Twist and Shout: Faster memory checking arguments via one-hot addressing and increments

Srinath Setty* Justin Thaler†

Abstract

A memory checking argument enables a prover to prove to a verifier that it is correctly processing reads and writes to memory. They are used widely in modern SNARKs, especially in zkVMs, where the prover proves the correct execution of a CPU including the correctness of memory operations.

We describe a new approach for memory checking, which we call the *method of one-hot addressing and increments*. We instantiate this method via two different families of protocols, called Twist and Shout. Twist supports read/write memories, while Shout targets read-only memories (also known as lookup arguments). Both Shout and Twist have logarithmic verifier costs. Unlike prior works, these protocols do not invoke "grand product" or "grand sum" arguments.

Twist and Shout significantly improve the prover costs of prior works across the full range of realistic memory sizes, from tiny memories (e.g., 32 registers as in RISC-V), to memories that are so large they cannot be explicitly materialized (e.g., structured lookup tables of size 2^{64} or larger, which arise in Lasso and the Jolt zkVM). Detailed cost analysis shows that Twist and Shout are well over $10\times$ times cheaper for the prover than state-of-the-art memory-checking procedures configured to have logarithmic proof length. Prior memory-checking procedures can also be configured to have larger proofs. Even then, we estimate that Twist and Shout are at least 2–4$\times$ faster for the prover in key applications.

Finally, using Shout, we provide two fast-prover SNARKs for non-uniform constraint systems, both of which achieve minimal commitment costs (the prover commits *only* to the witness): (1) SpeedySpartan applies to Plonkish constraints, substantially improving the previous state-of-the-art protocol, BabySpartan; and (2) Spartan++ applies to CCS (a generalization of Plonkish and R1CS), improving prover times over the previous state-of-the-art protocol, Spartan, by $6\times$.

*Microsoft Research
†a16 crypto research and Georgetown University

Contents

1 Introduction

A zero-knowledge virtual machine (zkVM) is a cryptographic protocol that allows an untrusted prover to succinctly prove that it correctly ran some computer program Ψ on a witness. In a zkVM, the computer program is specified to run on a specific CPU architecture, which is also called a virtual machine (VM) or an instruction set architecture (ISA). In a zkVM proof, the prover proves that it correctly ran Ψ cycle-by-cycle.

Generally speaking, zkVMs produce short proofs and have fast verification. Even if the zkVM first produces a proof π that is moderately large, π can be generically shrunk via *SNARK composition*. This means generating a smaller proof π' that establishes that the prover knows a valid π. Accordingly, the bottleneck in zkVM performance is prover runtime. zkVM provers today remain hundreds of thousands of times slower than native execution (i.e., the amount of work done to generate a proof π relative to that required to simply run Ψ with no proof of correctness).

Background on zkVMs and memory checking. zkVM provers prove that they did two things correctly, over and over. They first prove the correct execution of the "fetch-decode-execute" logic of the VM, once per cycle. Second, they prove that reads and writes to registers and RAM were executed faithfully.

A state-of-the-art zkVM, called Jolt [AST24], turns almost everything a VM does into reads and writes to memory. This includes the "fetch-decode-execute" logic of the VM. Roughly speaking, Jolt turns "fetch and decode" into a lookup into a read-only memory storing the program bytecode. Meanwhile, executing each primitive instruction is done via a lookup into a gigantic pre-determined table of size 2^{64}. Jolt accomplishes this by using Lasso [STW24], which internally turns this one lookup into a giant table into a short sequence of lookups into smaller tables, and combines the results to get the output of the instruction.

In summary, a key bottleneck for zkVM provers, and for the Jolt prover in particular, is proving that it correctly processed all reads and writes to memory. The fastest known methods for this task are called *offline memory checking* techniques. These techniques date back to work of Lipton [Lip89, Lip90] and Blum et al. [BEG+91]. In the context of zkVMs, offline memory checking techniques allow a prover to commit to a constant number of values, and perform a constant number of field operations, per memory operation performed by the CPU. Jolt currently uses Spice [SAGL18] as its memory-checking argument for registers and RAM, and Lasso [STW24] for read-only memories.

Our contributions. We describe a new approach to memory-checking that improves significantly on offline memory checking techniques, promising substantial end-to-end speedups for zkVM provers. For reasons that will become clear, we call our approach the *method of one-hot addressing and increments*.

We instantiate this method via two different families of protocols, called Twist and Shout. Twist supports read/write memories, while Shout targets read-only memories (also known as lookup arguments).[1] Both families are parameterized by an integer $d \geq 1$. The smaller the memory, the smaller d can be, and the cheaper each memory operation is for the prover as a result.

Twist and Shout significantly improve on prior works across the full range of realistic memory sizes, from tiny memories (e.g., 32 registers as in RISC-V), to read-only memories that are so large they can't be explicitly materialized (e.g., structured lookup tables of size 2^{64} or larger, as arise in the Jolt zkVM).

Detailed cost analysis shows that Twist and Shout are over $10\times$ cheaper for the prover than state-of-the-art memory-checking procedures configured to have similar proof length. Prior memory-checking procedures can also be configured to have larger proofs. Even then, we estimate that Twist and Shout are (at least) 2–$4\times$ faster for the prover.

Not only do Twist and Shout significantly speed up the prover across all practical memory sizes, they also give the prover a cost profile that more closely matches real CPUs. Unlike offline memory-checking techniques, the method of one-hot addressing is significantly faster for small memories than big ones, and Twist is faster for sequences of memory accesses with high locality (see Section 8.2.3 for details). This means that existing

[1] Twist is short for Tracking Write Increments via Sum-check Techniques and Shout for Sum-check-based Handling of Unchanging Tables.

compiler toolchains that optimize for program execution time on real CPUs are more likely to also lead to fast zkVM provers once memory-checking is done via Twist and Shout.

At the highest level, Twist and Shout leverage the power of the sum-check protocol [LFKN90] to minimize the amount of data the prover has to commit to, and to prove satisfaction of very large but very sparse constraint systems (i.e., a huge number of witness variables, but almost all of them are 0) while only "paying" for the sparsity of the witness (number of non-zeros) rather than the size of the witness.

The parameter d: where it comes from and what it buys. When applied to memories of size K, the prover in the simplest variations of Twist and Shout commits to $K-1$ 0s and a single 1 per read or write operation. This makes these protocols especially efficient when combined with elliptic-curve-based commitment schemes, which are uniquely fast at committing to 0s: any committed 0s literally do not alter the commitment so they are free to commit, and committing to a 1 requires a single group operation requiring only 6–7 field operations. This performance profile is an excellent fit for the Jolt zkVM for RISC-V [And17], which currently achieves state-of-the-art prover performance and uses elliptic curve commitment schemes (specifically, HyperKZG [ZSC24] or Zeromorph [KT23]).

However, even when 0s are free to commit to, large memories are problematic for the simplest instantiations of our methods, primarily because the commitment key (which must be stored by the prover) gets very large.[2] To address this, we describe an entire family of protocols in which, for any desired integer $d \geq 1$, the prover only commits to roughly $d \cdot K^{1/d}$ 0s per read or write operation. Using a large value of d comes at a cost: the prover also commits to d 1s per read or write operation, and each committed 1 forces the prover to perform one group operation. Hence, large values of d mean vastly fewer 0s are committed, which keeps the commitment key size bounded, but it also means slightly more 1s are committed, which modestly slows down the prover. Larger values of d also increase the amount of work the Twist and Shout provers do "outside of commitments", and proof size grows linearly with d too. So one wants to choose d as small as possible, subject to the constraint that the commitment key is not too large.

Choosing a value $d \geq 2$ renders our methods suitable for memory sizes K in the millions or billions. It also allows our techniques to be productively instantiated with hashing-based multilinear commitment schemes over binary fields like Binius [DP23], FRI-Binius [DP24], and Blaze [BCF+24]. Even though committing to 0s is not "free" with these schemes, committing to values that are known a priori to be in $\{0,1\}$ is cheap enough that our results still offer improvements over prior works.[3] For these commitment schemes, d should be set just large enough to ensure that commitment time is not a prover bottleneck.

Since the Twist and Shout provers commit to $dK^{1/d}$ bits per address, to ensure committing with a hashing-based scheme is sufficiently fast, $dK^{1/d}$ should be between a few dozen and a couple hundred bits, with larger memories leading to larger values of d (and more committed bits).

To summarize, in applications we expect d to be between 1 and 4 if using an elliptic curve commitment scheme and between 1 and 16 if using a binary-field hashing-based commitments. Larger memories necessitate larger values of d. See Section 2.8 for details.

The high-level ideas of Twist and Shout. A simple observation underlying Twist and Shout is that when memory addresses are specified in one-hot form, correctness of reads and writes becomes equivalent to very simple (i.e., rank-one) constraint systems. This observation is already explicit in prior works on lookup arguments [ZBK+22, ZGK+22, STW24, GM24].

These constraint systems can be directly checked for correctness via the standard sum-check-based zero-check PIOP for, say, R1CS (see Section 3.6 for details). But there is a challenge: the constraint systems are very large, so large that we cannot tolerate a prover that runs in time linear in the number of witness variables, since for T operations into a memory of size K, there are (more than) $K \cdot T$ witness variables. Fortunately, the constraint systems are highly *sparse*: although they involve a huge number of variables, only about T

[2] If the commitment key is structured, it must be generated via a trusted setup, which further limits its size.

[3] For memories consisting of just a few cells, such as the 2 registers in the Cairo Virtual Machine [GPR21, Section 2.4] and even the 32 registers in RISC-V, our protocols improve on prior work even when hashing-based commitment schemes over non-binary fields are used.

of them are non-zero. This lets us implement the sum-check prover applied to these constraint systems in roughly $O(K + T)$ time rather than $O(K \cdot T)$ time. Roughly speaking, the sum-check prover does not "pay" for 0-variables, and neither does the prover when committing to those variables, at least when using a curve-based commitment scheme.

On top of this, for very large but "structured" read-only memories, where $K = T^C$ with $C \geq 1$, by invoking (and generalizing) the sparse-dense sum-check protocol from the Generalized-Lasso protocol [STW24], we can implement Shout's sum-check prover in time $O(C \cdot T) \ll O(K + T)$.

When using parameter $d > 1$, Twist and Shout further exploit that every "one-hot" vector (meaning a vector in $\{0, 1\}^K$ with exactly one entry equal to 1) can be expressed as a tensor product of d smaller one-hot vectors, each of length $K^{1/d}$. We use this to control the negative effects of large memory sizes K, especially the size of the commitment key when using curve-based commitment schemes and the total number of committed bits when using hashing-based commitment schemes. This does come at the cost of higher-complexity constraints in the resulting constraint systems: they change from rank-1 to rank-d.

Twist needs to overcome an additional obstacle that threatens to bring in a $\Theta(K \cdot T)$ cost to the prover's runtime in the context of read/write memory. Specifically, a natural first attempt at designing Twist involves the prover committing to the value of *every register at every cycle*. This would be $K \cdot T$ committed non-zero values. Twist addresses this by instead having the prover commit to *increments*: the difference in value for each register cycle-by-cycle. Since at most one register is written per cycle, this entails committing to only T rather than $K \cdot T$ non-zero values (moreover, all T of these non-zero values are "small" and hence fast to commit to). At the end of the sum-check protocol, when the verifier needs to compute a random evaluation of (the multilinear extension of) the register values themselves, we offload that evaluation to the prover with a second invocation of the sum-check protocol that relates the register values to the increments. At the end of *that* sum-check, the verifier needs to evaluate the multilinear extension of the increments at a random point, which it can do because the increments themselves were committed.

We give two different implementations of the Twist prover. An important property of one of them is that the prover is faster when memory operations are *local*. Roughly, for a read or write operation to a memory cell that was accessed at most 2^i time steps prior, the Twist prover performs only $O(i)$ field multiplications to process that operation, less than the $O(\log K)$ multiplications that the prover performs per memory operation in the worst case. In order to achieve this, this variant of the Twist prover binds variables in a different order than the Shout prover: the Twist prover binds "cycle-count-specification" (i.e., time) variables first, rather than "memory-address-specification" (i.e., memory) variables. This ensures that in the first $\log K$ rounds, temporally-close accesses to the same memory cell quickly "coalesce", which is what allows the prover to benefit from locality of memory access. In other words, by binding time variables first, the sparsity (i.e., number of non-zeros) in the vectors the prover is processing falls quickly round-over-round when memory accesses are local, and the prover's runtime in each round grows only with the number of non-zeros in these vectors.

Fast-prover SNARKs for non-uniform constraints. We also use Shout to give two new, fast-prover SNARKs for non-uniform circuits or constraint systems. We call these SNARKs SpeedySpartan and Spartan++. Both offer a a significant improvement in overall prover time relative to the previous state of the art, BabySpartan [ST23], and in particular a $4\times$ improvement in commitment costs. In SpeedySpartan and Spartan++, the prover commits to the witness (the solution to the constraint system), and nothing else.

(At least this is the case when using polynomial commitment schemes with fast evaluation proof generation, e.g., Hyrax [WTS+18], Dory [Lee21]). For other schemes like HyperKZG, Zeromorph, and Bulletproofs/IPA, evaluation proofs themselves require committing to a linear amount of data.)

SpeedySpartan is concretely faster than Spartan++, due primarily to SpeedySpartan's use of a lookup table that is quadratically smaller than the one used by Spartan++. Hence, we consider SpeedySpartan to be of practical interest and Spartan++ to be primarily of conceptual interest (although even Spartan++ is concretely almost an order of magnitude faster than Spartan itself [Set20]). Spartan++'s most interesting performance property is that its prover's commitment costs grow linearly with the number of multiplication gates in the circuit and not the total gate count (i.e., addition gates do not contribute to commitment costs). But both SpeedySpartan

and Spartan++ reduce commitment costs so much that they are not a prover bottleneck, and hence this property of Spartan++ does not translate to major improvements in concrete costs.

