \in \{0,1\}^{\log m}$. This is done by applying the grand product argument exactly as in Spark to confirm that for each of the α memories, (see Claims 3 and 4 and Figure 7). At the end of the grand product argument, for each $i = 1, \ldots, \alpha$, the verifier needs to evaluate each of $\dim_i$, $\mathsf{read\_counts}_i$, $\mathsf{final\_counts}_i$ at a random point, which it can do with one query to each. The verifier also needs to evaluate the multilinear extension $\widetilde{t_i}$ of each sub-table T_i for each $i = 1, \ldots, \alpha$ at a single point. T being SOS guarantees that the verifier can compute each of these evaluations in $O(\log(N)/c)$ time.

Prover time. Besides committing to the polynomials $\dim_i, E_i, \mathsf{read\_counts}_i, \mathsf{final\_counts}_i$ for each of the α memories and producing one evaluation proof for each (in practice, these would be batched), the prover must compute its messages in the sum-check protocol used to compute Expression (69) and the grand product arguments (which can be batched). Using the linear-time sum-check protocol [**?**, Tha13], the prover can compute its messages in the sum-check protocol used to compute Expression (69) with $O(bk\alpha m)$ field operations, where recall that $\alpha = kc$ and b is the number of monomials in g. If $k = O(1)$, then this is $O(bcm)$ time. For many tables of practical interest, the factor b can be eliminated The costs for the prover in the grand product argument is similar to Spark: $O(\alpha m + \alpha N^{1/c})$ field operations, plus committing to a low-order number of field elements.

Verification costs. The sum-check protocol used to compute Expression (69) consists of $\log m$ rounds in which the prover sends a univariate polynomial of degree at most k in each round. Hence, the prover sends $O(k \log m)$ field elements, and the verifier performs $O(k \log m)$ field operations. The costs of the grand product arguments (which can be batched) for the verifier are identical to Spark.

- Input: A polynomial commitment to the multilinear polynomials $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$, and a description of an SOS table T of size N.
- The prover $\mathcal{P}$ sends a Surge-commitment to the multilinear extension $\widetilde{M}$ of a matrix $M \in \{0,1\}^{m \times N}$. This consists of c different $(\log(m))$-variate multilinear polynomials $\dim_1, \ldots, \dim_c$ (see Figure 9 for details).
- The verifier $\mathcal{V}$ picks a random $r \in \mathbb{F}^{\log m}$ and sends r to $\mathcal{P}$. The verifier makes one evaluation query to $\widetilde{a}$, to learn $\widetilde{a}(r)$.
- $\mathcal{P}$ and $\mathcal{V}$ apply Surge (Figure 9), allowing $\mathcal{P}$ to prove that $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r,y)T[y] = \widetilde{a}(r)$.

Figure 8: Description of the SuperLasso lookup argument. Here, a denotes the vector of lookups and t the vector capturing the lookup table (Definition 1.1). A polynomial commitments to the multilinear extension polynomial $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$ is given to the verifier as input. If t is unstructured, then c will be set to 1.

Completeness and knowledge soundness of the polynomial IOP. Completeness holds by design and by completeness of the sum-check protocol and grand-product argument used.

By soundness of the sum-check protocol and grand-product argument, if the prover passes the verifier's checks in the polynomial IOP with probability more than an appropriately chosen threshold $\gamma = O(m + N^{1/c}/|\mathbb{F}|)$, then $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r,y)T[y] = v$, where $\widetilde{M}$ is the multilinear extension of the following matrix M. For $i \in \{0,1\}^{\log m}$, row i of M consists of all zeros except for entry $M_{i,j} = 1$, where $j = (j_1, \ldots, j_c) \in \{0, 1, \ldots, N^{1/c}\}^c$ is the unique column index such that $j_1 = \dim_1(i)$, …, $j_c = \dim_c(i)$.

Acknowledgements. We are grateful to Luís Fernando Schultz Xavier da Silveira for optimizations to an earlier version of Lasso. We would also like to thank Luís, Arasu Arun, and Patrick Towa for insightful comments and conversations. Justin Thaler was supported in part by NSF CAREER award CCF-1845125. Both Justin Thaler and Riad Wahby were supported in part by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Government or DARPA.

