Lasso: Unlocking the lookup singularity

Srinath Setty* Justin Thaler† Riad Wahby‡

Abstract

A lookup argument allows an untrusted prover to commit to a vector $a \in \mathbb{F}^m$ and prove that all entries of a reside in some predetermined table t. This paper introduces Lasso, a new family of lookup arguments that unlocks the so-called "lookup singularity".

The simplest member of the Lasso family is called Basic-Lasso. For m lookups into a table of size N, the Basic-Lasso prover commits to just $m + N$ field elements. Moreover, the committed field elements are *small*, meaning that, no matter how big the field $\mathbb{F}$ is, they are all in the set $\{0, \ldots, m\}$.

When using an appropriate multiexponentiation-based commitment scheme, this results in prover costs dominated by only $O(m + N)$ group *operations* (elliptic curve point additions). Furthermore, unlike prior lookup arguments, if the table t is structured (in a precise sense that we define), then no party needs to commit to the table t.

Other members of the Lasso family, when applied to structured tables, avoid the need for the prover to explicitly materialize table entries, only "paying" in runtime for table entries that are accessed by the lookup operations. This applies to tables commonly used to implement range checks, bitwise operations, 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 (e.g., of size 2^{128} or larger).

Specifically, for any integer parameter $c > 1$, Lasso's prover's dominant cost for structured tables is committing to $3 \cdot c \cdot m + c \cdot n^{1/c}$ field elements. Furthermore, all these field elements are "small", meaning they are in the set $\{0, \ldots, \max\{m, n^{1/c}, q\} - 1\}$, where $q = a$ is the maximum value in a.

We achieve these results with a perhaps surprising generalization of Spark, a time-optimal sparse polynomial commitment scheme in Spartan (CRYPTO 2020). We first provide a stronger security analysis for *Spark*. 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. 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. We then show that one can generalize Spark to directly support a lookup argument for both structured and unstructured tables, with performance characteristics noted above.

*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 **??**), 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* **??** *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 Lasso: A new lookup argument

We describe a new lookup argument, Lasso.[2] Lasso is based on a commitment scheme for sparse multilinear polynomials (which is itself based on the sum-check protocol).

STVS: May be worth calling Lasso as Surge and then call the generalized lasso as Lasso.

A stronger analysis of Spark, an optimal commitment scheme for sparse polynomials. 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.[3] 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.

SS: Explain how lookups can be checked with a matrix vector product before introducing Surge?

Surge, a generalization of Spark. We reinterpret 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.

[2] Lasso is short for LASSO-of-Truth: Lookup Arguments via the Surge Sparse-polynomial-commitments, including for Oversized Tables.

[3] 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 $n := 2^\ell$ 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.

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 $T_1, \ldots, T_\alpha$.

Lasso. Surge directly provides a lookup argument for tables with SOS structure. We call the resulting lookup argument Lasso, where *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 makes 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.

We highlight that Lasso has new and attractive costs when applied to small tables in addition to large ones. Specifically, by setting $c = 1$, the Lasso 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.[4] Lasso 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.

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

STVS: REVISIT. Bring back the summary of attractive features as comparison with related work.

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 $a = h(\mathsf{cm})$.
- 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

[4] 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 ?? for discussion).

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.

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[5]. 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.

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:

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

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

Indeed, one can easily check that the right hand side of Equation (1) 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 (1) is the unique multilinear polynomial extending eq. Equation (1) 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.[6]

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:*

$$\widetilde{f}(x_1, \ldots, x_\ell) = \sum_{w \in \{0,1\}^\ell} f(w) \cdot \chi_w(x_1, \ldots, x_\ell), \tag{2}$$

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

$$\chi_w(x_1, \ldots, x_\ell) \coloneqq \prod_{i=1}^{\ell} \left(x_i w_i + (1-x_i)(1-w_i)\right). \tag{3}$$

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 (2).

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:

$$H \coloneqq \sum_{b \in \{0,1\}^\ell} g(b). \tag{4}$$

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.

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

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 $\mathcal{A}$

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

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

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

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 $\mu \geq 1$,*

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

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

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

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.[7] 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

3.1 Background on the Spark sparse polynomial commitment scheme

Lasso's starting point is Spark, a sparse polynomial commitment scheme from Spartan [Set20] called Spark (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). In this paper, we prove that Spark remains secure even if that metadata is committed by an untrusted party (e.g., the prover), providing a standard commitment scheme for sparse polynomials.