The high-level ideas of SpeedySpartan and Spartan++. SpeedySpartan and Spartan++ observe that correct processing of circuit wires boils down to lookups. By circuit wires here, we mean which gate outputs need to be treated as inputs to which other gates.

There are two different ways this reduction from wires to lookups can be done. One technique originates in Spartan and its use of the Spark sparse polynomial commitment scheme [Set20]. The other originates in BabySpartan [ST23]. SpeedySpartan is a major refinement of the BabySpartan approach, while Spartan++ is a major improvement of the Spartan/Spark approach.

In the case of SpeedySpartan, BabySpartan already showed that each input to any gate can be computed with a lookup into the table that stores all gate outputs. BabySpartan invokes Lasso as the lookup argument. SpeedySpartan replaces Lasso with Shout. We further observe that this replacement allows certain polynomials that were explicitly committed by the BabySpartan prover to instead be *virtual*: they are *not* committed by the prover, and Shout nonetheless allows the verifier to evaluate these polynomials at the necessary evaluation point. This leads to a $4\times$ reduction in commitment costs, with field work improvements coming from the improved efficiency of Shout relative to Lasso.

In the case of Spartan++, we directly give a major improvement of the Spark sparse polynomial commitment scheme. In its evaluation proofs, Spark has the prover prove it correctly ran a fast algorithm for evaluating the committed sparse polynomial at the requested evaluation point (see [STW24, Section 3.1] for details of this view of Spark). This algorithm performs lookups into a large table that stores evaluations of multilinear Lagrange basis polynomials at the requested (random) evaluation point. Spark effectively applies the Lasso lookup argument to this table. This requires the prover to commit to two random values per non-zero coefficient of the sparse polynomial (these random values are the results of lookups into "subtables" decomposing the large table). We replace Lasso with Shout, and further observe that this allows for the values returned by the lookups to be virtual polynomials. This eliminates the major prover bottleneck in Spark.

To summarize, SpeedySpartan and Spartan++ use lookups in very different ways to deal with circuit wires. For circuits with n gates, SpeedySpartan computes a gate's inputs via lookups into the size-n table storing all gate outputs. Spartan++ performs lookups into a size-n^2 table storing all evaluations of multilinear Lagrange basis polynomials at a random evaluation point. Spartan++ pays increased costs relative to SpeedySpartan because the lookup table it uses is quadratically bigger, its use of the larger table does enable its commitment costs to grow linearly only with the number of multiplication gates.

2 Overview of Twist and Shout and their costs

2.1 Background

zkVMs and Jolt. The RISC-V instruction set architecture involves 32 registers. In each CPU cycle, at most two registers are read and one is written. The Jolt zkVM [AST24] forces an untrusted prover to correctly process reads and writes to registers (and random access memory) using an offline-memory-checking procedure called *Spice* [SAGL18]. Spice's application in Jolt currently accounts for about 20% of total prover time today, with most of this cost coming from registers (which are read and written nearly every CPU cycle, unlike RAM, which is accessed only via explicit load and store instructions).

On top of the cost of Spice for checking registers and RAM, another 25% of Jolt prover time is spent on Lasso [STW24], an offline-memory-checking procedure tailored to read-only memories. Jolt uses Lasso to ensure the prover correctly executes the appropriate primitive RISC-V instruction during each CPU cycle (Jolt turns primitive instruction execution into a handful of lookups into fixed tables of size about 2^{16}). So in total, the Jolt prover currently spends almost half of its time in offline-memory-checking procedures. We anticipate that the fraction of Jolt's prover costs will grow substantially over time, as other parts of the Jolt protocol continue to be optimized.

Polynomial IOPs and polynomial commitments. As with most SNARKs, Jolt consists of a *polynomial IOP* (PIOP), which can be combined with any polynomial commitment scheme (for multilinear polynomials, in the case of Jolt) to obtain a SNARK. A polynomial IOP is an interactive protocol where the prover is allowed to send one or more large polynomials to the verifier, but the verifier is only allowed to query those polynomials' evaluation at, say, a single randomly chosen point.

A polynomial commitment scheme is a cryptographic protocol specifically designed to transform any polynomial IOP into a succinct argument. Specifically, rather than having the polynomial IOP prover explicitly send large polynomials to the verifier, a polynomial commitment scheme allows a prover to succinctly commit to those polynomials, and later reveal only the evaluations of the polynomials that are queried by the verifier (along with *evaluation proofs*, which prove that the returned evaluations are indeed consistent with the committed polynomials). One can render the succinct argument non-interactive with the Fiat-Shamir transformation [FS86a].

Jolt uses multilinear polynomial commitment schemes based on elliptic curves, like HyperKZG [ZSC24] and Zeromorph [KT23]. The commitment to a polynomial of size n with these schemes consists of a single group element, and evaluation proofs consist of $O(\log n)$ group elements.[4] In the future, Jolt plans to incorporate support for hashing-based commitment schemes over binary fields.

Costs of elliptic-curve-based commitments. When a polynomial IOP is combined with an elliptic curve commitment scheme (e.g., HyperKZG [ZSC24]), the vast majority of the work done by the prover consists of elliptic curve group operations and finite field operations (over the field $\mathbb{F}$ used by the polynomial IOP, which is also the *scalar field* of an elliptic curve). For small memories, the method of one-hot addressing substantially lowers *both* the number of field operations over $\mathbb{F}$ and the number of group operations performed by the prover.

There are two facts about elliptic curve commitments that the method of one-hot addressing takes heavy advantage of. The first is that committing to 0s is "free". That is, when committing to a vector $v \in \mathbb{F}^\ell$, any i for which $v_i = 0$ literally does not affect the commitment, and can just be "ignored" by the prover when computing the commitment to v. The number of committed 0s does influence some other prover costs arising in the protocol, especially the size of the commitment key, but it does not influence commitment time. Fortunately, the effect of the 0s on these other costs can be controlled, see Section 3.1 for details. Hence, in this informal overview it is both clearest and reasonably accurate to treat committed 0s as literally free for the prover. Second, *small* (but non-zero) values are cheap (but not free) to commit to. For example, any i such that $v_i = 1$ costs the prover just one group operation when computing the commitment.

For simplicity, in this work we consider any value in $\{0, 1, \ldots, 2^{32} - 1\}$ to be "small". We select the threshold of $2^{32} - 1$ as the limit of smallness for convenience of accounting: most zkVMs today use 32-bit data types, and so registers store arbitrary elements of $\{0, 1, \ldots, 2^{32} - 1\}$.[5] Since the prover has no choice but to commit to the values read from or written to registers, it is convenient to call these values small. But this is also a reasonable notion of smallness: at practical instance sizes, committing to a 32-bit value with an elliptic curve commitment scheme takes roughly two group operations via Pippenger's bucketing algorithm[6]. This is much cheaper than committing to an arbitrary field element, which can require a dozen group operations or more.[7]

Costs of hashing-based commitments over binary fields. A nice property of hashing-based commitment schemes relative to curve-based ones is that the commitment key is small (often it simply specifies a cryptographic hash function), no matter the size of the vectors that need to be committed. However, hashing-based commitments have a major downside: committing to 0s is not free. This is a challenge because,

[4] If p is a v-variate multilinear polynomial, then its size is 2^v. This is because p is uniquely described by 2^v coefficients, or equivalently its evaluations over an interpolating set such as $\{0,1\}^v$, which has size 2^v.

[5] In this work, we use the terms *register* and *memory cell* interchangeably.

[6] See [EHB22, Section 4] for a clear overview of Pippenger's algorithm and its costs

[7] For maximal precision of cost accounting it may be useful to distinguish the "smallness" of elements of, say, $\{0, 1, \ldots, 2^{16} - 1\}$ vs. $\{2^{16}, \ldots, 2^{32} - 1\}$. The former take only about one group operation to commit to at practical instance sizes, while the latter require about two group operations. To simplify our cost accounting, we do not make this distinction in this paper. However, our findings would not be substantially changed if we did.

when Twist and Shout (say, with $d = 1$) are applied to prove T memory operations into a memory of size K, the prover needs to commit to $K \cdot T$ values, and we cannot tolerate $\Theta(K \cdot T)$ prover time.

Fortunately, all of these $K \cdot T$ values are known *a priori* to be in $\{0, 1\}$: not only are almost all of the values 0, the non-zeros are always equal to 1. There is one (recently-identified) subclass of hashing-based commitment schemes that make such known-to-be-in-$\{0, 1\}$ values sufficiently fast to commit to that Twist and Shout are useful when combined with such commitments. These are hashing-based commitment schemes for multilinear polynomials over *binary fields*, which allow the prover to "pack" 128 values into a single element of $\mathsf{GF}(2^{128})$, and commit to the packed values with an appropriate multilinear commitment scheme defined over $\mathsf{GF}(2^{128})$.[8] This packing technique results in a 128-fold reduction in the number of committed $\mathsf{GF}(2^{128})$ field elements, compared to the naive approach of assigning an entire $\mathsf{GF}(2^{128})$-element to represent each committed value. Examples include Binius [DP23], FRI-Binius [DP24], and Blaze [BCF+24].

For tiny memories like the 32 registers in RISC-V, $d = 1$ is fine: the means the prover commits to 32 bits per address, and four such addresses can be packed into a single $\mathsf{GF}(2^{128})$ field element before committing.[9]

As another attractive example, consider applying Twist with $d = 4$ to a memory with $K = 2^{20}$ cells (which corresponds to a zkVM with about 4 MBs of memory). Rather than committing to K values in $\{0, 1\}$ per memory operation as in Twist with $d = 1$, with $d = 4$ the Twist prover commits to only $4 \cdot 2^5 = 128$ values in $\{0, 1\}$. These 128 values can all be packed into a single element of $\mathsf{GF}(2^{128})$ before committing with the above hashing-based commitment schemes. On the other hand, increments are not guaranteed to be in the set $\{0, 1\}$ (although most increments will be 0s). Each increment is guaranteed to be an element of $\mathsf{GF}(2^{32})$, so four increments can be packed into single element of $\mathsf{GF}(2^{128})$ before committing.

2.2 Formulation of the memory checking problem

Motivation. Most primitive RISC-V instructions read two registers (the "source registers" containing the inputs to the primitive instruction executed at that cycle) and writes to one register (the "destination register") that stores the output of the primitive instruction. We denote the source registers ra_1 and ra_2 (ra is short for read address), and the destination register by wa (short for write address).

To force an untrusted prover to prove that it is reading and writing to registers correctly, RISC-V zkVMs demand that the prover commit to

- the register addresses ra_1, ra_2, and wa each cycle.
- the values that the prover claims is returned by the reads to ra_1 and ra_2,
- the value that is written to wa.

The goal of the memory checking problem is to devise a succinct argument that lets the prover prove that indeed the value committed for every read operation is the value most recently written to the relevant register.

Formulation for read/write memories. Say there are K memory cells, which we will refer to as *registers* for short (for example, there are $K = 32$ registers in RISC-V). In the case of read/write memory, let us assume that operations alternate between reads and writes.[10] Let T denote the *number of cycles* being checked. More specifically, T denotes the number of read operations in the case of read-only memory. For read/write memory, each cycle consists of a read followed by a write, so there are T reads and T writes (i.e., $2T$ memory operations in total).

[8] All hashing-based polynomial commitment schemes ultimately need to work over a field that is at least 128 bits in size to ensure adequate security.

[9] Committing to N $\mathsf{GF}(2^{128})$ elements with these schemes involves $\Theta(N)$ or $\Theta(N \log N)$ $\mathsf{GF}(2^{128})$ multiplications to apply the encoding procedure of an error-correcting code, followed by $\Theta(N)$ cryptographic hash evaluations. Computing evaluation proofs for these schemes can be fairly expensive for the prover, due to various invocations of the sum-check protocol in order to relate the "packed" committed values to the unpacked values.

[10] In a zkVM for RISC-V like Jolt, there are two committed register read operations and one committed register write operation per CPU cycle. But since this paper is focused on memory-checking, we find it clearest to use the term cycle to refer to a single memory operation rather than, say, a sequence of two reads and one write.

The memory-checking problem begins with the prover sending commitments to four vectors: raf, waf, rv and wv. We denote the j'th entry of these vectors with function notation, i.e., the j'th entry of raf is $\mathsf{raf}(j)$.

The j'th entries of raf and waf specify the address of the registers read and written in cycle j. In the context of zkVMs (and in typical offline-memory checking procedures like Spice), each register is indexed by one field element, and so raf and waf are also both in $\mathbb{F}^T$, with $\mathsf{raf}(j), \mathsf{waf}(j) \in \mathbb{F}$ specifying the register addresses read and written at cycle j. Here, any injective mapping from registers to field elements suffices for the indexing. In large prime-order fields, the K field elements used to address the registers can simply be $\{0, 1, \ldots, K-1\}$; for simplicity, this is the mapping we use throughout this manuscript. We use the letter f at the end of raf and waf to reflect that these are addresses specified via one field element per address. The j'th entry of rv specifies the value returned by the j'th read operation, and wv specifies the value written by the write operation. rv and wv are vectors in $\mathbb{F}^T$ (i.e., registers store field elements).