T is an SOS lookup table of size N, meaning there are $\alpha = kc$ tables $T_1, \ldots, T_\alpha$, each of size $N^{1/c}$, such that for any $r \in \{0,1\}^{\log N}$, $T[r] = g(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c])$. During the commit phase, $\mathcal{P}$ commits to c multilinear polynomials $\dim_1, \ldots, \dim_c$, each over $\log m$ variables. $\dim_i$ is purported to provide the indices of $T_{(i-1)k+1}, \ldots, T_{ik}$ the natural algorithm computing $\sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i, r) \cdot T[\mathsf{nz}[i]]$ (see Equation (68)).

$//\mathcal{V}$ requests $\langle u, t \rangle$, where the ith entry of t is $T[i]$.

1. $\mathcal{P} \rightarrow \mathcal{V}$: 2α different $(\log m)$-variate multilinear polynomials $E_1, \ldots, E_\alpha$, $\mathsf{read\_counts}_1, \ldots \mathsf{read\_counts}_\alpha$ and α different $(\log(N)/c)$-variate multilinear polynomials $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_\alpha$.
   $//E_i$ is purported to specify the values of each of the m reads into T_i.
   $//\mathsf{read\_counts}_1, \ldots \mathsf{read\_counts}_\alpha$ and $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_\alpha$, are "counter polynomials" for each of the α sub-tables T_i.
2. $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $h(k) := \widetilde{\mathsf{eq}}(r, k) \cdot g(E_1(k), \ldots, E_\alpha(k))$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} g(E_1(k), \ldots, E_\alpha(k))$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_z) \stackrel{?}{=} v_{E_i}$ for $i = 1, \ldots, \alpha$. Here, $v_{E_1}, \ldots, v_{E_\alpha}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the above equalities hold with one oracle query to each E_i.
4. // The following checks if E_i is well-formed, i.e., that $E_i(j)$ equals $T_i[\dim_i(j)]$ for all $j \in \{0,1\}^{\log m}$.
5. $\mathcal{V} \rightarrow \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
   //In practice, one would apply a single sum-check protocol to a random linear combination of the below polynomials. For brevity, we describe the protocol as invoking c independent instances of sum-check.
6. $\mathcal{V} \leftrightarrow \mathcal{P}$: For $i = 1, \ldots, \alpha$, run a sum-check-based protocol for "grand products" ([Tha13, Proposition2] or [SL20, Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r_i'' \in \mathbb{F}^\ell, r_i''' \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_i''') \stackrel{?}{=} v_{E_i}$
   - $\dim_i(r_i''') \stackrel{?}{=} v_i$; $\mathsf{read\_counts}_i(r_i''') \stackrel{?}{=} v_{\mathsf{read\_counts}_i}$; and $\mathsf{final\_counts}_i(r_i'') \stackrel{?}{=} v_{\mathsf{final\_counts}_{\mathsf{row}}}$
7. $\mathcal{V}$: Check the equations hold with an oracle query to each of $E_i, \dim_i, \mathsf{read\_counts}_i, \mathsf{final\_counts}_i$.

Figure 9: Surge's polynomial IOP for proving that $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) T[y] = v$.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[BBB$^+$18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[BCC$^+$16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[BCG$^+$18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2018.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orru. Gemini: Elastic snarks for diverse environments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2022.

[BEG$^+$91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[BGH20] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo2, 2020. URL: `https://github.com/zcash/halo2`.

[BMM$^+$21] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2021.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2023.

[CDD$^+$03] Dwaine Clarke, Srinivas Devadas, Marten Van Dijk, Blaise Gassend, G. Edward, and Suh Mit. Incremental multiset hash functions and their application to memory integrity checking. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2003.

[CHM$^+$20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

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

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

[DGM21] Justin Drake, Ariel Gabizon, and Izaak Meckler. Checking univariate identities in linear time, 2021. `https://hackmd.io/@arielg/ryGTQXWri`.

[DP23] Benjamin E. Diamond and Jim Posen. Proximity testing with logarithmic randomness. Cryptology ePrint Archive, Paper 2023/630, 2023. `https://eprint.iacr.org/2023/630`.

