Unlocking the Lookup Singularity with Lasso

Srinath Setty[1(✉)], Justin Thaler[2], and Riad Wahby[3]

[1] Microsoft Research, Redmond, USA
srinath@microsoft.com
[2] a16z crypto research and Georgetown University, Washington DC, USA
[3] Carnegie Mellon University, Pittsburgh, USA

Abstract. This paper introduces Lasso, a new family of lookup arguments, which allow 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 $t \in \mathbb{F}^n$. Lasso's performance characteristics unlock the so-called "lookup singularity". Lasso works with any multilinear polynomial commitment scheme, and provides the following efficiency properties.

- For m lookups into a table of size n, Lasso's 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 a multiexponentiation-based commitment scheme, this results in the prover's costs dominated by only $O(m+n)$ group *operations* (e.g., elliptic curve point additions), plus the cost to prove an evaluation of a multilinear polynomial whose evaluations over the Boolean hypercube are the table entries. This represents a significant improvement in prover costs over prior lookup arguments (e.g., plookup, Halo2's lookups, logUp).
- Unlike all prior lookup arguments, if the table t is structured (in a precise sense that we define), then no party needs to commit to t, enabling the use of much larger tables than prior works (e.g., of size 2^{128} or larger). Moreover, Lasso's prover only "pays" 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. Specifically, for any integer parameter $c > 1$, Lasso's prover's dominant cost 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 is the maximum value in any of the sub-tables that collectively capture t (in a precise manner that we define).

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.

M. Joye and G. Leander (Eds.): EUROCRYPT 2024, LNCS 14656, pp. 180–209, 2024.
https://doi.org/10.1007/978-3-031-58751-1_7

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* [25,27], 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 paper describes techniques to enable such a table lookup without requiring

[1] As we explain later (Remark 1.2), 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 table such as this to 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 [6,17,18]. 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 [6], 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 (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 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 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.

A flurry of works (Caulk [37], Caulk+ [26], flookup [16], Baloo [38], and cq [14]) 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 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 untenable. 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 an additional pre-processing work of $O(N \log N)$ 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 $N < 2^{30}$.[2]

Indexed Lookup Arguments. Definition 1 is a standard formulation of lookup arguments in SNARKs (e.g., see [38]). 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, for reasons that will become apparent shortly (§1.3), we consider a variant notion to be equally natural. We refer to this variant as an *indexed lookup argument* (and refer to the standard variant in Definition 1 as an unindexed lookup argument.) In an indexed lookup argument, in addition to a commitment to $a \in \mathbb{F}^m$, the verifier is handed a commitment to

[2] See, for example, https://setup.aleo.org/stats.

a second vector $b \in \mathbb{F}^m$. The prover claims that for all $i = 1, \ldots, m$, $a_i = t_{b_i}$. We refer to a as the vector of *looked-up values*, and b as the vector of *indices*.

Definition 2 (Statement proven in an indexed lookup argument). *Given commitment* cm_a *and* cm_b*, and a public array* 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 a commitment* cm_t*, the prover knows an opening* $a = (a_0, \ldots, a_{m-1}) \in \mathbb{F}^m$ *of* cm_a *and* $b = (b_0, \ldots, b_{m-1}) \in \mathbb{F}^m$ *of* cm_b *such that for each* $i = 0, \ldots, m-1$*,* $a_i = T[b_j]$*, where* $T[b_j]$ *is short hand for the* b_j*'th entry of* t*.*

Any indexed lookup argument can easily be turned into an unindexed lookup argument: the unindexed lookup argument prover simply commits to a vector b such that $a_i = T[b_j]$ for all i, and then applies the indexed lookup argument to prove that indeed this holds. There is also a generic transformation that turns any unindexed lookup argument into an indexed one, at least in fields of large enough characteristic (see [32]). However, the protocols we describe in this work directly yield indexed lookup arguments, without invoking this transformation. Accordingly, our primary focus in this work is on indexed lookup arguments.

1.1 Lasso: A New Lookup Argument

Lasso's starting point is a polynomial commitment scheme for sparse multilinear polynomials. In particular, Lasso builds on Spark, an optimal polynomial commitment scheme for sparse multilinear polynomials from Spartan [28].

Lasso can be instantiated with any multilinear polynomial commitment scheme. Furthermore, Lasso can be used with any SNARK, including those that prove R1CS or Plonkish satisfiability. This is particularly seamless for SNARKs that have the prover commit to the witness using a multilinear polynomial commitment scheme. This includes many known prover-efficient SNARKs [10,20,28,31,36]. If a SNARK does not natively use multilinear polynomial commitments (e.g., Marlin [11], Plonk [19]), then one would need an auxiliary argument that the commitment cm_a used in Lasso is a commitment to the multilinear extension of the vector of all lookups performed in the SNARK.

Below, we provide an overview of Lasso's technical components.

(1) A stronger analysis of Spark, an optimal commitment scheme for sparse polynomials. A sparse polynomial commitment allows an untrusted prover to cryptographically commit to a *sparse* multilinear polynomial g and later provide a requested evaluation $g(r)$ along with a proof that the provided value is indeed equal to the committed polynomial's evaluation at r. Crucially, we require that the the prover's runtime depends only on the sparsity of the polynomial.[3] Spartan [28] provides such a commitment scheme, which it calls Spark.

[3] For multilinear polynomials, m-sparse refers to polynomials $g\colon \mathbb{F}^\ell \to \mathbb{F}$ in ℓ variables such that $g(x) \neq 0$ for at most m values of $x \in \{0,1\}^\ell$. That is, g has at most m non-zero coefficients in the so-called multilinear Lagrange 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 g is a *dense* polynomial.

Spartan assumed that certain metadata associated with the sparse polynomial is committed honestly, which was sufficient for its purposes. But, as we see later, Lasso requires an *untrusted* prover to commit to sparse polynomials (and the associated metadata).

A naive extension Spark to handle a maliciously committed metadata incurs concrete and asymptotic overheads, which is undesirable. Nevertheless, we prove that Spark in fact satisfies a stronger security property without any modifications (i.e., it is secure even if the metadata is committed by a potentially malicious party). This provides the first "standard" sparse polynomial commitment scheme with optimal prover costs, a result of independent interest. Furthermore, we specialize Spark for Lasso's use to obtain concrete efficiency benefits.