The memory checking problem then demands that the prover establish that $\mathsf{rv}(j)$ equals $\mathsf{wv}(j')$ where $j' < j$ denotes the largest cycle number less than or equal to j such that $\mathsf{waf}(j') = \mathsf{raf}(j)$. If no such j' exists, then it is required that $\mathsf{rv}(j) = 0$. Conceptually, this means that all registers are initialized to contain value 0, and every read to a register is required to return the value most recently written to that register.

It is also possible to demand that the contents of memory be initialized to a set of values that is not identically 0. In this case, either the vector of initial memory values (one value per cell) is committed by an honest party, or is simply public.[11]

Read-only memories. Our formulation of the memory-checking problem for read-only memories is identical to read/write memories, except that the contents of the memory cells are public (i.e., known to both the verifier and the prover) and each cycle involves only a read operation, rather than a read followed by a write. We sometimes refer to the contents of the read-only memory as the *lookup table*.[12]

One-hot encoding of addresses. Recall that in the above formulation of the memory-checking problem, each address is indexed by a single field element. This is how memory addresses are naturally specified in zkVM contexts. For example, in RISC-V, when performing a load or store from RAM, the relevant address is the value stored in the first source register, plus the "immediate" value for that instruction. Each zkVM register naturally stores a single field element, and the immediate value for any instruction is naturally represented by a single field element.

However, internal to the Twist and Shout proof systems, the register addresses must actually be specified via one-hot encoding. This means that within the Twist and Shout protocols, each address is a unit vector in $\{0,1\}^K \subseteq \mathbb{F}^K$. Accordingly, we introduce two vectors $\mathsf{ra}, \mathsf{wa} \in \mathbb{F}^{T \cdot K}$ that, for a sequence of T memory operations, specify all T memory addresses via one-hot encoding. We think of ra and wa as consisting of T rows each of length K. If the prover is honest, then the j'th row of ra, denoted $\mathsf{ra}(j)$, specifies via one-hot encoding the register read at cycle j, and similarly for wa. So if register ℓ is read at cycle j, then $\mathsf{ra}(j)$ is the ℓ'th unit vector $e_\ell \in \{0,1\}^K$. We denote the k'th entry of this vector by $\mathsf{ra}(k, j)$.

Because zkVMs naturally specify addresses via a single field element, when used within zkVMs, memory-checking arguments do need to give the verifier "access" to the vectors raf and waf, in which each address is specified via a single field element. In particular, the zkVM verifier needs to be able to evaluate the *multilinear extension polynomials* $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ at a random point. Twist and Shout support this. Even though $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ are *not* committed by the Twist and Shout provers (only $\widetilde{\mathsf{ra}}$ and $\widetilde{\mathsf{wa}}$ are), we still give a way for the verifier to evaluate $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ at a random point.

[11] In many applications, the initial contents of memory have a description that is much smaller than explicitly listing the value stored in each memory cell, and with some memory-checking arguments it is possible in these settings to avoid having the verifier spending time linear in the size of the memory to "process" the initial values [STW24, AST24].

[12] Many lookup arguments, including Arya [BCG+18], plookup [GW20b], Caulk [ZBK+22], and LogUp [Hab22], actually solve a different problem, which is sometimes referred to as *unindexed lookups* (in contrast to reads into read-only memory, which is sometimes referred to as *indexed lookups*) [STW24]. Unindexed lookup arguments ensure that a vector of committed values all reside *somewhere* in a read-only memory, but do not specify an address at which each value must reside. Unindexed lookup arguments can be transformed into indexed ones, though the transformation adds prover and verifier overheads. See [STW24] for details.

To use a term coined in work on Binius [DP23], Twist and Shout can be integrated into zkVMs by treating $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ as *virtual polynomials*: they are not explicitly committed by the prover, but can still be evaluated by the verifier at any point, by asking the prover to provide the requested evaluation and then prove that the evaluation is correct. This is done expressing the needed $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ evaluations in terms of $\widetilde{\mathsf{ra}}$ and $\widetilde{\mathsf{wa}}$, and then applying the sum-check protocol to compute this expression. This effectively reduces the verifier's task of evaluating $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ at a point, to the task of evaluating $\widetilde{\mathsf{ra}}$ and $\widetilde{\mathsf{wa}}$ at a different point. These evaluations can be obtained directly from the commitments to $\widetilde{\mathsf{ra}}$ and $\widetilde{\mathsf{wa}}$. See Section 4.1.2 for details.

Can read-values be virtual too? We formulated the memory-checking problem above to assume that the vector rv of values returned by read operations are committed by the prover (or more precisely, their multilinear extension polynomial $\widetilde{\mathsf{rv}}$ is committed). However, the "correct" value of every read operation is fully determined by the write operations and the read-addresses (or, in the case of read-only memories, simply by the read-addresses $\widetilde{\mathsf{ra}}$). And in applications to zkVMs, the SNARK verifier only ever needs to valuate $\widetilde{\mathsf{rv}}$ at a random point r.[13]

So, in principle, $\widetilde{\mathsf{rv}}$ need *not* be committed: as soon as the prover commits to $\widetilde{\mathsf{ra}}$ and to any polynomials specifying writes, $\widetilde{\mathsf{rv}}$ is *implicitly* specified as well, and all we need to ensure is that the verifier is able to evaluate $\widetilde{\mathsf{rv}}$ at a random point r. Twist and Shout directly give a protocol to accomplish this: from the verifier's perspective, Twist and Shout use the sum-check protocol to reduce the task of evaluating $\widetilde{\mathsf{rv}}(r)$ to the task of evaluating different polynomials that *are* committed by the prover.

Hence, unlike prior memory-checking arguments, Twist and Shout do allow the polynomial $\widetilde{\mathsf{rv}}$ to be virtual, rather than committed by the prover. Our prover cost estimates throughout this work reflect this.

Relevant memory sizes. In the motivating example of 32 RISC-V registers, the memory size K is tiny (32 cells), especially relative to the number of CPU cycles executed. Even simple computer programs often run for at least 2^{30} CPU cycles in total, and real computer programs often run for 2^{50} cycles or more.[14] Indeed, the clock rate of modern laptops allows for several billion CPU cycles to be executed per second, even by a single thread. Our results are strongest (and cleanest to understand) in this parameter regime where $K \ll T$, since then our the $O(K+T)$ field operations incurred by the Twist and Shout provers is clearly dominated by the T term.

However, all possible relationships between K and T are also of interest. For example, L1 cache in real CPUs is often dozens of KBs, corresponding to $K \approx 2^{13}$ memory cells, and L2 cache is an order of magnitude larger ($K \approx 2^{16}$).[15] And many programs may only use MBs of main memory (RAM), while others can use GBs (translating to 2^{30} memory cells or even more).

Less intuitively, in the case of read-only memories, we are also interested in the situation where $K \approx 2^{64}$ is so big that the entire contents of the memory cannot possibly be materialized. This setting arises in Jolt, where primitive instruction execution is handled by performing lookups into the "evaluation table" of the primitive instruction, which has size 2^{64}. These tables are highly structured, which ensures no one ever needs to materialize the entire gigantic memory. In other words, for "structured" read-only memories, one can achieve a prover runtime that is sublinear in K.

2.3 Prior work: Arguments via offline memory checking

Memory-checking procedures such as Spice [SAGL18] work by reducing the task of proving that all reads and writes were processed correctly, to the task of proving two related vectors $a \in \mathbb{F}^\ell$ and $b \in \mathbb{F}^\ell$ are permutations of each other. The precise value of ℓ differs amongst different memory-checking procedures, but is typically $O(T+K)$. Confirming that a and b are permutations of each other is done via "Lipton's trick" [Lip89, Lip90]:

[13] It is, of course, essential that the prover not be able to choose the polynomial $\widetilde{\mathsf{rv}}$ with knowledge of the point r at which the verifier will evaluate it.

[14] Though in zkVMs today, program executions are broken into "shards" consisting of only about 2^{20} cycles each, with each shard proved semi-independently, in order to keep the prover space bounded. See Section 3.1.1 for additional details.

[15] Reads into L1 cache on real CPUs typically take just 1-4 clock cycles, while reads into L2 cache often take 5-20 cycles.

the verifier picks a random $r \in \mathbb{F}$, and confirms that

$$\prod_{i=1}^{\ell}(a_i - r) = \prod_{i=1}^{\ell}(b_i - r). \tag{1}$$

Equation (1) clearly holds if a and b are permutations of each other, and if they are not, it holds with probability at most $\ell/|\mathbb{F}|$ over the random choice of r.

Appendix A provides a brief overview of how these transformations from checking read-write memories to checking permutations work.

Offline memory-checking procedures confirm that Equation (1) holds using of a *grand product argument*: a SNARK for computing the product of many committed values. Many memory-checking procedures in the literature invoke grand product arguments that interpret the committed vectors as univariate polynomials, and prove that a certain divisibility relationship holds between the committed polynomials. These arguments can lead to excellent verifier costs (often with proofs consisting of only a constant number of group elements), but generally lead to slow provers.

The grand product arguments used by Spice as instantiated in Jolt (as with the method of one-hot addressing) interpret committed vectors as *multilinear polynomials* and invokes the sum-check protocol of Lund, Fortnow, Karloff, and Nisan [LFKN90]. There are several such grand product arguments one can choose from, offering various tradeoffs between prover time and verifier costs [Tha13, Set20, SL20]. Depending on which grand product argument it invokes, Spice can have proofs of size between $O(\log^2 n)$ and $O(\log n)$ field elements, where n is the number of memory operations proven. The bigger the proofs, the faster the Spice prover.

As described in Appendix A, there are only a handful of known approaches to offline memory-checking for read/write memory [SAGL18, ZGK+18]. For *read-only* memories there is somewhat more diversity in solutions. Memory-checking arguments for read-only memories are also known as *lookup arguments*, and examples include Arya [BCG+18], plookup [GW20b], cq [EFG22], Lasso [EFG22], LogUp [Hab22, PH23], and others. We give a brief overview of this body of work in Appendix A.

Baselines for read/write memory: costs of Spice. We refer to Appendix A for an overview of how Spice works. Here, we merely discuss its costs. On each read to memory, the Spice prover commits to 5 values. The first three of these are an address, value, and "timestamp", while the remaining two arise during range checks that are performed on values derived from the timestamp. In the context of memory-checking within zkVMs like Jolt, all five of these values are "small", and hence fast to commit to (see Section 3.1 and Footnote 7 for details).[16] (If proving more than 2^{32} memory operations, the committed timestamps will not actually meet our threshold for "smallness". But we will treat them as small for simplicity of accounting–this only leads to an underestimate of the speedups we achieve relative to Spice.)

On top of committing to these 5 values, each read contributes six additional terms contributed to various grand products (see Equation (1)), and proving these grand products requires work by the prover. (Additionally, irrespective of how many memory operations are processed, each memory cell contributes two committed values and two factors to the grand product).

Using the fastest known grand product argument, due to Thaler [Tha13][17], the prover can process these six factors per read with about six field multiplications each, so roughly 40 field operations in total. This grand product argument has proofs of size $O(\log^2 n)$. In total this means the Spice prover using Thaler's grand product argument does $40T + 40K$ field operations.

Setty and Lee, in a work called Quarks [SL20], describe an alternative that has proofs of size $O(\log n)$ but has much higher prover costs. In particular, for computing a grand product over a vector of size n, the

[16] Whether we should count committed addresses and write-values towards the costs of the memory-checking procedure is debatable, since the addresses and write-values have to be committed simply to specify the memory-checking instance that Spice is applied to. We choose to count these two committed values towards prover costs. This accounting only has the effect of reducing our claimed factor speedups relative to prior work, since it increases the costs of all memory-checking arguments by the same absolute amount.

[17] Thaler's grand product argument is a refinement of the GKR protocol [GKR15], and our cost accounting for it incorporates recent optimizations [Gru24, DT24].

prover has to commit to n partial products, which are going to be random field elements in our context. In the memory checking context, in addition to performing about 40 field operations per read operation as with Thaler's grand product argument, the prover in Quarks' has to commit to 6 *random* field elements per operation.[18] Committing to six random field elements is expensive, equivalent in cost to performing over 500 field operations. Quarks more generally describe a spectrum of grand product arguments that have prover time and proof size costs "in between" the above two extremes.

Writes to memory in Spice have similar costs to reads; the only difference is that rather than the Spice prover committing to 5 values, for each write it commits to 6 values.

The above describes Spice's costs in large prime-order fields. Prover costs are slightly higher for fields of small characteristic (e.g., binary fields), due to complications with how committed timestamps must be specified in these fields [DP23, Section 4.4]. This applies to prior lookup arguments as well.

Baselines for read-only memory: Small or unstructured tables. Lasso's prover [STW24] commits to $3T + K$ small values in total, and performs about $12T + 12K$ field operations within the grand product argument. Inspired by the use of Thaler's grand product argument [Tha13] in Spark [Set20] and Lasso [STW24], LogUpGKR [PH23] combines the LogUp lookup argument [Hab22] with GKR protocol. LogUpGKR's prover commits $2T + K$ small values, which is slightly fewer than the $3T$ committed by Lasso, at least when $K \leq T$. But LogUpGKR's prover performs about twice as many field operations as Lasso's prover. This stems from LogUpGKR's use of a "grand sum of rational values" in place of a grand product: summing two rationals, $a/b + c/d$, requires three products, namely $a \cdot d$, $b \cdot c$, and $b \cdot d$, to ultimately derive $a/b + c/d = (ad + bc)/bd$.

We state the costs of these baselines in Figures 1–3. For all baselines, we incorporate all of the most recent known optimizations to the prover in the GKR protocol and Thaler's grand product argument [DT24, Gru24]. Further, we assume that the multilinear extension of the table values is efficiently evaluable by the verifier (as is the case for all lookup tables in Jolt, see Footnote 11).