[EFG22] Liam Eagen, Dario Fiore, and Ariel Gabizon. cq: Cached quotients for fast lookups. *Cryptology ePrint Archive*, 2022.

[EHB22] Youssef El Housni and Gautam Botrel. Edmsm: Multi-scalar-multiplication for recursive snarks and more. *Cryptology ePrint Archive*, 2022.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 186–194, 1986.

[GK22] Ariel Gabizon and Dmitry Khovratovich. flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. *Cryptology ePrint Archive*, 2022.

[GLS$^+$21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and post-quantum snarks for R1CS. Cryptology ePrint Archive, 2021.

[GW20a] Ariel Gabizon and Zachary Williamson. Proposal: The TurboPlonk program syntax for specifying SNARK programs, 2020.

[GW20b] Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. 2020.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. ePrint Report 2019/953, 2019.

[Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. *Cryptology ePrint Archive*, 2022.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2022.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 177–194, 2010.

[Lee21] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *Theory of Cryptography Conference*, pages 1–34. Springer, 2021.

[LFKN90] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, October 1990.

[Pip80] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.

[PK22] Jim Posen and Assimakis A Kattis. Caulk+: Table-independent lookup arguments. *Cryptology ePrint Archive*, 2022.

[PT12] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.

[PT13] Mihai Pătraşcu and Mikkel Thorup. Twisted tabulation hashing. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 209–228, 2013.

[RIS] RISC-V Foundation. The RISC-V instruction set manual, volume I: User-Level ISA, Document Version 20180801-draft. May 2017.

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

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

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

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

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

[Whi] Barry Whitehat. Lookup singularity. `https://zkresear.ch/t/lookup-singularity/65/7`.

[WTS+18] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2022.

[ZBK+22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. *Cryptology ePrint Archive*, 2022.

[ZGK+22] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. Baloo: Nearly optimal lookup arguments. *Cryptology ePrint Archive*, 2022.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

Additional Material

A Details on polynomial commitment schemes

In this section, we give more information about the properties and cost profiles of the polynomial commitment schemes in Table 2.2. Below, let $n = 2^\ell$, and assume that the prover knows all n evaluations of q over domain $\{0,1\}^\ell$, i.e., knows $\{(x, q(x)) \colon x \in \{0,1\}^\ell\}$.

- Hyrax [WTS$^+$18] is based on the hardness of the discrete logarithm problem. To commit to q, the prover performs $\sqrt{n}$ multiexponentiations each of size $\sqrt{n}$. The commitment size is $\sqrt{n}$ group elements. Evaluation proofs are also $\sqrt{n}$ group elements. To compute the evaluation proof, the prover performs $O(n)$ field operations and a multiexponentiation of size $\sqrt{n}$. Verifying the evaluation proof requires a multiexponentiation of size $\sqrt{n}$.
- Dory [Lee21] requires pairing-friendly groups and is based on the SXDH assumption. Its primary benefit over Bulletproofs is that verifying evaluation proofs can be done with just logarithmically many group operations. In addition, *computing* an evaluation proof requires $O(n)$ field operations and roughly $O(\sqrt{n})$ cryptographic work. Dory does use a transparent pre-processing phase for the verifier that requires $O(\sqrt{n})$ cryptographic work.
- In Brakedown, Orion, and Orion+ [GLS$^+$21, XZS22, CBBZ23], evaluation proofs are computed with $O(n)$ field operations and cryptographic hash evaluations. Commitments are just a single hash value. However, Brakedown proof sizes include $O(\sqrt{\lambda n})$ field elements, where λ is the security parameter. The verifier performs $O(\sqrt{\lambda n})$ field operations and also hashes this many field elements. Brakedown is also *field-agnostic*—it applies to polynomials defined over any sufficiently large field $\mathbb{F}$. Orion reduces verification costs to polylogarithmic via SNARK composition, but is not field-agnostic. Neither Brakedown nor Orion are homomorphic, but they are plausibly post-quantum secure. Orion+ [CBBZ23] reduces the proof size further to logarithmic (concretely under 10 KBs) but gives up transparency and post-quantum security in addition to field-agnosticism.