(2) Surge: A generalization of Spark. We reinterpret Spark sparse polynomial commitment scheme as a technique for computing the inner product of an m-sparse committed vector of length N with a dense—but highly structured—lookup table of size N (the table is represented as a vector of size N). Specifically, in the sparse polynomial commitment scheme, the table consists of all $(\log N)$-variate Lagrange basis polynomials evaluated at a specific point $r \in \mathbb{F}^{\log N}$. Furthermore, this table is a *tensor product* of $c \geq 2$ smaller tables, each of size $N^{1/c}$ (here, c can be set to any desired integer in $\{1, \ldots, \log N\}$). We further observe that many other lookup tables can similarly be decomposed has product-like expressions of $O(c)$ tables of size $N^{1/c}$, and that Spark extends to support all such tables.

Exploiting this perspective, we describe Surge, a generalization of Spark that allows an untrusted prover to commit to any sparse vector and establish the sparse vector's inner product with any dense, structured vector. We refer to the structure required for this to work as *Spark-only structure* (SOS). We also refer to this property as *decomposability*. In more detail, an SOS table T is one that can be decomposed into $\alpha = O(c)$ "sub-tables" $\{T_1, \ldots, T_\alpha\}$ of size $N^{1/c}$ satisfying the following properties. First, any entry $T[j]$ of T can be expressed as a simple expression of a corresponding entry into each of $T_1, \ldots, T_\alpha$. Second, the so-called *multilinear extension polynomial* of each T_i can be evaluated quickly (for any such table, we call T_i *MLE-structured*, where MLE stands for multilinear extension). For example, as noted above, the table T arising in Spark itself is simply the tensor product of MLE-structured sub-tables $\{T_1, \ldots, T_\alpha\}$, where $\alpha = c$.

(3) Lasso: A lookup argument for SOS tables and small/unstructured tables. We observe that Surge directly provides a lookup argument for tables with SOS structure. We call the resulting lookup argument Lasso. Lasso has the important property that *all* field elements committed by the prover are "small", meaning they are in the set $\{0, 1, \ldots, \max\{m, N^{1/c}, q\} - 1\}$, where q is such that $\{T_1, \ldots, T_\alpha\}$ all have entries in the set $\{0, 1, \ldots, q - 1\}$. As elaborated upon shortly (Sect. 1.2), this property of Lasso has substantial implications for prover efficiency.

Lasso has new and attractive costs when applied to small and unstructured tables in addition to large SOS ones. Specifically, by setting $c = 1$, the Lasso prover commits to only about $m + N$ 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.[4,5] 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. For $c > 1$, the number of field elements that the Lasso prover commits to is $3cm + \alpha \cdot N^{1/c}$.

(4) GeneralizedLasso: Beyond SOS and small/unstructured tables. Finally, we describe a lookup argument that we call GeneralizedLasso, which applies to any MLE-structured table, not only decomposable ones.[6] The main disadvantage of GeneralizedLasso relative to Lasso is that cm out of the $3cm+cN^{1/c}$ field elements committed by the GeneralizedLasso prover are random rather than small. The proofs are also somewhat larger, as GeneralizedLasso involves one extra invocation of the sum-check protocol compared to Lasso.

GeneralizedLasso is reminiscent of a sum-check based SNARK (e.g., Spartan [28]) and is similarly built from a combination of the sum-check protocol and the Spark sparse polynomial commitment scheme. There are two key differences: (1) In GeneralizedLasso, the (potentially adversarial) prover commits to a sparse polynomial, rather than an honest "setup algorithm" committing to a sparse polynomial in a preprocessing step in the context of Spartan (where the sparse polynomial encodes the circuit or constraint system of interest); and (2) invoking the standard linear-time sum-check protocol [13,24,33] makes the prover incur costs linear in the *table size* rather than the number of lookups. To address (1), we invoke our stronger security analysis of Spark. To address (2), we introduce a new variant of the sum-check protocol tailored for our setting, which we refer to as the *sparse-dense* sum-check protocol. Conceptually, GeneralizedLasso can be viewed as using the sparse-dense sum-check protocol to reduce lookups into any MLE-structured table into lookups into a decomposable table (namely, a certain lookup table arising within the Spark polynomial commitment scheme).

Additional discussion of the benefits and costs of GeneralizedLasso relative to Lasso can be found in the full version of this paper [32].

1.2 Additional Discussion of Lasso's Costs

Polynomial Commitments and MSMs. As indicated above, a central component of most SNARKs is a cryptographic protocol called a *polynomial commitment*

[4] Lasso makes blackbox use of any so-called grand product argument. If using the grand product argument from [30, Section 6], a low-order number, say at most $O(m/\log^3 m)$, of large field elements need to be committed (see the full version of this paper [32] for discussion).

[5] If Lasso is used as an indexed lookup argument, the prover commits to $m + N$ field elements. If used as an unindexed lookup argument, the number can increase to $2m + N$ because in the unindexed setting one must "charge" for the prover to commit to the index vector $b \in \mathbb{F}^m$.

[6] In fact, GeneralizedLasso applies to any table with *some* low-degree extension, not necessarily its multilinear one, that is evaluable in logarithmic time.

scheme. Such a scheme allows an untrusted prover to succinctly commit to a polynomial p and later reveal an evaluation $p(r)$ for a point r chosen by the verifier (the prover will also return a *proof* that the claimed evaluation is indeed equal to the committed polynomial's evaluation at r). In Lasso, the bottleneck for the prover is the polynomial commitment scheme.

Many popular polynomial commitments are based on multiexponentiations (also known as multi-scalar multiplications, or MSMs). This means that the commitment to a polynomial p (with n coefficients $c_0, \ldots, c_{n-1}$ over an appropriate basis) is $\prod_{i=0}^{n-1} g_i^{c_i}$, for some public generators $g_1, \ldots, g_n$ of a multiplicative group $\mathbb{G}$. Examples include KZG [21], IPA [5,8], Hyrax [35], and Dory [23].[7]

The naive MSM algorithm performs n group exponentiations and n group multiplications (note that each group exponentiation is about 400× slower than a group multiplication). But Pippenger's MSM algorithm saves a factor of about $\log(n)$ relative to the naive algorithm. This factor can be well over 10× in practice.

Working Over Large Fields, But Committing to Small Elements. If all exponents appearing in the multiexponentiation are "small", one can save another factor of 10 relative to applying Pippenger's algorithm to an MSM involving random exponents. This is analogous to how computing $g_i^{2^{16}}$ is 10× faster than computing $g_i^{2^{160}}$: the first requires 16 squaring operations, while the second requires 160 such operations. In other words, if one is promised that all field elements (i.e., exponents) to be committed via an MSM are in $\{0, 1, \ldots, K\} \subset \mathbb{F}$, the number of group operations required to compute the MSM depends only on K and not on the size of $\mathbb{F}$.[8]

Quantitatively, if all exponents are upper bounded by some value K, with $K \ll n$, then Pippenger's algorithm only needs (about) one group *operation* per term in the multiexponentiation.[9] More generally, with any MSM-based commitment scheme, Pippenger's algorithm allows the prover to commit to roughly $k \cdot \log(n)$-bit field elements (meaning field elements in $\{0, 1, \ldots, n\}$) with only k group operations per committed field element.