Other relevant baselines are FLI [GM24], a folding scheme for lookups, and Proofs for Deep Thought [BC24], a folding scheme for read/write memory. We discuss these schemes further in Section 2.4.2 below.

Baselines for read-only memory: Gigantic, structured tables. Lasso [STW24] gives two different approaches to performing lookups into gigantic, structured tables. By gigantic, we mean the table size K is on the order of T^C for some integer $C \geq 2$. One, called Generalized-Lasso, applies to any MLE-structured table but requires the prover to commit to c random field elements, which is generally a bottleneck for the prover. The other, called simply Lasso, applies to tables satisfying a natural decomposability property, which roughly means that one lookup into the giant table can be turned into $O(C)$ lookups into "subtables" of size at most $T^{1/C}$ (Lasso then applies a "base" lookup argument to prove validity of the $O(C)$ subtable lookups). Lasso does not require the prover to commit to any random field elements. Other works have considering using different "base" lookup arguments within this approach [Dor24].

In order to prove that primitive instructions were correctly executed, Jolt applies Lasso to gigantic decomposable tables (namely, the entire evaluation table of the relevant primitive instruction). However, there are some overheads that come from the interaction of how zkVMs work with Lasso's use of decompositions. Specifically, the addresses of the subtable lookups have two "parts", with one part coming from the first input to the primitive instruction, and the other coming from the second part. This forces inputs to be decomposed into smaller chunks than would otherwise be necessary.

For example, to compute the bitwise XOR of two 32-bit inputs x and y, Jolt currently splits x and y each into four 8-bit chunks, and uses one subtable lookup (into a subtable of size 2^{16}) to compute the bitwise XOR of each chunk of x with the corresponding chunk of y. So even though the subtable addresses are 16 bits, x

[18] Two of these six random committed values are from Spice turning each read or write to memory into two separate operations. The other four are from range checks (done using Lasso) that Spice must do to implement appropriate updates to "timestamps" associated with each read and write to memory. See Appendix A for details. Given how expensive it is to commit to random values, when using Quarks as the grand product argument, a naive range check based on bit-decomposition would actually result in a faster prover than using Lasso. This would lower the Spice-with-Quarks-grand-product prover cost by a factor of about 2 compared to what is reported here, but it would not change the qualitative comparison to Twist.

Read/write memory checker	Non-zero committed values	Field multiplications	Proof size
Spice	$5R + 6W + 2K \approx 11T$	$80T + 80K \approx 80T$	$O(\log^2 T)$
Twist ($d = 1$)	$R + 3W = 4T$	$(5\log(K) + 16)T + O(K \log K)$	$O(d \log T)$
Twist ($d = 2$)	$2R + 4W = 6T$	$(5\log(K) + 32)T + O(K)$	$O(d \log T)$

Figure 1: Prover costs for Twist: read/write memory checking for $R = T$ reads and $W = T$ writes ($R+W = 2T$ memory operations in total). We report a worst-case bound on field multiplications for Twist. For $d = 1$, the Twist prover performs many fewer field operations if memory accesses are local. Roughly, 2^i-local memory accesses (see Section 8.2.1 for a definition) cost only $7i$ field multiplications rather than $5 \log K$. Appropriate values for d in Twist are discussed in Section 2.8. Reported Twist costs in this table include Booleanity-checking costs.

Read-only memory checker	Non-zero committed values	Field multiplications	Proof size
Lasso	$3T + \min\{K, T\}$	$12T + 12K$	$O(\log^2 T)$
LogUpGKR	$2T + \min\{K, T\}$	$21T + 21K$	$O(\log^2 T)$
Shout ($d = 1$)	T	$4T + O(K \log K)$	$O(d \log T)$
Shout ($d = 2$)	$2T$	$11T + O(K)$	$O(d \log T)$

Figure 2: Prover costs for Shout (read-only memory checking for T reads, into a memory of size K). All committed values are small. Appropriate values for d in Shout are discussed in Section 2.8. Reported costs for Shout are inclusive of field multiplications required to prove that committed addresses are valid one-hot encodings (including Booleanity-checking).

and y must each be decomposed into only 8-bit chunks. Furthermore, these committed 8-bit chunks must all be range-checked to confirm they are indeed field elements in $\{0, 1, \ldots, 2^8 - 1\}$. As we will see, Shout avoids any explicit table decomposition, and hence also avoids the above overheads when used in the context of a zkVM such as Jolt.

Closest related work: Generalized-Lasso. Shout can be viewed as a natural generalization or improvement of the Generalized-Lasso protocol from [STW24]. Shout with $d = 1$ is in fact equivalent to Generalized-Lasso, except that Shout uses any standard (i.e., dense) polynomial commitment scheme to commit to the one-hot encodings of addresses, rather than the sparse polynomial commitment scheme Spark [Set20] invoked by Generalized-Lasso. Shout thereby avoids Generalized-Lasso's need to commit to random field elements (which comes from Spark). However, setting $d = 1$ only works for very small lookup tables due to the constraints described in Section 1. At the other extreme, Shout with $d = \log K$ (the largest meaningful value of d) is roughly equivalent to combining Generalized-Lasso with a different sparse polynomial commitment scheme called "Spark-Naive" [Set20]. Setting d this large brings high costs–see Remark 1 for details.

From this standpoint, Shout can be viewed as an improvement in sparse polynomial commitment schemes rather than in lookup arguments. Shout effectively identifies a family of sparse polynomial commitment schemes, with all schemes in the family avoiding Spark's need to commit to random field elements. There is a different scheme for each value of $d \geq 1$, where committing to bigger, sparser polynomials calls for higher settings of d. Shout then simply combines Generalized-Lasso with this family of sparse polynomial commitment schemes. This perspective, of Shout as an advance in the speed of sparse polynomial commitment schemes is made explicit in Section 9.3.1 (as Spartan++, one of our SNARKs for non-uniform circuits, explicitly uses such a Shout-based sparse polynomial commitment, which we call Spark++).

2.4 Costs of Twist and Shout

As with offline memory-checking techniques such as Spice, Twist and Shout can be formulated as a polynomial IOPs for the memory-checking problem, and the PIOP can be turned into a SNARK by combining it with any suitable polynomial commitment scheme. Recall that Twist and Shout actually refer to a family of PIOPs

Read-only memory checker	Non-zero committed values	Number of field multiplications	Proof size
Lasso	$\geq 16T$	$\geq 144T$	$O(\log^2 T)$
Shout ($d = 2$)	$2T$	$42T$	$O(d \log T)$
Shout ($d = 4$)	$4T$	$65T$	$O(d \log T)$
Shout ($d = 8$)	$8T$	$112T$	$O(d \log T)$

Figure 3: Prover costs for Shout when doing T lookups into an appropriately structured read-only memory of size K such that $K^{1/C} = o(T)$ for integer $C = 4$. All committed values are small. Reported Shout costs are inclusive of field multiplications required to prove that committed addresses are valid one-hot encodings (including Booleanity-checking). For comparison, we report (optimistic estimates of) the costs of Lasso as used today in Jolt, whereby each lookup into the table of size $K \approx 2^{64}$ is decomposed into between 4 and 10 lookups into "subtables" of size 2^{16}, and where committing to the addresses for those subtables necessitates committing to 2 small values per subtable address and range-checking each such value.

Memory checker	Memory size	Bits of Committed Data	Field multiplications	Proof size
Lasso	$K = 2^{64}$ (structured)	at least $928T$	at least $194T$	$O(\log^2 T)$
Shout ($d = 16$)	$K = 2^{64}$ (structured)	$256T$	$131T$	$O(d \log T)$
Spice	$K = 32$	$421T$	$95T$	$O(\log^2 T)$
Twist ($d = 1$)	$K = 32$	$128T$	$35T$	$O(d \log T)$

Figure 4: Amount of committed data measured in bits for Twist and Shout vs. prior work for two important applications: the 32 read/write registers of the RISC-V instruction set, and lookups into structured tables of size 2^{64} used for proving correct primitive instruction execution in Jolt [AST24]. Bits of committed data is the relevant measure of commitment costs when using binary-field hashing-based multilinear commitments like Binius [DP23, DP24] and Blaze [BCF+24]. T denotes the number of reads in the read-only case, and the number of alternating reads and writes in the read/write case ($2T$ total memory operations). We assume values read or written are 32 bits as in RISC-V, and that timestamps used in Spice are up to 32 bits (which is roughly sufficient for values of T up to 2^{30}). For Twist and Shout, we omit the field multiplications needed to prove Booleanity, as the commitment schemes themselves ensure this. For prior work (Lasso and Spice), we incorporate the slightly larger prover costs these protocols incur in the setting of small-characteristic fields [DP23, Section 4.4].

parametrized by an integer parameter $d > 0$ that helps control commitment key size (in the case of elliptic curve commitment schemes) or commitment time (in the case of hashing-based commitment schemes). The higher d is, the more non-zero values the Twist and Shout provers commit to, and the more field work the Twist and Shout provers do. With elliptic curves commitment schemes d will typically be between 1 and 4, and with hashing-based commitments d will typically be between 1 and 16. See Section 2.8 for details.

2.4.1 Proof size and verifier costs

Twist and Shout avoid invoking a grand product argument. They inherently invokes the sum-check protocol a constant number of times, leading to proofs of size $O(d \cdot (\log(T) + \log(K)))$, where T is the number of memory operations and K the size of the memory. When combined with a commitment scheme that has logarithmic-size evaluation proofs such as HyperKZG, Zeromorph, or Dory, they yield SNARKs for memory-checking whose proofs consist of $O(d(\log(T) + \log(K)))$ field and group elements. Via standard batching techniques, the verifier only needs to perform a single evaluation proof verification, which involves a constant number of MSMs of size $O(\log(T) + \log(K))$ and a constant number of pairing evaluations.

2.4.2 Prover time

Commitment costs. For read operations, both the Twist and Shout provers commit only to the address ra specified via (d-dimensional) one-hot encoding. These commitments are unavoidable, as the memory-checking problem formulation itself requires that the address to be committed. The *values* returned by any read operation, captured by the multilinear polynomial $\widetilde{\mathsf{rv}}$, need *not* be committed: $\widetilde{\mathsf{rv}}$ is a virtual polynomial.

For write operations, the Twist prover commits to the address and the value written, plus a single additional "small" value (which we call the write *increment*, a notion we will define later).

Field work. We give a variety of algorithms for quickly implementing the prover in the various sum-check invocations throughout the Twist and Shout PIOPs. Which algorithm to use depends on various aspects including: is the number of memory operations T significantly bigger than the memory size K? In the case of Twist, are reads and writes to memory local? Here, we provide a concise summary of the prover times we achieve in different contexts. We explicitly state Booleanity-checking costs in each regime (i.e., the cost of proving that various committed values lie in $\{0, 1\}$), as Booleanity-checking can be omitted when using a commitment scheme like Binius [DP23, DP24] that itself enforces Booleanity of committed values.[19]

Shout with $K = o(T)$. We hide additive terms of $O(K)$ and $O(K^{1/d} \log K)$ in this summary. The core Shout PIOP with $d = 1$ (Figure 5) costs T field multiplications for the prover (Section 6.1). The core Shout PIOP for $d > 1$ (Figure 7) costs $d^2 + 1$ field multiplications (Section 6.2). Booleanity-checking (Section 6.3) costs an additional $3dT$ field multiplications (other parts of the one-hot-encoding checking PIOP in Figure 8) contribute only low-order costs for the prover). This translates to $4T$ multiplications in total when $d = 1$, $12T$ multiplications when $d = 2$, and so forth.

Shout for gigantic structured memories. Let $C = cd$ be a constant such that $CK^{1/C} = o(T)$.[20] Then the core Shout PIOP (Figure 5) combined with one-hot-encoding-checking costs at most $\left(7C + d^2 + 3d + c + 2\right) T$ field multiplications. For example, when $C = 4$ and $d = 2$, this translates to $40T$ multiplications. Of these, $(4C + 3d)T$ are due to Booleanity checking. See Section 7.5 for details.

In Appendix C, we describe a variation of Shout with prover work that has a linear rather than quadratic dependence on d. Specifically, the d^2T term in the prover time above is reduced to $(8d - 7.5)T$. This variant is more efficient for values of d that are greater than or equal to 8. Such values of d naturally arise for gigantic lookup tables when combining Shout with hashing-based commitment schemes, but not when using commitments based on elliptic curves.

Twist costs. Again we assume $K = o(T)$ and $K^{1/d} \log K = o(T)$ and hence suppress additive terms of $O(K)$ and $O(K^{1/d} \log K)$. The core Twist PIOP (Figure 9) costs $(5 \log(K) + 2d^2 + 4d + 4)T$ field multiplications in the worst case (Section 8.3). However, when $d = 1$, this calls falls substantially for local memory accesses. Specifically, if the bulk of the memory accesses are 2^i-local (meaning they access a cell that was previously accessed at most 2^i cycles prior), then the $5 \log K$ term falls to just $7i$. Booleanity-checking costs an additional $6dT$ field multiplications (again, other parts of the one-hot-encoding checking PIOP are low-order costs).

Comparing to prior works. Figures 1–3 give precise comparisons of Twist and Shout to the previously fastest provers in the context of elliptic-curve-based commitment schemes. Twist and Shout generally improve on *both* field multiplications and commitment costs. This improvement holds even when Twist and Shout are configured to have significantly shorter proofs than the prior works.