B Simple sparse polynomial commitment with logarithmic overhead

In this section, we describe a sparse polynomial commitment scheme that is suboptimal by a logarithmic factor. We include this merely for illustration, because this suboptimal scheme is substantially simpler than Spartan's scheme (Section 5).

Notation. For the remainder of this section, let $\widetilde{f}$ denote an ℓ-variate multilinear polynomial to be committed, with sparsity m in the Lagrange basis. Let $f \colon \{0,1\}^\ell \to \mathbb{F}$ denote the function with domain equal to the Boolean hypercube that $\widetilde{f}$ extends.

In this work, we only apply a sparse polynomial commitment scheme in a setting where (if the prover is honest)

$$f(x) \in \{0,1\} \text{ for all } x \in \{0,1\}^\ell. \tag{70}$$

We describe a commitment scheme that applies to multilinear extensions of functions of this form. This slightly simplifies the description of the scheme, and makes the bound on the prover runtime to compute the commitment slightly cleaner.[23]

Let $v_f \in \mathbb{F}^{m\ell}$ be the following "densified" description of f. Break v_f into m blocks each of length ℓ. Impose an arbitrary order on the set $S \coloneqq \{x \in \{0,1\}^\ell \colon f(x) \neq 0\}$, and let $x^{(i)}$ denote the ith element in S. Assign the ith block of v to be $x^{(i)} \in \{0,1\}^\ell$. In other words, v_f simply lists each x such that $f(x) \neq 0$.

[23] To clarify, the scheme as described in this section does *not* guarantee that the committed polynomial satisfies Equation (70). While it cannot be used by an honest prover to commit to arbitrary polynomials, it can always be used to commit to $\widetilde{f}$ if f satisfies Equation (70).

The commit phase. To commit to a sparse polynomial $\widetilde{f}$, we apply any desired dense polynomial commitment scheme to commit to the multilinear extension $\widetilde{v}_f$ of the vector v_f.

Evaluation proofs. As a warm-up, we begin with a conceptually simple high-level sketch of an evaluation proof procedure. We then specify full details of a more direct protocol with similar costs. The direct evaluation proof procedure involves a single application of the sum-check protocol.

Warm-up: a conceptually simple procedure (sketch). To reveal an evaluation $\widetilde{f}(r)$ for $r \in \mathbb{F}^{\ell}$, we apply any sum-check-based SNARK (e.g., Spartan, Brakedown, Orion, Libra, etc.) to the natural arithmetic circuit of size $O(m \cdot \ell)$ that takes as input the densified description v_f of f and outputs $\widetilde{f}(r)$. This circuit has a "uniform" wiring pattern that ensures that the verifier in any of these SNARKs will run in polylogarithmic time when applied to this circuit (without any pre-processing), plus the time to check a single evaluation proof from the dense polynomial commitment scheme applied to $\widetilde{v}_f$.

Complete description of an evaluation proof procedure via direct application of sum-check. Let us assume that m and ℓ are both powers of 2. If the prover is honest, then

$$\widetilde{f}(r) = \sum_{k \in \{0,1\}^{\log m}} \prod_{j \in \{0,1\}^{\log(\ell)}} \left(\widetilde{v}_f(k,j) r_j + (1 - \widetilde{v}_f(k,j))(1 - r_j)\right). \tag{71}$$

To compute Equation (71), we can apply the sum-check protocol to the polynomial g defined below:

$$g(k) = \prod_{j \in \{0,1\}^{\log(\ell)}} \left(\widetilde{v}_f(k,j) r_j + \left(1 - \widetilde{v}_f(k,j)\right)(1 - r_j)\right).$$