Polynomial Evaluation Proofs. In any SNARK or lookup argument, the prover not only has to commit to one or more polynomials, but also reveal to the verifier an evaluation of the committed polynomials at a point of the verifier's choosing. This requires the prover to compute a so-called evaluation proof, which establishes that the returned evaluation is indeed consistent with the committed polynomial. For some polynomial commitment schemes, such as Bulletproofs/IPA [5,8], producing evaluation proofs is quite slow and this cost can

[7] In Hyrax and Dory, the prover does $\sqrt{n}$ MSMs each of size $\sqrt{n}$.

[8] Of course, the cost of each group operation depends on the size of the group's base field, which is closely related to that of the scalar field $\mathbb{F}$. However, the *number* of group operations to compute the MSM depends only on K, not on $\mathbb{F}$.

[9] To be very precise, if $K \leq n$, then Pippenger's algorithm performs only $(1 + o(1))n$ group operations.

bottleneck the prover. However, for others, evaluation proof computation is a low-order cost [2,35].

Moreover, evaluation proofs exhibit excellent batching properties (whereby the prover can commit to many polynomials and only produce a single evaluation proof across all of them) [4,7,22]. So in many contexts, computing opening proofs is not a bottleneck even when using a scheme such as Bulletproofs/IPA.

For all of the above reasons, our accounting of prover cost in this work generally ignores the cost of polynomial evaluation proofs.

Summarizing Lasso's Prover Costs. Based on the above accounting, Lasso's prover costs when applied to a lookup table T can be summarized as follows.

- Setting the parameter $c = 1$, the Lasso prover commits to just $m + N$ field elements (using any multilinear polynomial commitment scheme), all of which are in $\{0, \ldots, m\}$.[10] Using an MSM-based commitment scheme, this translates to very close to $m + N$ group operations.
- For $c > 1$, the Lasso prover applied to any decomposable table commits to $3cm + \alpha N^{1/c}$ field elements, all of which are in the set $\{0, \ldots, \max\{m, N^{1/c}, q\} - 1\}$, where q is the largest value in any of the α sub-tables $T_1, \ldots, T_\alpha$. This cost accounting does not "charge" the prover for committing a of lookup results. We do so because the natural formulation of lookup arguments as a self-contained problem (Definition 2) considers the commitment to a to be part of the problem statement
- The GeneralizedLasso prover applies to any MLE-structured table, and commits to the same number of field elements as the Lasso prover, but cm of them are random field elements, instead of small ones.

In all cases above, no party needs to cryptographically commit to the table T or subtables $T_1, \ldots, T_\alpha$, so long as they are MLE-structured. The full version of this paper [32] compares these costs with those of existing lookup arguments.

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 [1] (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. Jolt shows that for each of the RISC-V instructions, the resulting table has the structure that we require to apply Lasso. 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.

[10] In fact, for any $k \geq 1$, at most m/k of these field elements are larger than k.

2 Technical Overview

Suppose that the verifier has a commitment to a table $t \in \mathbb{F}^n$ as well as a commitment to another vector $a \in \mathbb{F}^m$. Suppose that a prover wishes to prove that all entries in a are in the table t. A simple observation in prior works [37,38] is that the prover can prove that it knows a sparse matrix $M \in \mathbb{F}^{m \times n}$ such that for each row of M, only one cell has a value of 1 and the rest are zeros and that $M \cdot t = a$, where $\cdot$ is the matrix-vector multiplication.[11] This turns out to be equivalent, up to negligible soundness error, to confirming that

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

for an $r \in \mathbb{F}^{\log m}$ chosen at random by the verifier. Here, $\widetilde{M}$, $\widetilde{a}$ and $\widetilde{t}$ are the so-called *multilinear extension polynomials* (MLEs) of M, t, and a (see Thaler [34]).

Lasso proves Eq. (1) by having the prover commit to the sparse polynomial $\widetilde{M}$ using Spark and then prove the equation directly with a generalization of Spark called Surge. This provides the most efficient lookup argument when either the table t is "decomposable" (we discuss details of this below), or when t is unstructured but small. It turns out most tables that occur in practice (e.g., the ones that arise in Jolt are decomposable). When t is not decomposable, but still structured, a generalization of Lasso, which we refer to as GeneralizedLasso, proves Eq. (1) using a combination of a new form of the sum-check protocol (which we refer to as the sparse-dense sum-check protocol) and the Spark polynomial commitment scheme. We defer further details of GeneralizedLasso to the full version of this paper [32].

2.1 Starting Point: Spark Sparse Polynomial Commitment Scheme

Lasso's starting point is Spark, an optimal sparse polynomial commitment scheme from Spartan [28]. It allows an untrusted prover to prove evaluations of a sparse multilinear polynomial with costs proportional to the size of the dense representation of the sparse multilinear polynomial. Spartan established security of Spark under the assumption that certain metadata associated with a sparse polynomial is committed honestly, which sufficed for its application in the context of Spartan. In this paper, perhaps surprisingly, 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 unique dense representation of the sparse polynomial g,

[11] Lasso's approach to prove $M \cdot t = a$ deviates significantly from the approaches in Baloo [38] and Caulk [37] despite Lasso starting with the same observation. In particular, Lasso's approach originates in Spartan [28], a work that predates Baloo and Caulk. Moreover, if one only demands a quasilinear prover time rather than linear, one can prove that $M \cdot t = a$ via "Spark-naive" [28, §7.1], even when M is committed by an untrusted prover.

using any polynomial commitment scheme for "dense" (multilinear) polynomials. The dense representation of g is effectively a list of all of the monomials of g 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 elsewhere [32,34]).

When the verifier requests an evaluation $g(r)$ of the committed polynomial g, 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 dense representation of g, and outputs $g(r)$ in $O(c \cdot m)$ time. Spark amounts to the bespoke SNARK establishing that the prover correctly ran this sparse-polynomial-evaluation algorithm on the committed description of g. Note that this perspective on Spark is somewhat novel, though it is partially implicit in the scheme itself and in an exposition of [34, Section 16.2].

A Time-Optimal Algorithm for Evaluating a Multilinear Polynomial of Sparsity m. We first describe a naive solution and then describe an optimal solution in Spark. Note that Spark provides a time-optimal algorithm when c is a constant.