Exactly how big a prover-time improvement is achieved by Twist and Shout depends on context. Fortunately, the biggest improvements are achieved in the two settings that are most central to the Jolt zkVM: lookups into gigantic structured tables (i.e., how Lasso is current used in Jolt to prove correct execution of primitive RISC-V instructions), and reads and writes into 32 RISC-V registers (two such reads and one such write occur in Jolt for nearly every cycle of the RISC-V CPU).

In the case of gigantic structured lookup tables, Shout improves up to 8× in commitment costs and up to about 3× in field multiplications relative to Lasso's application in Jolt today. The largest improvements are achieved when Shout is applied with $d = 2$, which is appropriate if using Dory as the polynomial commitment scheme (see Section 2.8 for details). Shout actually achieves even bigger improvements for some lookup tables,

[19] Arguably, this merely pushes the field multiplication necessary for Booleanity-checking into the polynomial evaluation protocol of the polynomial commitment scheme, as that phase itself invokes the sum-check protocol [DP24].

[20] C may be smaller than d when d is very large, e.g., $d = 16$. Such settings of C and d naturally arise when using hashing-based commitments but typically not with elliptic curve commitments.

since the costs of prior work depend on how "nicely decomposable" the table is, and that varies for the lookup tables arising in Jolt.

In the case of 32 read/write registers, where $d = 1$ is appropriate regardless of the commitment scheme used (again, see Section 2.8 for details), Twist improves over the commitment costs of Spice by about $3\times$ and the field multiplications of Spice by a factor of 2.

For larger read/write memories, we expect Twist to achieve more modest but still significant speedups over prior work (while also achieving significantly reduced proof size).

Figure 4 compares Twist and Shout to prior works in the context of hashing-based commitment schemes. For small memories, both the commitment costs and field work of Twist improve by $2\times$-$3\times$. For very large real-only memories (e.g., structured lookup tables of size 2^{64} as arise in Jolt), Shout's commitment costs improve at least $3\times$ compared to prior work, while prover field work improves more modestly. Again, Shout achieves bigger improvements for some lookup tables.

Comparison of Shout to FLI. Shout with $c = 1$ has quite a bit in common with FLI [GM24]: They both represent addresses via one-hot encoding, commit to the one-hot encodings with curve-based commitments (at the cost of one group operation per address for the prover), and observe that given the one-hot encodings, the validity of the lookups can then easily be expressed via R1CS. Unlike Shout, FLI then directly applies a Nova-type folding scheme [KST22] to fold these R1CS instances. Shout instead applies the sum-check protocol to directly prove satisfaction of the R1CS.

Inspired by Lasso [STW24], FLI also considers lookups into giant, decomposable tables. In this context, our improvements over FLI are analogous to our improvements over Lasso: by avoiding table decompositions, Shout lowers the amount of committed data (see Section 2.3 for details).

Ideas from FLI offer one possible approach to combining Shout with folding. In a nutshell, Shout can be directly turned into a folding scheme using the NeutronNova framework [KS24b], achieving better efficiency. More generally, Section 2.9.3 discusses various approaches to combining both Twist and Shout (and zkVMs that use them) with folding techniques.

Comparison of Twist to Proofs for Deep Thought. Bünz and Chen [BC24] present an incremental verifiable computation (IVC) framework for read/write memory (while FLI targets read-only memory). Their solution involves techniques reminiscent of Twist's use of increments. However, to check correctness of the increments they invoke Spice [SAGL18] and LogUp [Hab22]. The focus of Twist and Shout is precisely to eliminate the overheads of these prior memory-checking arguments. Hence, Bünz and Chen's work can be viewed as targeting how to efficiently "link" the state of memory between adjacent shards in an instance of IVC, but they incur overheads similar to the baselines we consider and improve upon.

2.5 Overview of Shout

2.5.1 The sum-check protocol in a nutshell

Let g be an ℓ-variate polynomial over field $\mathbb{F}$. From the verifier's perspective, the sum-check protocol is an efficient reduction from the task of computing

$$\sum_{x \in \{0,1\}^\ell} g(x) \tag{2}$$

to the potentially easier task of evaluate $g(r)$ for a $r \in \mathbb{F}^\ell$ chosen at random by the verifier over the duration of the protocol. So long as g has constant degree in each of its variables, the proof length of the sum-check protocol is $O(\ell)$ field elements, and the soundness error is $O(\ell/|\mathbb{F}|)$.

2.5.2 The Shout PIOP when $d = 1$

Shout is targeted at read-only memories. Let $\mathsf{Val}(k)$ denote the fixed value of register k. Shout with $d = 1$ is a special case: to obtain the best possible prover costs, our protocol for $d = 1$ deviates substantially from the

general case of $d > 1$.

As observed in Baloo [ZGK+22] and Lasso [STW24], the lookups are all valid if and only if for every cycle j,

$$\sum_{\text{registers } k} \mathsf{ra}(k, j)\mathsf{Val}(k) = \mathsf{rv}(j). \tag{3}$$

Indeed, since $\mathsf{ra}(k, j) = 1$ for the register k that was read at cycle j (and $\mathsf{ra}(k', j) = 0$ for all other registers $k' \neq k$), Equation (3) holds if and only if $\mathsf{rv}(j)$ equals $\mathsf{Val}(k)$. To check that these constraints are satisfied, the verifier picks a random $r_{\text{cycle}} \in \mathbb{F}^{\log T}$, and the sum-check protocol is applied to confirm that:

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r_{\text{cycle}}) \cdot \widetilde{\mathsf{Val}}(k). \tag{4}$$

Here, $\widetilde{\mathsf{ra}}$, $\widetilde{\mathsf{rv}}$, and $\widetilde{\mathsf{Val}}$ are the *multilinear extension polynomials* of the vectors ra, rv, and Val respectively,

Indeed, the right hand side of Equation (4) is a multilinear polynomial in r' and this polynomial agrees with $\widetilde{\mathsf{rv}}$ at all $r' \in \{0,1\}^{\log T}$ if and only if all constraints from Expression (7) are satisfied. Hence, the right hand side and left hand side of Equation (8) are equal as formal polynomials if and only if all constraints are satisfied. By the Schwartz-Zippel lemma, up to soundness error $\log(T)/|\mathbb{F}|$, checking that Equation (8) holds at a randomly chosen $r' \in \mathbb{F}^{\log T}$ is equivalent to checking all constraints are satisfied.

At the end of the sum-check protocol, the verifier has to evaluate $\widetilde{\mathsf{ra}}(r_{\text{address}}, r_{\text{cycle}})$, $\widetilde{\mathsf{Val}}(r_{\text{address}})$. for random values $r_{\text{address}} \in \mathbb{F}^{\log K}$ and $r_{\text{cycle}} \in \mathbb{F}^{\log T}$ chosen by the verifier. The verifier can can obtain the $\widetilde{\mathsf{ra}}(r_{\text{address}}, r_{\text{cycle}})$ evaluation from the commitment to that polynomial. For many lookup tables (including all arising in the Jolt zkVM [AST24]), the verifier can evaluate $\widetilde{\mathsf{Val}}(r_{\text{address}})$ on its own in $O(\log K)$ time. Following Lasso [STW24], we call such tables *MLE-structured*. For non-MLE-structured tables, $\widetilde{\mathsf{Val}}$ can be committed in advance by an honest party, and the verifier can force the prover to provide the necessary evaluation of $\widetilde{\mathsf{Val}}$.

This is the entire core Shout PIOP for $d = 1$. It is essentially just the Generalized-Lasso protocol from [STW24], but without using the Spark sparse polynomial commitment scheme to commit to the $\widetilde{\mathsf{ra}}$ polynomial. Avoiding Spark is important, as Spark requires committing to several random values per lookup, which is a prover bottleneck both concretely and asymptotically.

The core Shout PIOP sketched above assumes all committed addresses are valid one-hot encodings of values in $\{0, 1, \ldots, K-1\}$. In Section 4.1.2, we give a separate PIOP for confirming this is the case. This PIOP amounts to identifying a straightforward (very large, but sparse) constraint system capturing correctness of one-hot encodings, and showing that the satisfaction of all constraints can be proved in time proportional to the sparsity of the constraint system rather than the size.

2.5.3 The Shout PIOP when $d > 1$

A more expensive but generalizable protocol for $d = 1$. The $d = 1$ case is special because only when $d = 1$ is the right hand side of Equation (4) multilinear in r_{cycle}. This allowed us to avoid a sum over $j \in \{0,1\}^{\log T}$ on the right hand side of Equation (4). Before covering the $d > 1$ case, we give a slightly *different* PIOP for the $d = 1$ case that *does* sum over $j \in \{0,1\}^{\log T}$. This PIOP is less efficient than the one in Section 2.5.2, but generalizes straightforwardly to $d > 1$.

Recall from Equation (3) that checking that all reads are correct is equivalent to confirming that for all cycles j, the following constraint is satisfied:

$$\sum_{\text{registers } k} \mathsf{ra}(k, j)\mathsf{Val}(k) = \mathsf{rv}(j).$$

To check that this, the verifier can pick a random $r' \in \mathbb{F}^{\log T}$ and apply the sum-check protocol to confirm that:

$$\widetilde{\mathsf{rv}}(r') = \sum_{k \in \{0,1\}^{\log K},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', j) \cdot \widetilde{\mathsf{ra}}(k, j) \cdot \widetilde{\mathsf{Val}}(k). \tag{5}$$

Here, $\widetilde{\mathsf{ra}}$, $\widetilde{\mathsf{rv}}$, and $\widetilde{\mathsf{Val}}$ are the multilinear extension polynomials of the vectors ra, rv, and Val respectively, while $\widetilde{\mathsf{eq}}$ is the multilinear extension of a standard function known as the equality function (see Section 3.2 for details).

At the end of the sum-check protocol, the verifier has to evaluate $\widetilde{\mathsf{eq}}(r', r_{\text{cycle}})$, $\widetilde{\mathsf{ra}}(r_{\text{address}}, r_{\text{cycle}})$, $\widetilde{\mathsf{Val}}(r_{\text{address}})$ for random values $r_{\text{address}} \in \mathbb{F}^{\log K}$ and $r', r_{\text{cycle}} \in \mathbb{F}^{\log T}$ chosen by the verifier over the course of the protocol. The verifier can compute the $\widetilde{\mathsf{eq}}$ evaluations on its own in $O(\log K + \log T)$ time, and the remaining evaluations are obtained exactly as in Section 2.5.2.

Obtaining a fast prover in this application of the sum-check protocol is a challenge that did not arise in the simpler application of Section 2.5.2. The issue is that the sum in Equation (5) is over $K \cdot T$ terms, so naively implemented, the prover in this application of sum-check would perform $O(KT)$ field operations. This would make Shout more expensive than prior lookup arguments, except for very small memories (say, $K \leq 10$ or so).

Fortunately, Equation (5) is a *very* special application of sum-check. For example, $\widetilde{\mathsf{ra}}(k, j)$ is sparse: only T out of KT of its evaluations are non-zero over the domain $\{0,1\}^{\log K} \times \{0,1\}^{\log T}$. And there's additional structure on top of that, such as the fact that $\widetilde{\mathsf{Val}}$ depends only on k and not on j. Leveraging all of this structure, one can implement the sum-check prover with just $O(K) + 5T$ field operations (up to low-order terms).

Shout for $d > 1$. Fix an integer $d > 1$ and let $N = K^{1/d}$. We view the memory of size K as a d-dimensional cube of length N in each dimension, indexing the cube by $[N]^d$ where $[N] = \{0, 1, \ldots, N-1\}$. As in Section 2.2 for a sequence of T reads into memory, let $\mathsf{raf}(j) \in [K] \subseteq \mathbb{F}$ denote the address read by the j'th memory operation, with the address specified as a single field element. Reinterpret $\mathsf{raf}(j)$ as an element of $[N]^d$ using the natural bijection between $[K]$ and $[N]^d$, and for $i = 1, \ldots, c$, let $\mathsf{ra}_i(\cdot, j)$ denote the one-hot encoding of the i'th coordinate of $\mathsf{raf}(j)$, i.e., $\mathsf{ra}_i(k, j)$ equals 1 if the i'th coordinate of $\mathsf{ra}(j)$ equals k, and equals 0 for all other $k \in [N]$. We call this the *d-dimensional one-hot encoding* of $\mathsf{raf}(j)$.

The Shout prover commits to the d multilinear extension polynomials $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$. This entails committing to dN field elements per cycle, of which exactly d per cycle are 1, and the rest of 0.

Because only $dN = dK^{1/d}$ rather than K field elements are committed, the size of the commitment key (when using a polynomial commitment scheme like HyperKZG) falls from K group elements to dN group elements. For example, by setting $d = 2$, a HyperKZG commitment key of size $2 \cdot 2^{20}$ suffices to prove 2^{20} reads into a memory of size $K = 2^{20}$, and this commitment key is under 100 MBs in size. This can be reduced even further at the cost of a small increase in verifier costs via standard batching techniques (see Section 3.1.1).

Then the Shout PIOP applies the sum-check protocol to compute the following variation of Equation (5):

$$\widetilde{\mathsf{rv}}(r') = \sum_{k=(k_1,\ldots,k_d) \in (\{0,1\}^{\log N})^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', j) \cdot \left(\prod_{\ell=1}^{d} \widetilde{\mathsf{ra}}_\ell(k_\ell, j) \right) \cdot \widetilde{\mathsf{Val}}(k). \tag{6}$$

The right hand side of Equation (6) has degree three in each variable of k exactly as in Equation (5), but the degree in each variable of j increases from 3 in Equation (5) to $2 + d$. This degree increase raises the prover's time to $O(K + d^2 \cdot T)$. In Appendix C, we give a variation of Shout that reduces the prover's runtime dependence of d from quadratic to linear. It comes with worse constant factors, and hence concretely offers an improvement only for $d \geq 8$. We suspect that additional improvements for large values of d are possible.