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)

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

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.

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 $r = (r_1, \ldots, r_c) \in \left(\mathbb{F}^{\log m}\right)^c$. 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.[8]

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:

[8]This is also closely analogous to the behavior of tabulation hashing discussed earlier in the introduction, which is why we chose to highlight this example from algorithm design.

- 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 2. *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.*

3.2 Surge: A substantial 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 following: 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 $\sum_{i:\, u_i \neq 0} u_i \cdot t_i$.

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 $r = (r_1, \ldots, r_c) \in \left(\{0,1\}^{\log(N)/c}\right)^c$, i.e., break r into c pieces of equal size. Suppose that for every $r \in \{0,1\}^{\log N}$,

$$T[r] = g\left(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c]\right). \tag{5}$$

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
$$\{0, 1, \ldots, \max\{m, N^{1/c}\} - 1\},$$
and the remaining αm of them are elements of the subtables $T_1, \ldots, T_\alpha$. For many lookup tables T, these elements are themselves in the set $\{0, 1, \ldots, N^{1/c} - 1\}$.
- 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 (13) as having *Spark-only structure* (SOS).

4 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 1) 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 $N \coloneqq 2^\ell$ 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 1 (Special case of Theorem 2 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

$$D(r_x, r_y) = \sum_{(i,j) \in \{0,1\}^{\log \mathsf{M}} \times \{0,1\}^{\log \mathsf{M}} :\ D(i,j) \neq 0} D(i,j) \cdot \widetilde{eq}(i, r_x) \cdot \widetilde{eq}(j, r_y). \tag{6}$$

Claim 1. *Let* $\mathsf{to\text{-}field}$ *be the canonical injection from $\{0,1\}^{\log \mathsf{M}}$ to $\mathbb{F}$ and* $\mathsf{to\text{-}bits}$ *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 $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$.*

$$D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x) \cdot \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y). \tag{7}$$

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

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 3: 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 $x \in \{0,1\}^{2\log(\mathsf{M})}$.

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 $\mathsf{to\text{-}field}$ 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 $\mathsf{to\text{-}bits}$ 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 (6) and (7) 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 (7) 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 (6), and the other memory with the quantities $\widetilde{eq}(j, r_x)$, and then computed each term of Equation (7) 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 3, where the polynomial IOP assumes that the verifier has oracle access to the three $(\log m)$-variate multilinear polynomial oracles that encode D (namely $\mathsf{row}, \mathsf{col}, \mathsf{val}$).

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.[9] 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.

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.[10] $\mathsf{RS} = \{\}$, 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:

[9] 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.

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

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 $(k, v, t-1)$.[11] 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 3. *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 $\mathsf{WS} = \mathsf{RS} \cup S$.*

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.

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

After that, in addition to performing the polynomial IOP depicted earlier in the proof (Figure 3), 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*

$$\forall k \in \{0,1\}^{\log m},\ E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x). \tag{8}$$

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 (8) *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*
- $S' = \{(\mathsf{to\text{-}field}(j), \widetilde{eq}(j, r_y), \mathsf{final\_counts}_{\mathsf{col}}(j)) \colon j \in \{0,1\}^{\log(\mathsf{M})}\}$.

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 (8) 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 3 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 4, which assumes that the verifier has an oracle access to multilinear polynomial oracles that encode D (namely, $\mathsf{row}, \mathsf{col}, \mathsf{val}$)