Observe that g has $\log m$ variables and degree ℓ in each of them, so the proof length of the sum-check protocol applied to g is $O(\ell \cdot \log m)$ field elements. At the end of the sum-check protocol, the verifier has to evaluate g at a random point $r' \in \mathbb{F}^{\log m}$. This can be done in $O(\ell)$ time given ℓ evaluations of $\widetilde{v}_f$, namely $\widetilde{v}_f(r', j)$ for each $j \in \{0,1\}^{\log(\ell)}$. Standard techniques can efficiently reduce these ℓ evaluations of $\widetilde{v}_f$ to a *single* evaluation of $\widetilde{v}_f$. Specifically, the prover is asked to send the entire $(\log(\ell))$-variate polynomial $h(y) = \widetilde{v}_f(r', y)$, i.e., the polynomial obtained from $\widetilde{v}_f$ by fixing the first $\log m$ variables to r'. This costs only $\ell + 1$ field elements in communication. The verifier picks a random point r'' and confirms that $h(r'') = \widetilde{v}_f(r', r'')$ with a single evaluation query to the committed polynomial $\widetilde{v}_f$. By the Schwartz-Zippel lemma, if $h(y) \neq \widetilde{v}_f(r', y)$, then with probability at least $1 - \log(1+\ell)/|\mathbb{F}|$, the verifier's check will fail.

Theorem 8. *The above protocol is an extractable polynomial commitment scheme for multilinear polynomials.*

Proof. Suppose that $\mathcal{P}$ is a prover that, with non-negligible probability, produces evaluation proofs that pass verification. By extractability of the dense polynomial commitment scheme used to commit to $\widetilde{v}_f$, there is a polynomial time algorithm $\mathcal{E}$ that produces a multilinear polynomial polynomial p that explains all of $\mathcal{P}$'s evaluation proofs, in the following sense. If $\mathcal{P}$ is able to, with non-negligible probability, produce an evaluation proof for the claim that the committed polynomial polynomial's evaluation at any input $(r', r'') \in \mathbb{F}^{\log m} \times \mathbb{F}^{\ell}$ equals value $v \in \mathbb{F}$, then $p(r', r'') = v$.

By soundness of the sum-check protocol, if the prover passes the verifier's checks with probability more than $O\left((\log m + \log(1+\ell))/|\mathbb{F}|\right)$ then v equals

$$\sum_{k \in \{0,1\}^{\log m}} \prod_{j \in \{0,1\}^{\log \ell}} \left(\widetilde{v}_f(k,j) r_j + (1 - \widetilde{v}_f(k,j))(1 - r_j)\right).$$

This is a multilinear polynomial in (r', r'').

□

Costs of the sparse polynomial commitment scheme. There are two sources of costs in the sparse polynomial commitment scheme above.

- One is applying the dense polynomial commitment scheme to commit to the multilinear extension $\widetilde{v}$ of a vector $v \in \{0,1\}^{m\ell}$, and later produce a single evaluation proof for $\widetilde{v}(r', r'')$ via this commitment scheme.

  Prover costs. If the dense polynomial commitment scheme used is Hyrax [WTS+18], Dory [Lee21], or BMMTV [BMM+21], the commitment can be computer with only $O(m\ell)$ group operations. Here, we exploit that the entries of v are al in $\{0,1\}$). Dory and BMMTV require pairing-friendly groups and also require the prover to compute a multi-pairing of length-$O(\sqrt{m\ell})$.

  Evaluation proofs for all three commitment schemes require $O(m \log n)$ field operations and roughly $O(\sqrt{m\ell})$ cryptographic work.

  Verifier costs. Hyrax's proofs consist of $O(\sqrt{m\ell})$ group elements, and the verifier must perform a multi-exponentiation of size $O(\sqrt{m\ell})$. Dory and BMMTV proofs consist of $O(\log(m\ell))$ elements of the target group $\mathbb{G}_t$, and the verifier performs $O(\log(m\ell))$ exponentiations/scalar-multiplications in $\mathbb{G}_t$.

- The other is the costs of the sum-check protocol, and the final step in reducing $1+\ell$ evaluations of $\widetilde{v}_f$ to a single evaluation. The verification costs of these two protocols is $O((\log m)\cdot \log \ell)$ field operations. Meanwhile, via standard techniques, the prover can be implemented with $O(m\ell)$ field operations in total across all rounds of the protocol.

C Additional details on the grand product argument

For completeness of exposition, we provide additional details on the grand product argument we use (Lines 6 and 10 of Figure 6) and its application in our context. Note that these details are identical to prior works [Set20, SL20, GLS+21].