Shout for gigantic, structured tables. In fact, for "structured" tables (including all of those arising in the Jolt zkVM), we can invoke the sparse-dense sum-check protocol [STW24, Appendix G] to nearly remove the dependence on K. If K is very large (i.e., $K \approx T^C$ for $C > 1$) the sum-check prover can in fact be implemented in time $O(CK^{1/C} + CT) = O(CT)$. We further explain how to generalize the sparse-dense sum-check protocol to achieve a similar time bound for proving that all committed addresses are correct one-hot encodings of values in $\{0, 1, \ldots, K-1\}$.

Remark 1 (The relationship between Shout and Generalized-Lasso)**.** *Shout with $d = \log K$ (the largest possible value of d) is roughly equivalent to running the Generalized-Lasso protocol of [STW24], but using "Spark-naive" [Set20, Section 7.1] instead of Spark [Set20, Section 7.2] as the sparse polynomial commitment scheme to commit to the one-hot encodings of addresses (see also [STW24, Appendix D] for an overview of Spark-naive).*

Indeed, Spark-naive commits to the binary representation *of each address (i.e., the index of each non-zero entry of the sparse vector being committed). This is very similar to what Shout does when $d = \log K$.*

What's going on is that Shout has the prover commit to addresses using the one-hot encoding of the digits of the address when represented in base $K^{1/d}$. *Spark-naive represents addresses in base-2 (i.e., binary), which matches the base used by Shout when $d = \log K$. Prior works like Spark-naive do not use one-hot encodings. However, the standard encoding and one-hot encoding of a value* almost coincide *for base-2: in both cases each bit in the binary representation of an address is represented in the encoding with one or two bits.*

Setting d in Shout to its maximum value of $\log K$ minimizes the number of committed bits for the prover, leading to the fastest possible commitment computation when using a binary-field hashing-based commitment scheme like Binius [DP23, DP24]. However, the downsides of taking d this big outweigh the advantages: the prover time "outside of committing" is superlinear (i.e, at least $O(T \log K)$ field multiplications) and the proof size is somewhat big (i.e, $O(\log^2(K) + \log(T)\log(K))$ field elements).

The fastest prover in Shout (as well as short proofs) is obtained by taking d to be as small as possible, subject to the appropriate constraints.

2.6 Overview of Twist

Recall that there are $2T$ memory operations, which alternate between reads and writes (say, with the first operation being a read). It is convenient to number both the read-cycles and the write-cycles from 0 up to T.

A first attempt. We first explain a somewhat naive approach to memory-checking, which has high costs, before explaining how to lower the costs.

Recall the memory has size K, and let $[K] = \{0, \ldots, K-1\}$. We can have the prover commit to the value, $\mathsf{Val}(k, j)$ of every register $k \in [K]$ for every cycle $j \in [T]$. We do have to find a way to ensure that the committed values are actually correct, i.e., that the committed value $\mathsf{Val}(k, j)$ is indeed the value most recently written to register k when cycle j is executed. We address how to do this later in our overview.

Once the $\mathsf{Val}(k, j)$ values are committed and confirmed to be correct, one can force rv_j to equal the value stored in cell ra_j at cycle j by confirming that

$$\sum_{k \in [K]} \mathsf{ra}(k, j) \cdot \mathsf{Val}(k, j) = \mathsf{rv}(j) \text{ for all cycles } j \in [T]. \tag{7}$$

If register k is the (unique) register read at cycle j, then $\mathsf{ra}(k, j) = 1$ and $\mathsf{ra}(k', j) = 0$ for all $k' \neq k$ with $k' \in \{0, 1\}^{\log K}$, and Equation (7) is therefore satisfied if and only if $\mathsf{rv}(j) = \mathsf{Val}(k, j)$, i.e., the value returned by the read at cycle j equals the value stored in register k at that cycle. To check that these constraints are satisfied, the verifier picks a random $r' \in \mathbb{F}^{\log T}$, and the sum-check protocol is applied to confirm that:

$$\widetilde{\mathsf{rv}}(r') = \sum_{(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', j) \cdot \widetilde{\mathsf{ra}}(k, j) \cdot \widetilde{\mathsf{Val}}(k, j). \tag{8}$$

Indeed, the right hand side is multilinear in r' and agrees with the left hand side at all $r' \in \{0,1\}^{\log T}$ if and only if Equation (7) is satisfied. Hence, the right hand side and left hand side are equivalent as formal polynomials if and only if Equation (7) is satisfied, and up to soundness error $\log(T)/|\mathbb{F}|$ it suffices to check equality at a single random point $r' \in \mathbb{F}^{\log T}$.

We call this invocation of sum-check the *read-checking sum-check*. It is very similar to the core Shout PIOP (Section 2.5.2). It is not quite as structured as the sum-check instance arising in Shout, but we are nonetheless able to show how to implement the prover in $O(K + T \log K)$ time, an exponential improvement in the dependence on K compared to a naive $O(KT)$-time prover implementation. We do this by leveraging two properties. First, that $\mathsf{ra}(k, j)$ is sparse: only T out of kT of its entries are non-zero. Second, that when moving from cycle j to $j + 1$, only one entry of $\mathsf{Val}(k, j)$ changes (namely, the register written at cycle j).

At the end of the read-checking sum-check, the verifier needs to evaluate $\widetilde{\mathsf{rv}}$ at a random point $r_{\text{cycle}} \in \mathbb{F}^{\log T}$, $\widetilde{\mathsf{Val}}$ at a random point $(r_{\text{address}}, r_{\text{cycle}}) \in \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$, and $\widetilde{\mathsf{eq}}$ at $(r', r_{\text{cycle}}) \in \mathbb{F}^{\log T} \times \mathbb{F}^{\log T}$. The $\widetilde{\mathsf{eq}}$ evaluation can be computed by the verifier directly in $O(\log T)$ time. The verifier can obtain the evaluations of $\widetilde{\mathsf{rv}}$ and $\widetilde{\mathsf{Val}}$ from the commitments to these polynomials.

The key performance issue with the above proposal is the cost of committing to the vector $\mathsf{Val}(k, j)$ (or more precisely, its multilinear extension $\widetilde{\mathsf{Val}}$). This vector specifies all register values at all cycles means committing to K non-zero values per cycle. Even for very small memories like $K = 32$, this alone is more expensive than Spice, which commits to only 7 small values per read and 8 per write.

Enter increments: virtual polynomials to the rescue. To avoid this, we use an idea implicit in many works involving the sum-check protocol, and explicit in Binius, which refers to the notion of "virtual polynomials" [DP23]. These are polynomials that are *not* committed by the prover, but that the verifier can "pretend" are committed. We already discussed virtual polynomials in Section 2.2, in the context of evaluating the multilinear polynomials $\widetilde{\mathsf{raf}}$ and $\widetilde{\mathsf{waf}}$ (which specify addresses via a single field element rather than one-hot encoding).

In more detail, if the sum-check protocol is applied to a polynomial derived from p, then at the end of the sum-check protocol the verifier has to evaluate p at a random point r. If p were committed with a polynomial commitment scheme, then $p(r)$ could be provided by the prover along with a proof that the provided evaluation is correct.

However, if p is expensive to commit to directly, the prover will not want to commit to p. The idea of virtual polynomials is to have the prover commit to p *indirectly* as follows. Suppose there is some alternative ℓ-variate polynomial q that is cheaper to commit to than p. Moreover for any input r to p, there is some polynomial g_r (which depends on r) such that the following two properties hold: (1) $p(r) = \sum_{x \in \{0,1\}^\ell} g_r(x)$, and (2) For any point r', the verifier can quickly derive $g_r(r')$ from $q(r')$.

Then when the verifier needs to evaluate $p(r)$, it can simply apply the sum-check protocol to g_r, at the end of which the verifier needs to evaluate $g_r(r')$ for a random point r'. And by (2) above, for this, it suffices for the verifier to learn $q(r')$. Since q *was* committed by the prover, the prover can provide this evaluation directly.

In the case of Twist, the polynomial $p(k, j) = \widetilde{\mathsf{Val}}(k, j)$ is expensive to commit to directly, since it requires committing to K non-zero values per cycle j. Instead, let us define the polynomial $q(k, j) = \widetilde{\mathsf{Inc}}(k, j)$ as the (multilinear extension of) the *increments* of Val. Here, we define the increment of register k at cycle $j \in [T]$ to be the difference between the register's value at cycle j vs. cycle $j - 1$. In other words,

$$\mathsf{Inc}(k, j) \coloneqq \mathsf{Val}(k, j + 1) - \mathsf{Val}(k, j) = \mathsf{wa}(k, j) \cdot (\mathsf{wv}(k, j) - \mathsf{Val}(k, j)) \,. \tag{9}$$

Here, the final equality holds by the following reasoning. If register k is not written at cycle j, then $\mathsf{wa}(k, j) = 0$, and that forces $\mathsf{Inc}(k, j)$ to 0 per the right hand side of Equation (9). While if register k is written at cycle j, then $\mathsf{wa}(k, j) = 1$, and the right hand side of Equation (9) forces $\mathsf{Inc}(j)$ to be the difference between $\mathsf{wa}(k, j)$, i.e., the value written at cycle j, and $\mathsf{Val}(k, j)$, the value stored at that cell at the start of the cycle.

The key point is that, since only one register is written per cycle j, $\mathsf{Inc}(k, j)$ is non-zero for only one register k. Moreover, since registers contain 32-bit values, the one non-zero increment at each cycle j is "small" (i.e., in $\{-2^{32}-1, \ldots, 2^{32}-1\}$), and hence fast to commit to. This makes $\widetilde{\mathsf{Inc}}$ roughly K times cheaper to commit to than $\widetilde{\mathsf{Val}}$.

Giving the verifier query-access to $\widetilde{\mathsf{Val}}$. But now we run into the key issue arising from the use of virtual polynomials: recall that at the end of the zero-check for correctness of read operations, the verifier has to evaluate $\widetilde{\mathsf{Val}}(r_{\text{address}}, r_{\text{cycle}})$ for a random point

$$(r_{\text{address}}, r_{\text{cycle}}) \in \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}. \tag{10}$$

Since $\widetilde{\mathsf{Val}}$ is not committed directly, it is not obvious how the verifier can obtain this evaluation. The way to do it is to apply the sum-check protocol to the following equation:

$$\widetilde{\mathsf{Val}}(r_{\text{address}}, r_{\text{cycle}}) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(r_{\text{address}}, j') \cdot \widetilde{\mathsf{LT}}(j', r_{\text{cycle}}). \tag{11}$$

We call this the $\widetilde{\mathsf{Val}}$-*evaluation sum-check*. Here, LT is the multilinear extension of the function the so-called *less-than* function, which takes as input two binary strings $j', j \in \{0,1\}^{\log T}$ and outputs 1 if and only if the integer represented by j' is *strictly* less than the integer represented by j. One can check that Equation (11) holds because the right hand side is multilinear in the variables of r_{address} and r_{cycle}, and agrees with the right hand side whenever $(r_{\text{address}}, r_{\text{cycle}}) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$, owing to the fact that the value of any register k at time j is the sum of the increments for that register at all cycles $j' < j$.

At the end of the $\widetilde{\mathsf{Val}}$-evaluation sum-check, the verifier needs to evaluate $\widetilde{\mathsf{Inc}}$ at a random point and $\widetilde{\mathsf{LT}}$ at a random point. The former evaluation can come from the commitment to $\widetilde{\mathsf{Inc}}$, while the latter the verifier can perform on its own in time $O(\log T)$ (see Section 3.2 for details).

The remaining issue. We have deferred until now discussion of how to ensure that each value $\mathsf{Val}(k, j)$ is the actual value stored in register k at cycle j. Given that $\mathsf{Val}(k, j)$ is in fact implicitly specified by $\mathsf{Inc}(k, j)$ per Equation (11), this problem manifests as ensuring that the committed polynomial $\widetilde{\mathsf{Inc}}$ indeed equals (the multilinear extension of) the *correct increments*, per the definition in Equation (9).

To check this, we apply the zero-check PIOP once again, to confirm that indeed for all $k \in \{0,1\}^{\log K}$ and $j \in \{0,1\}^{\log T}$,

$$\mathsf{Inc}(k, j) = \mathsf{wa}(k, j) \cdot (\mathsf{wv}(j) - \mathsf{Val}(k, j)) .$$

This involves the verifier picking a random $r' \in \mathbb{F}^{\log K}$ and $r'' \in \mathbb{F}^{\log T}$ and applying the sum-check protocol to confirm that

$$\widetilde{\mathsf{Inc}}(r', r'') = \sum_{(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', k) \cdot \widetilde{\mathsf{eq}}(r'', j') \cdot \left(\widetilde{\mathsf{wa}}(k, j)(\widetilde{\mathsf{wv}}(j) - \widetilde{\mathsf{Val}}(k, j)) \right). \tag{12}$$

We call this the *write-checking sum-check*.

At the end of the write-checking sum-check, the verifier needs to evaluate each of $\widetilde{\mathsf{Inc}}$, $\widetilde{\mathsf{wa}}$, $\widetilde{\mathsf{wv}}$, and $\widetilde{\mathsf{Val}}$ each at a random point. Since $\widetilde{\mathsf{Inc}}$, $\widetilde{\mathsf{wa}}$ and $\widetilde{\mathsf{wv}}$ are committed by the prover already, the required evaluations of these polynomials can be obtained directly from the prover. The evaluation of $\widetilde{\mathsf{Val}}$ can be obtained via another invocation of the $\widetilde{\mathsf{Val}}$-evaluation sum-check. In fact, by running the read-checking and write-checking sum-check instances in parallel, we can ensure only one evaluation of $\widetilde{\mathsf{Val}}$ is needed, and hence we only need to run the $\widetilde{\mathsf{Val}}$-evaluation once.