A Naive Solution. Consider an algorithm that iterates over each Lagrange basis polynomials specified in the dense representation, evaluates that basis polynomial at r, multiplies by the corresponding coefficient, and adds the result to the evaluation. Unfortunately, a naive evaluation of a $(\log N)$-variate Lagrange basis polynomial at r takes $O(\log N)$ time, resulting in a total runtime of $O(m \cdot \log N)$.

Eliminating the Logarithmic Factor. The key to achieving time $O(c \cdot m)$ is to ensure that each Lagrange basis polynomial can be evaluated in $O(c)$ time. This is done as follows. This procedure is reminiscent of Pippenger's algorithm for multiexponentiation, with m being the size of the multiexponentiation, and Lagrange basis polynomials with non-zero coefficients corresponding to exponents.

Decompose the $\log N = c \cdot \log m$ 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 now-standard 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 $O(c \cdot m)$ total time, one can evaluate *all* m of these basis polynomials at each r_i, storing the results in a (write-once) memory M.

Given M, the time-optimal 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.[12] Note that we chose to decompose the $\log N$ variables into c blocks of length $\log m$ (rather than more,

[12] This is also closely analogous to the behavior of tabulation hashing discussed earlier in §1, which is why we chose to highlight this example from algorithm design.

smaller blocks, or fewer, bigger blocks) to balance the runtime of the two phases of the algorithm, namely:

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

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

How the Spark Prover Proves it Correctly Ran the Above Time-Optimal Algorithm. To enable an untrusted prover to efficiently prove that it correctly ran the above algorithm to compute an evaluation of a sparse polynomial g at r, Spark uses *offline memory checking* [3] to prove read-write consistency. Furthermore, the contents of the memory is determined succinctly by r, so the verifier does not need any commitments to the contents of the memory. Spark effectively forces the prover to commit to the "execution trace" of the algorithm (which has size $O(c \cdot m)$, because the algorithm runs in time $O(c)$ for each of the m Lagrange basis polynomials with non-zero coefficient) plus $c \cdot N^{1/c} = O(c \cdot m)$. The latter term arises because at the end of m operations, the offline memory-checking technique requires the prover to supply certain access counts indicating the number of times a particular memory location was read during the course of the protocol. Moreover, note that this memory has size $c \cdot N^{1/c}$ if the algorithm breaks the $\log N$ variables into c blocks of size $\log(N)/c$. As we will see later, this is why Lasso's prover cryptographically commits to $3 \cdot c \cdot m + c \cdot N^{1/c}$ field elements.

Remark 1 The cost incurred by Spark's prover to "replay" to provide access counts 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 aforementioned cost in the offline memory checking is reused across all k sparse polynomials.

2.2 Surge: A Generalization of Spark

Re-imagining 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 g over the Lagrange basis, $N = 2^\ell$, m is the number of evaluation points over the Boolean hypercube $x \in \{0,1\}^\ell$ such that $g(x) \neq 0$, and u is the vector of evaluations of g at all evaluation points over the hypercube $\{0,1\}^\ell$.

An evaluation query to g 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.

Spark → Surge. To obtain Surge from Spark, we critically examine the type of structure in t that is exploited by Spark, and introduce Surge as a natural generalization of Spark that supports any table t with this structure. More importantly, we observe that many lookup tables critically important in practice (e.g., those that arise in Jolt) exhibit this structure.

In more detail, the Surge prover establishes that it correctly ran a natural $O(c \cdot m)$-time algorithm for computing $\langle u, t\rangle$. This algorithm is a natural analog of the sparse polynomial evaluation algorithm described in Sect. 2.1: it iterates over every non-zero entry u_i of u, quickly computes $t_i = T[i]$ by performing one lookup into each of $O(c)$ "sub-tables" 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, this algorithm takes just $O(c \cdot m)$ time to compute the desired 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 $\forall r \in \{0,1\}^{\log N}$,

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

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 $3 \cdot \alpha \cdot m + \alpha \cdot N^{1/c}$ field elements, where $2 \cdot \alpha \cdot m + \alpha \cdot N^{1/c}$ of the committed elements are in the set $\{0, 1, \ldots, \max\{m, N^{1/c}\} - 1\}$, and the remaining $\alpha \cdot m$ of them are elements of the sub-tables $T_1, \ldots, T_\alpha$. 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 $O(k \cdot \alpha N^{1/c}) = O(b \cdot c \cdot N^{1/c})$ field operations. In many cases, the factor of b in the number of prover field operations can be removed.

We refer to tables that can be decomposed into sub-tables of size $N^{1/c}$ as per Eq. (2) as having *Spark-only structure* (SOS), or being *decomposable*.

3 A Stronger Analysis of Spark

We prove a substantial strengthening of a result from Spartan [28, Lemma 7.6]. In particular, we prove that in Spartan's sparse polynomial commitment scheme, which is called Spark, one does not need to assume that certain metadata associated with a sparse polynomial is committed honestly (in the case of Spartan, the metadata is committed by the setup algorithm, so it was sufficient for its purposes). We thereby obtain the first "standard" polynomial commitment scheme with prover costs *linear* in the number of non-zero coefficients. We prove this result without any substantive changes to Spark.

For simplicity, we 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. [20]. It is natural for the reader to conceptualize the Spark sparse polynomial commitment scheme as a bespoke SNARK for a prover to prove it correctly ran the sparse $(\log N)$-variate multilinear polynomial evaluation algorithm described in Sect. 2.1 using c memories of size $N^{1/c}$.

3.1 A (slightly) Simpler Result: $c = 2$

We first prove a special case of the 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 for $c = 2$ memories of size $N^{1/2}$.

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

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

multilinear polynomials, and c many dense $(\log(N^{1/c}))$-variate polynomials. We begin with $c = 2$ 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 $\mathsf{M} > 0$ and WLOG a power of 2): (1) the size of the commitment is $\mathsf{c}(\mathsf{M})$; (2) the running time of the commit algorithm is $\mathsf{tc}(\mathsf{M})$; (3) the running time of the prover to prove a polynomial evaluation is $\mathsf{tp}(\mathsf{M})$; (4) the running time of the verifier to verify a polynomial evaluation is $\mathsf{tv}(\mathsf{M})$; and (5) 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: (1) the size of the commitment is $7\mathsf{c}(m) + 2\mathsf{c}(\mathsf{M})$; (2) the running time of the commit algorithm is $O(\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))$; (3) the running time of the prover to prove a polynomial evaluation is $O(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M}))$; (4) the running time of the verifier to verify a polynomial evaluation is $O(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M}))$; and (5) 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, we can write

$$D(r_x, r_y) = \sum_{(i,j) \in \{0,1\}^{\log \mathsf{M}} \times \{0,1\}^{\log \mathsf{M}} :\ D(i,j) \neq 0} D(i,j) \cdot \widetilde{eq}(i, r_x) \cdot \widetilde{eq}(j, r_y). \tag{3}$$