Thaler's grand product argument [Tha13, Proposition 2] is simply an optimized application of the GKR interactive proof for circuit evaluation to a circuit computing a binary tree of multiplication gates. The prover in the interactive proof does a number of field operations that is linear in the circuit size, which in the application of the grand product argument in our lookup argument is $O(m)$. The verifier in the GKR protocol has to evaluate the MLE of the input vector to the circuit at a randomly chosen point r. In our applications of the grand product argument, the input to the circuit is either:

$$\begin{gathered}\{a\cdot\gamma^2+v\cdot\gamma+t-\tau\colon (a,v,t)\in \mathsf{WS}\},\\ \{a\cdot\gamma^2+v\cdot\gamma+t-\tau\colon (a,v,t)\in \mathsf{RS}\}\cup\{a\cdot\gamma^2+v\cdot\gamma+t-\tau\colon (a,v,t)\in S\},\\ \{a\cdot\gamma^2+v\cdot\gamma+t-\tau\colon (a,v,t)\in \mathsf{WS}'\},\end{gathered}$$

or

$$\{a\cdot\gamma^2+v\cdot\gamma+t-\tau\colon (a,v,t)\in \mathsf{RS}'\}\cup\{a\cdot\gamma^2+v\cdot\gamma+t-\tau\colon (a,v,t)\in S'\}.$$

For simplicity of notation, let us assume that $N^{1/c}=m$, and let $k=(k_1,\ldots,k_{\log m})$ be variables, and let us focus for illustration on the second case above. In this second case above, the multilinear extension of the input to the circuit is

$$\begin{gathered}g(k_0,k_1,\ldots,k_{\log m}) = k_0\cdot\left(\gamma^2\mathsf{row}(k)+\gamma E_{\mathsf{rx}}(k)+\mathsf{read\_counts}_{\mathsf{row}}(k)\right)+\\ (1-k_0)\cdot\left(\gamma^2\left(\sum_{i=1}^{\log N^{1/c}} 2^{i-1}\cdot k_i\right)+\gamma\cdot\widetilde{\mathsf{eq}}\left(k,r_x\right)+\mathsf{final\_counts}_{\mathsf{row}}(k)\right)-\tau.\end{gathered}$$

Indeed, by the definition of RS and S in Claim 2, the expression above is multilinear and agrees with the input to the circuit whenever $(k_0, \ldots, k_{\log m}) \in \{0,1\}^{\log m}$. Here, k_0 acts a selector bit—when $k_0 = 1$ (respectively, $k_0 = 0$), it indicates that $(k_1, \ldots, k_{\log m}$ index into the set RS' (respectively, S'). Hence, it must equal the unique multilinear extension of the input. The expression above can be evaluated at any point $(k_0, \ldots, k_{\log m}) \in \mathbb{F}^{1+\log m}$ in logarithmic time by the verifier, with one evaluation query to each of row, E_{rx}, $\mathsf{read\_counts}_{\mathsf{row}}$ and $\mathsf{final\_counts}_{\mathsf{row}}$. A similar expression holds in the other three cases above.

As described in Section 1.1.3, we propose to use Setty and Lee's grand product argument [SL20, Section 6], which reduces the proof size of Thaler's to $O(\log(m) \cdot \log\log m)$ at the cost of committing to an additional, say, $m/\log^3(m)$, field elements. The rough idea is that the prover cryptographically commits to the values of the gates at all layers of circuit (the binary-tree of multiplication gates), except for the $O(\log\log m)$ layers closest to the inputs. While the committed gates account for *most of the layers* of the circuit, they account for a tiny fraction of the *gates* in the circuit, as there are only $m/\log^3 m$ gates at these layers. This commitment enables the prover to apply a Spartan-like SNARK to the committed layers, resulting in just logarithmic communication cost (whereas Thaler's interactive proof applied to those layers would have communication cost $O(\log^2 n)$).

Then Thaler's protocol is used to handle the $O(\log\log m)$ layers that were not committed. The total communication cost of applying Thaler's protocol just to these layers is $O(\log(m) \cdot \log\log m)$.