Summary of the protocol. Twist had to address three issues:

- How to "implicitly commit" to $\widetilde{\mathsf{Val}}$ given that explicitly committing to $\widetilde{\mathsf{Val}}$ would be expensive (requiring K committed non-zero values per cycle).
- How to confirm that $\widetilde{\mathsf{Val}}$ actually assigns the correct value to each register at each cycle, given the committed sequence of write operations specified by $\widetilde{\mathsf{wa}}$, and $\widetilde{\mathsf{wv}}$. In other words, how to make sure the prover processed all the writes correctly.
- How to ensure that each read operation (specified by the committed polynomial $\widetilde{\mathsf{ra}}$ and the virtual polynomial $\widetilde{\mathsf{rv}}$) returns the correct value. In other words, how to make sure the reads were processed correctly.

The first issue was addressed by committing to the *increments* $\widetilde{\mathsf{Inc}}$ rather than to $\widetilde{\mathsf{Val}}$. Committing to $\widetilde{\mathsf{Inc}}$ is cheap because $\widetilde{\mathsf{Inc}}$ is sparse and its non-zero entries are small.

The second issue (checking writes) was addressed via the write-checking sum-check (Equation (12)).

The third issue (checking reads, and giving the verifier query access to the virtual polynomial $\widetilde{\mathsf{rv}}$) was addressed by the read-checking sum-check (Equation (8)).

At the end of the read-checking and writing-checking sum-checks, the verifier has to evaluate $\widetilde{\mathsf{Val}}$ at a random point. This is addressed via the $\widetilde{\mathsf{Val}}$-evaluation sum-check (Equation (11)). This equation expresses the necessary $\widetilde{\mathsf{Val}}$ evaluations in terms of $\widetilde{\mathsf{Inc}}$ as per Equation (11).

2.7 Other benefits and implications

Improved soundness error. Offline memory-checking techniques introduce soundness error at least $(T + K)/|\mathbb{F}|$ where T is the number of memory operations proven and K is the size of the memory, thanks to the grand product in Equation (1) introducing a univariate polynomial in the random field element r, of degree $\Theta(T + K)$. Meanwhile, the method of one-hot addressing has soundness error of only $\log(TK)/|\mathbb{F}|$.

This difference is not important when working over 256-bit prime order fields, as required by most standard elliptic curves. However, there may be contexts where it is relevant. For example, consider elliptic curves defined over extensions of 64-bit prime fields and targeted at a security level of 128 bits [SSS22]. To keep as much of the prover's work as possible over a relatively small subfield (thereby keeping the field operations relatively fast), SNARK designers will want to choose the value r from Equation (1) at random from a 128-bit subfield (i.e., a degree-2 extension of the 64-bit base field). However, if the memory size K is, say, one billion, this will lead to only at most 98 bits of security, due to the soundness error of at least $2^{30}/2^{128} = 2^{-98}$. Processing a trillion cycles (equivalent to a couple of minutes of single-threaded compute on a modern laptop), at least without invoking SNARK recursion, would lead to an even lower security level, of 2^{-88}. The method of one-hot addressing, by contrast, achieves over 120 bits of security in these settings.

Benefiting from small memories and locality. Modern CPUs can make random accesses to a large memory. But they also have smaller, faster memories. At the extreme end are *registers*, which are very fast memory cells directly accessible to the processor. In contrast, offline memory checking techniques are not substantially more efficient when applied to small memories. That is, the cost of proving T reads and writes into a memory of size K is roughly $O(T + K)$ committed values and field operations, and hence this cost is about the same for all all $K \leq T$. In other words, zkVM provers do not currently enjoy the benefits of small, fast memories like registers or caches, the way actual CPUs do. The cost profile of Twist matches that of actual CPUs much more closely. For starters, the smaller the memory, the smaller the parameter d can be in both Twist and Shout, and the smaller d is, the lower the commitment costs and prover field multiplications in both Twist and Shout. Furthermore, the Twist prover (appropriately implemented, see Section 8 for details) performs many fewer field multiplications for *local* memory accesses–reads and writes to memory cells that were recently read from or written to.

The closer the prover cost profile of zkVMs matches that of actual CPUs, the more likely it is that existing compilers (optimized for actual CPUs rather than SNARK provers) lead to fast zkVM proving. In other words,

Twist and Shout offer a strong counterpoint to a prevailing belief that it is better for zkVM performance to design "SNARK-friendly" VMs, rather than using mature toolchains for existing VMs like RISC-V that were not designed with SNARK proving in mind.

2.8 Appropriate settings of d

How to set d when using elliptic curve commitments. When applying Twist and Shout with an elliptic curve commitment scheme, there are multiple tools at the protocol designer's disposal to control commitment key size. One is the precise choice of polynomial commit scheme. For example, Dory has square-root sized commitment key, while HyperKZG is linear-sized commitment key. So using Dory rather than HyperKZG helps to control commitment key size. Of course, the choice of Dory comes with its own costs, such as concretely larger polynomial evaluation proofs and the need to do (in addition to MSMs) a multi-pairing of size $\sqrt{n}$ when committing to a polynomial of size n.

Another tool is the ability to reduce commitment key size by any desired factor k at the cost of increasing commitment size from one group element to k (see Section 3.1.1 for details).

As a separate and complementary tool, we also introduced the parameter d to Twist and Shout themselves. If one uses parameter d with Dory as the commitment, the commitment key size is $\Theta\left((K^{1/d} \cdot T)^{1/2}\right)$ when proving T accesses to a memory of size K. If one uses parameter d and HyperKZG as the commitment, the commitment key size is $\Theta(K^{1/d} \cdot T)$.

Generally speaking, for commitment schemes based on elliptic curves, one wants to set d as small as possible subject to the constraint that the commitment key is not too large for the prover to generate and store (the key must be big enough to commit to d vectors of length $N = K^{1/d}$).

How to set d when using hashing-based commitments. For commitment schemes based on hashing over binary fields like Binius, one also wants to set d as small as possible, but subject to a very different constraint. Since 0s are not free to commit to with hashing-based schemes, the key constraint is ensuring that committing is sufficiently fast. The larger d is, the faster it is to commit to addresses in Twist and Shout with a binary-field hashing-based commitment scheme, but the more work the prover does in the sum-check protocol (and the larger the proof size). Hence, with hashing-based commitment schemes, d should be chosen large enough that committing to addresses in Twist and Shout is not the prover bottleneck, but no larger.

For illustration, we now discuss how one can set d in various applications covering the gamut of memory sizes, from tiny to large.

Application 1: $K = 32$, $T = 2^{20}$. Recall that zkVMs today break CPU executions into "shards" consisting of roughly 2^{19} cycles each. Also, the RISC-V ISA has 32 registers, with each cycle reading two registers and writing to one. Hence, one important application is to $K = 32$ with $T \approx 2^{21}$. In this case, setting $d = 1$ in Twist and using HyperKZG leads to a commitment key (powers-of-tau SRS) containing $32 \cdot 2^{21} = 2^{26}$ group elements. This is smaller than some powers-of-tau SRSes that have already been generated (which have contained up to 2^{29} powers of tau[21]), though it does require GBs of prover storage. Per Section 3.1.1, one can further lower SRS size, say to 2^{21}, by increasing the size of the commitment from one group element to $2^5 = 32$ group elements. Or one could keep the commitment size at one group element, but use $d = 2$, obtaining an SRS size of only about 2^{11} group elements.

With a hashing-based commitment scheme over $\mathsf{GF}(2^{128})$ like Binius or FRI-Binius [DP23, BCF+24] or Blaze [BCF+24], the key issue to address is not commitment key size, but rather the time cost of committing to each address (as 0s are not free to commit to with these schemes). For this application, the memory is sufficiently tiny that one can set $d = 1$ and maintain small commitment costs. This requires the prover to commit to 32 values in $\{0, 1\}$ per address. Four such addresses could be packed into a single $\mathsf{GF}(2^{128})$ before committing.

[21] See for example `https://github.com/privacy-scaling-explorations/perpetualpowersoftau`.

Application 2: $K = 2^{20}$, $T = 2^{20}$. This captures VMs with several MBs of main memory (alternatively, it captures typical L2 or even L3 cache sizes on modern CPUs). With HyperKZG, one cannot take $d = 1$, as this would lead to an SRS of size $KT = 2^{40}$. However, setting $d = 4$ is feasible, leading to an SRS size of $K^{1/d} \cdot T = 2^{25}$ with address commitments consisting of $d = 4$ group elements. As above, this SRS size can be driven lower with a further increase in commitment size. Using Dory instead of HyperKZG, one can take $d = 1$, leading to a commitment key of size $2 \cdot \sqrt{KT} = 2^{21}$ group elements in a pairing-friendly group (2^{20} from $\mathbb{G}_1$ and 2^{20} from $\mathbb{G}_2$).

As discussed in Section 1, $d = 4$ is also a reasonable setting in this application if using a hashing-based multilinear commitment scheme over $\mathsf{GF}(2^{128})$, such as Binius or FRI-Binius [DP23, BCF+24] or Blaze [BCF+24]. Then each each address is specified by $dK^{1/d} = 4 \cdot 2^{20/4} = 2^7 = 128$ bits, which can all be packed into a single $\mathsf{GF}(2^{128})$ field element before committing.

Application 3: $K = 2^{30}$, $T = 2^{32}$. We anticipate that techniques for controlling prover space *without* invoking SNARK recursion [NT25] will eventually enable efficient non-recursive proving of $T = 2^{30}$ or more RISC-V cycles. $K = 2^{30}$ corresponds to giving the VM about 4 GBs of main memory (RAM).

Applying Twist to prove $T = 2^{32}$ memory accesses is incompatible with commitment schemes like HyperKZG requiring a linear-size SRS. But with Dory, one could set $d = 2$ and obtain a commitment key of size $2 \cdot \sqrt{K^{1/d} \cdot T} \approx 2^{26}$ group elements. This could be further lowered per Section 3.1.1 at the cost of increasing the size of the commitments.

Application 4: $K = 2^{64}$, $T = 2^{20}$ **(structured read-only memory).** This captures "primitive instruction execution" lookups in Jolt. With HyperKZG, one could set $d = 8$ and have address commitments consist of $4c = 32$ group elements. This would lead to an SRS with $K^{1/d}/4 \cdot T = 2^{26}$ group elements, which is significantly smaller than some powers-of-tau SRSes that have been generated today. With Dory, setting $d = 4$ would lead to a (transparently generated) commitment key of size just $2\sqrt{K^{1/d} \cdot T} = 2 \cdot 2^{18} = 2^{19}$. One could even take $d = 2$, which would give a commitment key of size 2^{27}, and this key size could be further reduced at the cost of larger commitments per Section 3.1.1.

If using a hashing-based commitment scheme over $\mathsf{GF}(2^{128})$, a reasonable setting of d is 16. Then the number of bits in each committed address is $d \cdot K^{1/d} = 16 \cdot 2^{64/16} = 256$ bits. These can be packed into two $\mathsf{GF}(2^{128})$ field elements before committing.

2.9 Additional discussion

2.9.1 Pros and cons of Dory

Dory is an especially attractive choice of polynomial commitment scheme for combination with Twist and Shout. This is primarily because it is a curve-based commitment scheme with a sublinear-size commitment key, and this helps keep the parameter d in Twist and Shout small (which in turn ensures very few committed values, and minimal field work for the prover). Implementations of Dory exist[22], but Dory has not seen wide deployment to date. We briefly discuss the downsides of Dory that are the likely reasons it hasn't been widely deployed, and argue that these aspects of Dory do not preclude its use in contexts where its cost profile is particularly valuable. We believe Twist and Shout offer the most compelling such context to date.

One potential downside of Dory is that when committing to a polynomial of size N, computing the commitments involves (in addition to $\sqrt{N}$ multi-exponentiations of size $\sqrt{N}$, which is roughly equivalent in cost to other elliptic curve commitment schemes like KZG or HyperKZG, Bulletproofs/IPA, etc.), a multi-pairing of size $\sqrt{N}$. For small values of N the multi-pairing can be a prover bottleneck. But for moderate-to-large values of N, this will not be the case. Indeed, a pairing evaluation is equivalent in cost to at most about 4000 group operations, and multi-pairings also benefit from a "Pippenger's speedup" meaning a multi-pairing of size $\sqrt{N}$ is actually a factor of $\mathsf{pip} \approx (1/2)\log(N)$ faster than $\sqrt{N}$ independent pairings.

[22] `https://github.com/yacovm/DualDory`

This means that if Shout is applied (even with $d = 1$) to, say, $T \geq 2^{22}$ memory operations, the multi-pairing will not be a prover bottleneck, as the multi-pairing will be equivalent in cost to $4000 \cdot \sqrt{T}/\mathsf{pip} \ll 2^{22}$ group operations, while the MSMs involved in the commitment will cost this much or more. Furthermore, for any $t > 1$, Dory can be configured so the multi-pairing is only of size $\sqrt{N}/t$, at the cost of increasing the commitment key size to $t\sqrt{N}$.