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

$$D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot \widetilde{eq}(\mathsf{to{-}bits}(\mathsf{row}(k)), r_x) \cdot \widetilde{eq}(\mathsf{to{-}bits}(\mathsf{col}(k)), r_y). \tag{4}$$

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

Proof. Since D evaluates to a non-zero value at at most m locations over $\{0,1\}^{2 \log \mathsf{M}}$, D can be represented uniquely with m tuples of the form $(i, j, D(i,j)) \in (\{0,1\}^{\log \mathsf{M}}, \{0,1\}^{\log \mathsf{M}}, \mathbb{F})$. By using the natural injection $\mathsf{to{-}field}$ from $\{0,1\}^{\log \mathsf{M}}$ to $\mathbb{F}$, we can view the first two entries in each of these tuples as elements of $\mathbb{F}$ (let $\mathsf{to{-}bits}$ denote its inverse). Furthermore, these tuples can

be represented with three m-sized vectors $R, C, V \in \mathbb{F}^m$, where tuple k (for all $k \in [m]$) is stored across the three vectors at the kth location in the vector, i.e., the first entry in the tuple is stored in R, the second entry in C, and the third entry in V. Take row as the unique MLE of R viewed as a function $\{0,1\}^{\log m} \to \mathbb{F}$. Similarly, col is the unique MLE of C, and val is the unique MLE of V. The lemma holds by inspection since Eqs. (3) and (4) 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 Eq. (4) is *exactly* what the sparse polynomial evaluation algorithm described in Sect. 2.1 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 Eq. (3)), and the other memory with the quantities $\widetilde{eq}(j, r_x)$, and then computed each term of Eq. (4) via one lookup into each memory, to the respective memory cells with (binary) indices $\mathsf{to{-}bits}(\mathsf{row}(k))$ and $\mathsf{to{-}bits}(\mathsf{col}(k))$, 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 Lemma 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 "dense" 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 Sect. 2.1.

In the evaluation phase described below, the prover proves that it correctly ran the sparse polynomial evaluation algorithm (§2.1) 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 Fig. 1, where 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 Sect. 2.1 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. (1) $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to{-}bits}(\mathsf{row}(k)), r_x)$; and (2) $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to{-}bits}(\mathsf{col}(k)), r_y)$.

A core insight of Spartan [28] is to check these two conditions using memory-checking techniques [3]. These techniques amount to an efficient randomized

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 $\mathsf{val}, E_{\mathsf{rx}}, E_{\mathsf{ry}}$.

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

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: Offline Memory Checking. Recall that in the offline memory checking algorithm of [3], 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 [3,12,29], the trusted checker maintains a single timestamp counter and uses it to compute write timestamps, whereas in Spark and our 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.[13] For this reason,

[13] The same timestamp update procedure was used in Spartan's use of Spark [28, §7.2.3]. The purpose was 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.

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.[14] $\mathsf{RS} = \{\}$, 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, [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 lemma captures the invariant maintained on the sets of the checker:

Lemma 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 the field characteristic $|\mathbb{F}|$. 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 $k \in$ [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 $k \in$ [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 lemma. For notational convenience, let WS_i and RS_i $(0 \leq i \leq m)$ denote the multisets maintained by the trusted

[14] The checker in [3] maintains a fingerprint of these sets, but for our exposition, we let the checker maintain full sets.

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 $(k, v, t-1)$.[15] This is because we have assumed that the value v was not written in the initialization phase, and outside of the initialization phase, 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 in $\mathbb{F}$ of the form $t - i$ for $i = 1, 2, \ldots, \mathrm{char}(\mathbb{F})$, there is some read operation that returns (k, v, t'). Since there are m many read operations and the characteristic of field is greater than m, we obtain a contradiction.

Remark 2. The proof of Lemma 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 how these additional polynomials 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 two vectors requires computation comparable to $O(m)$ operations over $\mathbb{F}$.

Let $\mathsf{read\_cts} = \widetilde{C_r}, \mathsf{write\_cts} = \widetilde{C_r} + 1, \mathsf{final\_cts} = \widetilde{C_f}$. 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 $(2 \log \mathsf{M})$-variate multilinear polynomial D that evaluates to a non-zero value at at

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

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 $(\mathsf{read\_cts_{row}}, \mathsf{read\_cts_{col}})$, and two $(\log \mathsf{M})$-variate multilinear polynomials $(\mathsf{final\_cts_{row}}, \mathsf{final\_cts_{col}})$, where $(\mathsf{read\_cts_{row}}, \mathsf{final\_cts_{row}})$ and $(\mathsf{read\_cts_{col}}, \mathsf{final\_cts_{col}})$ are respectively the counter polynomials for the m addresses specified by row and col over a memory of size M. After that, in addition to performing the polynomial IOP depicted earlier in the proof (Fig. 1), the core idea is to check if the two oracles sent by the prover satisfy the conditions identified earlier using Lemma 2.

Lemma 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,* $(E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_cts_{row}}, \mathsf{final\_cts_{row}}, \mathsf{read\_cts_{col}}, \mathsf{final\_cts_{col}})$ *denotes 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*

$$\forall k \in \{0,1\}^{\log m},\ E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to{-}bits}(\mathsf{row}(k)), r_x). \tag{5}$$

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

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

Meanwhile, if Eq. (5) *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* $\forall k \in \{0,1\}^{\log m}$*,* $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to{-}bits}(\mathsf{col}(k)), r_y)$ *is equivalent (in the sense above) to checking that* $\mathsf{WS}' = \mathsf{RS}' \cup S'$*, where*

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

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

Here, we clarify the following subtlety. The expression $\mathsf{to{-}bits}(\mathsf{row}(k))$ appearing in Eq. (5) 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 2 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 Lemma 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:

// 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 $E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_cts}_{\mathsf{row}}, \mathsf{read\_cts}_{\mathsf{col}}$ and two $(\log \mathsf{M})$-variate multilinear polynomials $\mathsf{final\_cts}_{\mathsf{row}}$, $\mathsf{final\_cts}_{\mathsf{col}}$.
2. Recall that Claim 1 (see Equation (4)) 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" ([33, Proposition 2] or [30, 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:
   - $\widetilde{eq}(r_{\mathsf{M}}, r_x) \stackrel{?}{=} v_{eq}$
   - $E_{\mathsf{rx}}(r_m) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$
   - $\mathsf{row}(r_m) \stackrel{?}{=} v_{\mathsf{row}}$; $\mathsf{read\_cts}_{\mathsf{row}}(r_m) \stackrel{?}{=} v_{\mathsf{read\_cts}_{\mathsf{row}}}$; and $\mathsf{final\_cts}_{\mathsf{row}}(r_{\mathsf{M}}) \stackrel{?}{=} v_{\mathsf{final\_cts}_{\mathsf{row}}}$
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 $E_{\mathsf{rx}}, \mathsf{row}, \mathsf{read\_cts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{row}}$.
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" ([33, Proposition2] or [30, 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:
    - $\widetilde{eq}(r'_{\mathsf{M}}, r_y) \stackrel{?}{=} v'_{eq}$
    - $E_{\mathsf{ry}}(r'_m) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$
    - $\mathsf{col}(r'_m) \stackrel{?}{=} v_{\mathsf{col}}$; $\mathsf{read\_cts}_{\mathsf{col}}(r'_m) \stackrel{?}{=} v_{\mathsf{read\_cts}_{\mathsf{col}}}$; and $\mathsf{final\_cts}_{\mathsf{col}}(r'_{\mathsf{M}}) \stackrel{?}{=} v_{\mathsf{final\_cts}_{\mathsf{col}}}$
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 $E_{\mathsf{ry}}, \mathsf{col}, \mathsf{read\_cts}_{\mathsf{col}}, \mathsf{final\_cts}_{\mathsf{col}}$.

Fig. 2. Evaluation procedure of the Spark sparse polynomial commitment scheme.

Lemma 4 (*[28]*). *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, Lemma 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 [34, §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 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 Fig. 2, which assumes that the verifier has an oracle access to multilinear polynomial oracles that encode D (namely, $\mathsf{row}, \mathsf{col}, \mathsf{val}$)

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

Soundness. Applying a standard union bound to the soundness error introduced by probabilistic multiset equality checks with the soundness error of the sum-check protocol [24], 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 [30, 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 [33] in all three sum-check reductions (and using the linear-time prover in the grand product protocol [30,33]), the prover's time is $O(N)$ finite field operations for unstructured tables.

Finally, to prove Theorem 1, applying the compiler of [9] to the depicted polynomial IOP with the given dense polynomial commitment primitive, followed by the Fiat-Shamir transformation [15], provides the desired non-interactive argument of knowledge for proving evaluations of committed sparse multilinear polynomials, with efficiency claimed in the theorem statement.

The full version of this paper [32] provides details of the grand product argument.

Additional Discussion and Intuition. As previously discussed, the protocol in Fig. 2 allows the prover to prove that it correctly ran the sparse polynomial evaluation algorithm described in Sect. 2.1 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 offline 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 MLE-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 Fig. 2 to commit to these values.

3.2 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 the definition of $\widetilde{\mathsf{eq}}$ polynomial [32,34] that $\widetilde{\mathsf{eq}}_s(x,e) = \prod_{i=1}^{s}(x_i e_i + (1-x_i)(1-e_i))$.

Equation (3) expressed the evaluation $\widetilde{D}(r_x, r_y)$ of a sparse $2\log(\mathsf{M})$-variate multilinear polynomial $\widetilde{D}$ as

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

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

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, $\mathsf{read\_cts}_{\mathsf{row}}$, and one $\log(\mathsf{M})$-variate polynomial, e.g., $\mathsf{final\_cts}_{\mathsf{row}}$.

Supporting $\log N = c\log\mathsf{M}$ variables rather than $2\log\mathsf{M}$. 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: $\widetilde{\mathsf{eq}}_{3\log(\mathsf{M})}((i,j,k),(r_x,r_y,r_z)) = \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i,r_x)\cdot\widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j,r_y)\cdot\widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k,r_z)$.

Hence, if D is a $(3\log\mathsf{M})$-variate polynomial, we obtain the following analog of Eq. (6):

$$\widetilde{D}(r_x,r_y,r_z) = \sum_{(i,j,k)\in\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}} D(i,j,k)\cdot\widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i,r_x)\cdot\widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j,r_y)\cdot\widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k,r_z). \quad (7)$$

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 Sect. 2.1 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): (1) the size of the commitment is $\mathsf{c}(\mathsf{M})$; (2) the running time of the commit algorithm is $\mathsf{tc}(\mathsf{M})$; (3) the running time of the prover to prove a polynomial evaluation is $\mathsf{tp}(\mathsf{M})$; (4) the running time of the verifier*

to verify a polynomial evaluation is $\mathsf{tv}(\mathsf{M})$*; and (5) 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: (1) the size of the commitment is* $(3c+1)\mathsf{c}(m) + c \cdot \mathsf{c}(\mathsf{M})$*; (2) the running time of the commit algorithm is* $O\left(c \cdot (\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))\right)$*; (3) the running time of the prover to prove a polynomial evaluation is* $O\left(c\,(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M}))\right)$*; (4) the running time of the verifier to verify a polynomial evaluation is* $O\left(c\,(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M}))\right)$*; and (5) the proof size is* $O\left(c\,(\mathsf{p}(m) + \mathsf{p}(\mathsf{M}))\right)$*.*

3.3 Specializing the Spark Sparse Commitment Scheme to Lasso

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 Lemma 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 $\mathsf{to{-}field}(k) = \sum_{j=1}^{\log m} 2^{j-1} \cdot k_j$. Hence, in Eq. (4) of Lemma 1, $\mathsf{to{-}bits}(\mathsf{row}(k))$ is simply k.[16] 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, these effectively remove 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.

[32, Figure 3] describes Spark specialized for Lasso to commit to $\widetilde{M}$. 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 Sect. 3.1), $E_1, \ldots, E_c$, and $\mathsf{read\_cts}_1, \ldots, \mathsf{read\_cts}_c$, as well as c dense multilinear polynomials in $\log(N^{1/c}) = \log(N)/c$ variables, called $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$. Each dim_i is purported to be the memory cell from the i'th memory that the sparse

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

polynomial evaluation algorithm (Sect. 2.1) reads at each of its m time steps, $E_1, \ldots, E_c$ the values returned by those reads, and $\mathsf{read\_cts}_1, \ldots, \mathsf{read\_cts}_c$ the associated counts. $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$ are purported to be to counts returned by the memory checking procedure's 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 $\mathsf{read\_cts}_1, \ldots, \mathsf{read\_cts}_c$ each map $\{0,1\}^{\log m}$ to $\{0, \ldots, m-1\}$; $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$ 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 $\mathsf{read\_cts}_i$ and $\mathsf{final\_cts}_i$ can be larger than j.

4 Surge: A Generalization of Spark, Providing Lasso

The technical core of the Lasso lookup argument is Surge, a generalization of Spark. In particular, Lasso is simply a straightforward use of Surge.