//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 (7)) 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:
   - $\mathsf{val}(r_z) \stackrel{?}{=} v_{\mathsf{val}}$; and
   - $E_{\mathsf{rx}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$ 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 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:
   - $\widetilde{eq}(r_{\mathsf{M}}, r_x) \stackrel{?}{=} v_{eq}$
   - $E_{\mathsf{rx}}(r_m) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$
   - $\mathsf{row}(r_m) \stackrel{?}{=} v_{\mathsf{row}}$; $\mathsf{read\_counts}_{\mathsf{row}}(r_m) \stackrel{?}{=} v_{\mathsf{read\_counts}_{\mathsf{row}}}$; and $\mathsf{final\_counts}_{\mathsf{row}}(r_{\mathsf{M}}) \stackrel{?}{=} v_{\mathsf{final\_counts}_{\mathsf{row}}}$
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:
    - $\widetilde{eq}(r'_{\mathsf{M}}, r_y) \stackrel{?}{=} v'_{eq}$
    - $E_{\mathsf{ry}}(r'_m) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$
    - $\mathsf{col}(r'_m) \stackrel{?}{=} v_{\mathsf{col}}$; $\mathsf{read\_counts}_{\mathsf{col}}(r'_m) \stackrel{?}{=} v_{\mathsf{read\_counts}_{\mathsf{col}}}$; and $\mathsf{final\_counts}_{\mathsf{col}}(r'_{\mathsf{M}}) \stackrel{?}{=} v_{\mathsf{final\_counts}_{\mathsf{col}}}$
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 4: Evaluation procedure of our sparse polynomial commitment scheme.

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 **??** how to implement the prover in $O(cm)$ finite field operations.

Finally, to prove Theorem 1, 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.

Additional discussion and intuition. As previously discussed, the protocol in Figure 4 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 4 to commit to these values.

4.1 The general result

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

Equation (6) expressed the evaluation $\widetilde{D}(r_x, r_y)$ of a sparse $2\log(\mathsf{M})$-variate multilinear polynomial $\widetilde{D}$ as

$$\widetilde{D}(r_x, r_y) = \sum_{(i,j)\in\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}} D(i,j)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y). \tag{9}$$

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 (9):

$$\widetilde{D}(r_x, r_y, r_z) = \sum_{(i,j,k)\in\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}} D(i,j,k)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k, r_z). \tag{10}$$

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 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 $(c \log \mathsf{M})$*-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* $O\left(c\,(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M}))\right)$*;*

– *the running time of the verifier to verify a polynomial evaluation is* $O\left(c\,(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M}))\right)$*;*

– *the proof size is* $O\left(c\,(\mathsf{p}(m) + \mathsf{p}(\mathsf{M}))\right)$*.*

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 2 (i.e., prover and verifier costs for verifying polynomial evaluation do not grow with c).

4.2 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 $\mathsf{to\text{-}field}(k) = \sum_{j=1}^{\log m} 2^{j-1} \cdot k_j$. Hence, in Equation (7) of Claim 1, $\mathsf{to\text{-}bits}(\mathsf{row}(k))$ is simply k.[12] This means that $E_{\mathsf{rx}}(k) = \widetilde{\mathsf{eq}}(k, r_x)$, which the verifier can evaluate on its own in logarithmic time. With this fact in hand, the prover does not commit to E_{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 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 5. The prover commits to $3c$ dense $(\log(m))$-variate multilinear polynomials, called $\mathrm{dim}_1, \ldots, \mathrm{dim}_c$ (the analogs of the row and col polynomials of Section 4), $E_1, \ldots, E_c$, and $\mathsf{read\_counts}_1, \ldots, \mathsf{read\_counts}_c$, as well as c dense multilinear polynomials in $\log(N^{1/c}) = \log(N)/c$ variables, called $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_c$. Each dim_i is purported to be the memory cell from the i'th memory that the sparse polynomial evaluation

[12] 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 $\mathbb{F}^{\log M} = \mathbb{F}^{(1/2)(\log m + \log n)}$.

algorithm of Section 3.1 reads at each of its m timesteps, $E_1, \ldots, E_c$ the values returned by those reads, and $\mathsf{read\_counts}_1, \ldots, \mathsf{read\_counts}_c$ the associated counts. $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_c$ 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 $\mathsf{read\_counts}_1, \ldots, \mathsf{read\_counts}_c$ each map $\{0,1\}^{\log m}$ to $\{0, \ldots, m-1\}$ and $\mathsf{final\_counts}_1, \ldots, \mathsf{final\_counts}_c$ 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 $\mathsf{read\_counts}_i$ and $\mathsf{final\_counts}_i$ can be larger than j.

5 Surge: A Generalization of Spark

The technical core of the SuperLasso lookup argument is a generalization of Spark we call Surge. Recall (Section 4, Figure 5) 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 (**??**)), after committing to $\widetilde{M}$, the SuperLasso verifier picks a random $r \in \mathbb{F}^{\log m}$ and seeks to confirm that

$$\sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r, j) \cdot t(j) = \widetilde{a}(r). \tag{11}$$

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 (11) 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 (11) 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 (11) *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 (11). Essentially, the proof *proves* that the prover correctly ran a (very efficient) algorithm for evaluating the left hand side of Equation (11).