Recall from Sect. 3 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 same as that of Spark. Surge generalizes Spark in that the Surge prover proves a larger class of statements about the committed polynomial $\widetilde{M}$ (Spark focused only on proving *evaluations* of the sparse polynomial $\widetilde{M}$).

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

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

Indeed, if $M \cdot t$ and a are the same vector, then Eq. (8) holds for every choice of r, while if $Mt \neq a$, then by the Schwartz-Zippel lemma, Eq. (8) holds with probability at most $\frac{\log m}{|\mathbb{F}|}$. So up to soundness error $\frac{\log m}{|\mathbb{F}|}$, checking that $Mt = a$ is equivalent to checking that Eq. (8) holds.

In Lasso, the verifier obtains $\widetilde{a}(r)$ via the polynomial commitment to $\widetilde{a}$. Then, the prover establishes Eq. (8) 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 Eq. (8). Essentially, the proof *proves* that the prover correctly ran a (very efficient) algorithm for evaluating the left hand side of Eq. (8).

A Roughly $O(\alpha m)$-time Algorithm for Computing the LHS of Eq. (8). From Eq. (3), $\widetilde{M}(r, y) = \sum_{(i,j) \in \{0,1\}^{\log m + \log N}} M_{i,j} \cdot \widetilde{eq}(i, r) \cdot \widetilde{eq}(j, y)$.

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 Eq. (8) equals

$$\sum_{i \in \{0,1\}^{\log m}} \widetilde{eq}(i, r) \cdot T[\mathsf{nz}(i)]. \tag{9}$$

Suppose that T is a SOS table. 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 $r = (r_1, \ldots, r_c) \in \left(\{0,1\}^{\log(N)/c}\right)^c$,

$$T[r] = g\left(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c]\right). \tag{10}$$

For each $i \in \{0,1\}^{\log m}$, decompose $\mathsf{nz}(i)$ and $(\mathsf{nz}_1(i), \ldots, \mathsf{nz}_c(i)) \in [N^{1/c}]^c$. Then Expression (9) equals

$$\sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i, r) \cdot g\left(T_1[\mathsf{nz}_1(i)], \ldots, T_k[\mathsf{nz}_1(i)], T_{k+1}[\mathsf{nz}_2(i)], \ldots, T_{2k}[\mathsf{nz}_2(i)], \ldots, T_{\alpha-k+1}[\mathsf{nz}_c(i)], \ldots, T_\alpha[\mathsf{nz}_c(i)]\right). \tag{11}$$

The algorithm to compute Expression (11) simply initializes all tables $T_1, \ldots, T_\alpha$, then iterates over every $i \in \{0,1\}^m$ and computes the i'th term of the sum with a single lookup into each table (of course, the algorithm evaluates g at the results of the lookups into $T_1, \ldots, T_\alpha$, and multiplies the result by $\widetilde{\mathsf{eq}}(i, r)$).

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

The verifier chooses $r \in \{0,1\}^{\log m}$ at random and requests that the Surge prover prove that the committed polynomial $\widetilde{M}$ satisfy Eq. (9). The prover does so by proving it ran the aforementioned algorithm for evaluating Expression (11). Following the memory-checking procedure in Sect. 3, 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 $\mathsf{read\_cts}_i$ (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 $\mathsf{final\_cts}_i$ (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

$$\sum_{j \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, j) \cdot g\left(E_1(j), \ldots, E_\alpha(j)\right). \tag{12}$$

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 i.e., that $E_i(j)$ equals $T_i[\dim_i(j)]$ for all $j \in \{0,1\}^{\log m}$. This is done exactly as in Spark to confirm that for each of the α memories, $\mathsf{WS} = \mathsf{RS} \cup S$ (see Lemmas 3 and 4, and [32, Figure 3]). At the end of this procedure, for each $i = 1, \ldots, \alpha$, the verifier needs to evaluate each of $\dim_i$, $\mathsf{read\_cts}_i$, $\mathsf{final\_cts}_i$ at a random point, which it can do with one

query to each. The verifier also needs to evaluate the multilinear extension $\widetilde{t}_i$ of each sub-table T_i for each $i = 1, \ldots, \alpha$ at a single point. T being SOS guarantees that the verifier can compute each of these evaluations in $O(\log(N)/c)$ time.

Prover Time. Besides committing to the polynomials $\dim_i, E_i, \mathsf{read\_cts}_i$, $\mathsf{final\_cts}_i$ 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 (12) and the grand product arguments (which can be batched). Using the linear-time sum-check protocol [13,28,33], the prover can compute its messages in the sum-check protocol used to compute Expression (12) with $O(b \cdot k \cdot \alpha \cdot m)$ field operations, where recall that $\alpha = k \cdot c$ and b is the number of monomials in g. If $k = O(1)$, then this is $O(b \cdot c \cdot m)$ time. For many tables of practical interest, the factor b can be eliminated (e.g., if the *total degree* of g is a constant independent of b, such as 1 or 2). The costs for the prover in the memory checking argument is similar to Spark: $O(\alpha \cdot m + \alpha \cdot N^{1/c})$ field operations, plus committing to a low-order number of field elements.

Verification Costs. The sum-check protocol used to compute Expression (12) 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 $O(c \cdot k \cdot \log m)$ field elements, and the verifier performs $O(k \cdot \log m)$ field operations. The costs of the memory checking argument for the verifier are identical to Spark.

Completeness and Knowledge Soundness of the Polynomial IOP. Completeness holds by design and by the completeness of the sum-check protocol, and of the memory checking argument.

By the soundness of the sum-check protocol and the memory checking 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 $j_1 = \dim_1(i), \ldots, j_c = \dim_c(i)$.

Theorem 3. *Figure 3 is a complete and knowledge-sound polynomial IOP for establishing that the prover knows an $m \times N$ matrix $M \in \{0,1\}^{m \times N}$ with exactly one entry equal to 1 in each row, such that*

$$\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) T[y] = v. \tag{13}$$

The discussion surrounding Eq. (8) explained that checking that $Mt = a$ is equivalent, up to soundness error $\log(m)/|\mathbb{F}|$, to Eq. (13) holding for a random $r \in \mathbb{F}^{\log m}$. Combining this with Theorem 3 implies that the protocol in [32, Figure 5] i.e., Lasso, is a lookup argument.

T is an SOS lookup table of size N i.e., 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 $\mathrm{dim}_1, \ldots, \mathrm{dim}_c$, each over $\log m$ variables. dim_i is purported to provide the indices of $T_{(i-1)k+1}, \ldots, T_{ik}$ read by the natural algorithm computing $\sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i, r) \cdot T[\mathsf{nz}[i]]$ (Equation (11)).

$//\mathcal{V}$ requests $\langle u, t \rangle$, where the ith entry of t is $T[i]$ and the yth entry of u is $\widetilde{M}(r, y)$.

1. $\mathcal{P} \to \mathcal{V}$: 2α different $(\log m)$-variate multilinear polynomials $E_1, \ldots, E_\alpha$, $\mathsf{read\_cts}_1, \ldots \mathsf{read\_cts}_\alpha$ and α different $(\log(N)/c)$-variate multilinear polynomials $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_\alpha$.
   $//E_i$ is purported to specify the values of each of the m reads into T_i.
   $//\mathsf{read\_cts}_1, \ldots \mathsf{read\_cts}_\alpha$ and $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_\alpha$, are "counter polynomials" for each of the α sub-tables T_i.
2. $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $h(k) := \widetilde{\mathsf{eq}}(r, k) \cdot g(E_1(k), \ldots, E_\alpha(k))$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} g(E_1(k), \ldots, E_\alpha(k))$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_z) \stackrel{?}{=} v_{E_i}$ for $i = 1, \ldots, \alpha$. Here, $v_{E_1}, \ldots, v_{E_\alpha}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the above equalities hold with one oracle query to each E_i.
4. // The following checks if E_i is well-formed, i.e., that $E_i(j)$ equals $T_i[\mathrm{dim}_i(j)]$ 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" ([33, Proposition2] or [30, 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:
   - $E_i(r_i''') \stackrel{?}{=} v_{E_i}$
   - $\mathrm{dim}_i(r_i''') \stackrel{?}{=} v_i$; $\mathsf{read\_cts}_i(r_i''') \stackrel{?}{=} v_{\mathsf{read\_cts}_i}$; and $\mathsf{final\_cts}_i(r_i'') \stackrel{?}{=} v_{\mathsf{final\_cts}_i}$
7. $\mathcal{V}$: Check the equations hold with an oracle query to each of $E_i, \mathrm{dim}_i, \mathsf{read\_cts}_i, \mathsf{final\_cts}_i$.

Fig. 3. Surge's polynomial IOP for proving that $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) T[y] = v$.

References

1. Arun, A., Setty,S., Thaler, J.: Jolt: SNARKs for virtual machines via lookups. In: EUROCRYPT, pp. xx–yy. Springer, Cham (2024)
2. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev., M., In: Fast reed-solomon interactive oracle proofs of proximity. In: ICALP (2018)

3. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: FOCS (1991)
4. Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Halo infinite: recursive zk-SNARKs from any additive polynomial commitment scheme. In: Cryptology ePrint Archive, Report 2020/1536 (2020)
5. Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 327–357. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_12
6. Bootle, J., Cerulli, A., Groth, J., Jakobsen, S., Maller., M.: Arya: nearly linear-time zero-knowledge proofs for correct program execution. In: ASIACRYPT (2018)
7. Bowe, S., Grigg, J., Hopwood, D.: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021 (2019)
8. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: short proofs for confidential transactions and more. In: S&P (2018)
9. Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK compilers. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 677–706. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_24
10. B. Chen, B. Bünz, D. Boneh, and Z. Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In: Hazay, C., Stam, M. (eds.) EUROCRYPT. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30617-4_17
11. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 738–768. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_26
12. Clarke, D., Devadas, S., Dijk, M.V., Gassend, B., Edward, G., Mit, S.: Incremental multiset hash functions and their application to memory integrity checking. In: ASIACRYPT (2003)
13. Cormode, G., Thaler, J., Yi, K.: Verifying computations with streaming interactive proofs. Proc. VLDB Endow. **5**(1), 25–36 (2011)
14. Eagen, L., Fiore, D., Gabizon, A.: CQ: cached quotients for fast lookups. Cryptology ePrint Archive (2022)
15. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
16. Gabizon, A., Khovratovich, D.: Flookup: fractional decomposition-based lookups in quasi-linear time independent of table size. Cryptology ePrint Archive (2022)
17. Gabizon, A., Williamson, Z.: Proposal: the TurboPlonk program syntax for specifying SNARK programs (2020)
18. Gabizon, A., Williamson, Z.J.: plookup: a simplified polynomial protocol for lookup tables (2020)
19. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. ePrint Report 2019/953 (2019)
20. Golovnev, A., Lee, J., Setty, S., Thaler, J., Wahby, R.S.: Brakedown: linear-time and post-quantum snarks for R1CS. Cryptology ePrint Archive (2021)
21. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: ASIACRYPT, pp. 177–194 (2010)
22. Kothapalli, A., Setty, S., Tzialla, I.: Nova: recursive zero-knowledge arguments from folding schemes. In: CRYPTO. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15985-5_13

23. Lee, J.: Dory: efficient, transparent arguments for generalised inner products and polynomial commitments. In: Nissim, K., Waters, B. (eds.) TCC 2021. LNCS, vol. 13043, pp. 1–34. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90453-1_1
24. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. In: FOCS, October 1990
25. Pătraşcu, M., Thorup, M.: Twisted tabulation hashing. pp. 209–228 (2013)
26. Posen, J., Kattis, A.A.: Caulk+: table-independent lookup arguments. Cryptology ePrint Archive (2022)
27. Pătraşcu, M., Thorup, M.: The power of simple tabulation hashing. J. ACM (JACM) **59**(3), 1–50 (2012)
28. Setty, S.: Spartan: efficient and general-purpose zkSNARKs without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 704–737. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_25
29. Setty, S., Angel, S., Gupta, T., Lee, J.: Proving the correct execution of concurrent services in zero-knowledge. In: OSDI, October 2018
30. Setty, S., Lee, J.: Quarks: quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020)
31. Setty, S., Thaler, J., Wahby, R.: Customizable constraint systems for succinct arguments. Cryptology ePrint Archive (2023)
32. Setty, S., Thaler, J., Wahby, R.: Unlocking the lookup singularity with Lasso. Cryptology ePrint Archive (2023)
33. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 71–89. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_5
34. Thaler, J.: Proofs, arguments, and zero-knowledge. Found. Trends Privacy Secur. **4**(2–4), 117–660 (2022)
35. Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. In: S&P (2018)
36. T. Xie, Y. Zhang, and D. Song. Orion: Zero knowledge proof with linear prover time. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15985-5_11
37. Zapico, A., Buterin, V., Khovratovich, D., Maller, M., Nitulescu, A., Simkin, M.: Caulk: lookup arguments in sublinear time. Cryptology ePrint Archive (2022)
38. Zapico, A., Gabizon, A., Khovratovich, D., Maller, M., Ràfols, C.: Baloo: nearly optimal lookup arguments. Cryptology ePrint Archive (2022)