A roughly $O(\alpha m)$-time algorithm for computing the left hand side of Equation (11). Recalling Equation (**??**),

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

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 (11) equals

$$\sum_{i \in \{0,1\}^{\log m}} \chi_i(r) \cdot T[\mathsf{nz}(i)]. \tag{12}$$

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

$$T[r] = g\left(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c]\right). \tag{13}$$

//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 4. $\dim_i$ 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 10) 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 $g(k) \coloneqq \widetilde{\mathsf{eq}}(r, k) \cdot \prod_{i=1}^{c} E_i(k)$, 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:
   - $E_i(r_z) \stackrel{?}{=} v_{E_i}$ 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:
   - $E_i(r'''_i) \stackrel{?}{=} v_{E_i}$
   - $\dim_i(r'''_i) \stackrel{?}{=} v_i$; $\mathsf{read\_counts}_i(r'''_i) \stackrel{?}{=} v_{\mathsf{read\_counts}_i}$; and $\mathsf{final\_counts}_i(r''_i) \stackrel{?}{=} v_{\mathsf{final\_counts}_{\mathsf{row}}}$
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 5: Evaluation procedure of our sparse polynomial commitment scheme, optimized for its application to M in Lasso.

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 (12) equals

$$\sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i, r) \cdot g\left(T_1[\mathsf{nz}_1(i)], \ldots, T_k[\mathsf{nz}_1(i)], T_{k+1}[\mathsf{nz}_2(i)], \ldots, T_{2k}[\mathsf{nz}_2(i)], \ldots, T_{\alpha-k+1}[\mathsf{nz}_c(i)], \ldots, T_\alpha[\mathsf{nz}_c(i)]\right). \tag{14}$$

The algorithm to compute Expression (14) 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 $\mathrm{dim}_1, \ldots, \mathrm{dim}_c$, 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 (12). The prover does so by proving it ran the aforementioned algorithm for evaluating Expression (14). Following the memory-checking procedure in Section 4, with each table T_i: $i = 1, \ldots, \alpha$ 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

$$\sum_{j \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, j) \cdot g\left(E_1(j), \ldots, E_\alpha(j)\right). \tag{15}$$

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 $T_i[\mathrm{dim}_i(j)]$ 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 5). 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.

- 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 $\mathrm{dim}_1, \ldots, \mathrm{dim}_c$ (see Figure 7 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 7), allowing $\mathcal{P}$ to prove that $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) T[y] = \widetilde{a}(r)$.

Figure 6: 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.

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 (14)).

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

1. $\mathcal{P} \to \mathcal{V}$: 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 $h(k) \coloneqq \widetilde{\mathsf{eq}}(r, k) \cdot g(E_1(k), \ldots, E_\alpha(k))$, 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. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
   //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.
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 $\mathsf{final\_counts}_i(r_i'') \stackrel{?}{=} v_{\mathsf{final\_counts}_i}$
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 7: Surge's polynomial IOP for proving that $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) T[y] = v$.

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 (15) 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 (15) 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 (15) consists of $\log m$ rounds in which the prover sends a univariate polynomial of degree at most $1 + \alpha$ in each round. Hence, the prover sends $O(ck \log m)$ 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.

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 $j_1 = \dim_1(i)$, $\ldots$, $j_c = \dim_c(i)$.

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.

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 4).

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)

$$f(x) \in \{0,1\} \text{ for all } x \in \{0,1\}^\ell. \tag{16}$$

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.[13]

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 $f(x) \neq 0$.

[13] To clarify, the scheme as described in this section does *not* guarantee that the committed polynomial satisfies Equation (16). 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 (16).

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

$$\widetilde{f}(r) = \sum_{k \in \{0,1\}^{\log m}} \prod_{j \in \{0,1\}^{\log(\ell)}} (\widetilde{v}_f(k,j) r_j + (1 - \widetilde{v}_f(k,j))(1 - r_j)) \,. \tag{17}$$

To compute Equation (17), we can apply the sum-check protocol to the polynomial g defined below:

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

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 3. *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

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

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 4) 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:

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

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

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

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 **??**, 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 $O(\log(m) \cdot \log\log m)$.