Another aspect of Dory that may be construed as a downside is that its proofs consists of $6 \log N$ elements of the target group of a pairing-friendly group, translating to roughly a dozen KBs. This is still much smaller than the proof size of hashing-based commitment schemes like FRI. And proof size can be reduced via SNARK composition.

A final potential downside of Dory is verifier *time*: verifying Dory evaluation proofs requires (logarithmically many) scalar multiplications in the target group of a pairing-friendly elliptic curve, and these can be slow, as, naively implemented, target group operations are about 6 times slower than $\mathbb{G}_1$ operations. If this indeed turns out to be a significant verifier cost, there are multiple possible mitigations. First, as with proof size, verifier time can be reduced via SNARK composition. Second, even short of SNARK composition, one could outsource the target group operations to the prover using, say, data parallel variants of the GKR protocol [GKR15, Tha13]. Third, there has been recent progress on speeding up target-group operations in BLS12-381, at least on high-end consumer CPUs.[23] Such techniques could keep Dory's verification time from becoming problematic even in the absence of SNARK composition or outsourcing verification.

We expect the benefits of Dory in the context of Twist and Shout (especially controlling commitment key size while keeping d small) to outweigh any downsides.

Section 9.2.3 contains yet more discussion of Dory's costs and how it works.

2.9.2 Twist and Shout design philosophy

The design of Twist and Shout reflects a philosophy that leverages the unique strengths of the sum-check protocol to achieve high prover performance. Below, we highlight key aspects of this approach and its broader implications for SNARK design.

Barely paying for zeros: a superpower of sum-check (and elliptic curve commitments). The method of one-hot addressing leans heavily on the ability of elliptic-curve commitment schemes to commit to 0-values for "free" (i.e., committing to a vector incurs prover costs proportional to its *sparsity*, i.e., number of non-zeros, not its length). The method also works well with hashing-based commitments over binary fields such as Binius [DP23, DP24], since such schemes ensure that it is cheap to commit to values known a priori to reside in $\{0, 1\}$. Importantly, these commitment schemes "pack" many small values into a single field element, and later use use the sum-check protocol to relate the packed field elements to unpacked ones.

Indeed, committing very quickly to 0s (or small values in general) is not enough on its own to get a fast SNARK prover. Also essential is the ability of the sum-check protocol to allow the prover to avoid paying *field operations* to process 0-values. This appears difficult or impossible to achieve with polynomial IOPs based on univariate polynomials. This is because the number of non-zero evaluations of various "quotient polynomials" arising in polynomial IOPs based on univariate polynomials grows with the degree of the polynomials regardless of how sparse they are. These non-zero quotient evaluations increase prover work. This prover cost is in addition to 0s contributing significantly to commitment time for popular univariate hashing-based polynomial commitments like FRI [BBHR18].

Twist and Shout are new examples in a growing body of work showing that the "free 0s" cost profile of elliptic-curve-based commitments opens up a rich design space enabling highly performative SNARKs. This body of work offers a strong counterpoint to the widespread view that SNARKs using elliptic curves—due to their use of 256-bit fields—are less performative than alternatives that rely only on hashing-based commitment schemes.

[23] `https://github.com/mratsim/constantine/pull/485`

Indeed, the Jolt RISC-V zkVM implementation[24] already extensively leverages this property of elliptic curves (even without yet incorporating Twist and Shout). For example, Jolt forces the prover to evaluate primitive instructions correctly by performing lookups into "subtables" of size 2^{16}. There are over a dozen subtables used across all primitive instructions, but each instruction only accesses a handful of the subtables. This is loosely analogous to how RISC-V has 32 registers but each cycle only two are read and one is written. The way Jolt handles the subtable lookups today is analogous to how Twist and Shout handle registers: For each cycle, the Jolt prover commits to a vector of binary flags indicating for each subtable whether or not it is actually accessed at that cycle. If not, the Jolt prover is free to commit to only 0s for all values "capturing" the lookup into that subtable for that cycle. This is analogous to how Twist and Shout commit to a 0 for each register that is not accessed in a given cycle (and on writes, Twist, additionally commits to a 0-increment for each register that is not accessed).

Similar techniques leveraging "free 0s" have also recently been exploited in Nebula [AS24], a collection of techniques for efficiently applying folding schemes in a zkVM context.

Going fully multilinear. The technical ethos underlying our work is that SNARKs based on the sum-check protocol and multilinear polynomials, rather than univariate polynomials (and establishing quotient relationships between them), are best for prover speed. In sum-check-based zkVMs like Jolt, there is one place today where univariate polynomials still arise: permutation-checking via Lipton's trick (see Equation (1)), which is the central component of offline memory-checking procedures and other key SNARK components. This entails treating the two vectors to be permutation-checked as the roots of a univariate polynomial and evaluating that polynomial at a random point via a grand product argument. Twist and Shout replace this source of "univariate-ness" with the sum-check protocol, eliminating grand product arguments completely.

2.9.3 Integrating Jolt (with Twist and Shout) with folding schemes

As described in the Jolt paper [AST24] and implemented in code as of this writing, Jolt is a "monolithic" SNARK, meaning that it produces a proof of execution of a fixed number of CPU cycles at once. We now discuss how to extend Jolt to prove *any* number of CPU cycles. Here, we assume that Jolt is instantiated with an additively homomorphic commitment scheme such as HyperKZG [ZSC24] or Zeromorph [KT23].

We focus on a simple approach, which we refer to as "naive folding", that composes Jolt (with Twist and Shout) with a folding scheme such as Nova [KST22]. There are alternatives to "naive folding", which are more efficient but require additional design and engineering: distill reductions in protocols such as Twist and Shout and directly construct folding schemes for virtual machine executions, offering a folding-centric route to constructing zkVMs (see NeutronNova [KS24b] and Nebula [AS24] for more details).

Details of naive folding. We restrict our attention to Nova here because using a more advanced folding scheme than Nova does not immediately provide significant benefits for naive folding. However, the description applies if we replace Nova with another similar scheme.

Recall that Nova [KS24b] is a proof system (also called an IVC scheme) that proves incremental computations of the form $y = F^{(\ell)}(x)$, where F is a possibly non-deterministic polynomial time computation, x is the initial input, and y is the output after $\ell > 0$ iterations. The proof is produced in an incremental fashion: the prover takes as input a proof of i iterations of F and updates it to produce a proof of $i+1$ iterations of F. Crucially, the prover's work, the proof size, and the verifier's work do not grow with the number of iterations executed. Internally, Nova is built using a simple primitive called a folding scheme, which reduces the task of checking two NP instances of certain size n to the task of checking a single NP instance of size n.

Suppose that Jolt is modified to prove a certain pre-determined number of CPU cycles n starting from a given register and memory state S and ending with a given register and memory state S'. Then given a program that executes for N CPU cycles (without loss of generality, assume that N is a multiple of n), we can break the execution of the program into $\ell = N/n$ "shards", and have Jolt produce ℓ separate proofs, one per shard. Each proof attests to the correct execution of the CPU for n cycles transitioning state from

[24] `https://github.com/a16z/jolt`

S to S'. When using Twist, this requires commiting to the state of memory and registers at the end of shard, which requires time linear in memory size for each shard to prove that S' is consistent with committed increments. Note that approaches such as Nebula [AS24] do not incur this $O(M)$ costs for each shard, where M is the size of memory and register state. They only incur this cost at the end of N cycles. Obtaining a similar performance profile with Twist remains open.

We then write a function F in Nova that verifies a single Jolt proof. F also needs to ensure that the register state and memory state are consistent from shard-to-shard. This can be done by making F output commitments to memory and register state, which are provided as part of Jolt's proof of a shard. Furthermore, F can then check that the starting state of a Jolt proof (for shard i where $i > 1$) is consistent with the ending state of prior shard $i - 1$ by comparing commitments provided at shard i (as part of Jolt proof for shard i) with outputs provided by F at shard $i - 1$. In Nova, the output of a shard is automatically fed as input to the next shard. We could then have the prover feed the ℓ Jolt proofs it produced to ℓ sequential steps of Nova's IVC scheme. This produces a single Nova proof attesting to the correct execution of ℓ shards of the program's execution, which in turn proves the correct execution of N CPU cycles.

The upshot is that this composition of Jolt with Nova allows it scale to prove any number of CPU cycles while benefiting from the prover-memory-efficiency properties of Nova. Note that we only need to fix a value of n and we do not need to fix a particular value for N. In other words, with this design, the prover can pause its VM and resume it at any time to execute additional cycles.

An optimization to defer polynomial evaluation arguments. Recall that a Jolt proof consists of three components: (1) multilinear polynomial commitments, (2) sum-check proofs, and (3) polynomial evaluation arguments to prove the evaluation of committed polynomials at a random point chosen over the course of the sum-check protocol. In the description above, F verifies both the sum-check proofs and polynomial evaluation proofs arising from "monolithic" Jolt. We now discuss an optimization that allow the Jolt prover to defer producing polynomial evaluation proofs. This optimization is particularly attractive because polynomial evaluation arguments are particularly expensive for the prover. For example, with HyperKZG and Zeromorph, for a polynomial of size m (i.e., a multilinear polynomial with $\log m$ variables) the prover must compute several MSMs of size m and the scalars in the MSMs are random. Furthermore, with Shout and Twist, the committed polynomials are slightly superlinear in size. For example, they are of size $m = T \cdot K^{1/d}$, where T is the number of CPU cycles proven in a shard, K is the size of memory, and d is a parameter. So, it is important to avoid proving evaluations of those polynomials for every shard.

Instead of producing a polynomial evaluation proof, the Jolt prover outputs an instance-witness pair (and the Jolt verifier outputs an instance) in the $\mathcal{R}_{\mathsf{polyeval}}$ relation [KS24a, Definition 21]. In a nutshell, an instance in this relation is of the form $u = (C, x, y)$ and a witness is a polynomial $w = P$. A witness is satisfying if P is a multilinear polynomial, C is a commitment to P, and that $P(x) = y$.

These instances can be folded using a small variant of HyperNova's multi-folding scheme, which we now sketch. The "running" instance is a single instance in the $\mathcal{R}_{\mathsf{polyeval}}$ relation and the "incoming" instances (which are produced via a Jolt proof) are rerandomized so that they all share the same evaluation point. This is analogous to how HyperNova rerandomizes running instances in its multi-folding scheme for CCS by running the sum-check protocol. Once all the instances share the same evaluation point, all instances can be folded into a single instance with a simple random linear combination.

With this folding scheme in place, F would simply fold instance-witness pairs in $\mathcal{R}_{\mathsf{polyeval}}$ (present in a Jolt proof) into a running instance. This requires representing HyperNova's folding scheme verifier in F, which involves verifying a sum-check proof and taking a weighted sum of k commitments, where k is the number of instance-witness pairs folded per Jolt proof. The prover does incur $O(m) = O(T \cdot K^{1/d})$ field operations in the folding scheme. This is tolerable in our setting as long as we set d sufficiently large to ensure that the cost contribution from this component is smaller than the cost to generate a Jolt proof of a shard. For example, with $K = 2^{30}$ memory cells (representing several GBs of memory), we can set $d = 4$ or $d = 8$.

With this optimization, Jolt+Nova's IVC proof includes a Nova proof and a single instance-witness pair (u, w) in $\mathcal{R}_{\mathsf{polyeval}}$. In addition to verifying Nova's IVC proof, the Jolt+Nova verifier will have to verify the validity of that instance-witness pair. Alternatively, the prover can replace the instance-witness pair (u, w)

with (u, π), where π is a polynomial evaluation argument proving the knowledge of a witness satisfying u and the verifier would validate π in addition to validating a Nova proof.

3 Technical preliminaries

3.1 Prover costs in elliptic-curve-based SNARKs

Cost of field operations. On modern CPUs, multiplying two elements of a 256-bit prime field using Montgomery multiplication costs between 40 and 80 CPU cycles. Field additions for such fields are an order of magnitude cheaper.

Elliptic curve commitments. In order to understand the advantages of the method of increments, it is necessary to explain the prover cost profile in SNARKs that use elliptic curve commitments. Three points are paramount: (1) committing to 0s is "free" (2) commitment to "small" values is faster than committing to "big values", and (3) under appropriate conditions, we can "ignore" the cost of computing evaluation proofs. Details follow.

Let $\ell = \log(n)$ and p be an ℓ-variate multilinear polynomial over a field $\mathbb{F}$. Let $\mathbb{G}$ be a cryptographic group with scalar field equal to $\mathbb{F}$.

With elliptic-curve-based polynomial commitment schemes, the commitment to an ℓ-variate multilinear polynomial p is simply a multi-exponentiation of size $n = 2^\ell$, namely:

$$\prod_{x_i \in \{0,1\}^\ell} g_i^{y_i}, \tag{13}$$

where $y_i = p(x_i)$ is the evaluation p at input $x_i \in \{0,1\}^\ell$. (In this paper, we treat $\mathbb{G}$ as a multiplicative group, so the product in Equation (13) refers to the group operation.) Here, each $g_i \in \mathbb{G}$ is an element of the *commitment key*, a publicly known vector of group elements of length $n = 2^\ell$ that is needed to produce commitments.

Equation (13) is also referred to as a *Pedersen vector commitment*, with the vector at hand being the exponents in Equation (13), i.e., the vector y of evaluations of p as its input ranges over $\{0,1\}^\ell$. Accordingly, we will often refer to the procedure of committing to p as "committing to $n = 2^\ell$ values", those values being the entries of y.

Commitment costs. The "smaller" the y_i values, the faster it is to compute the commitment in Equation (13). Specifically, using Pippenger's bucketing algorithm, for any desired $B \ll n$, roughly one group operation is incurred for each $y_i \in \{0, 1, \ldots, B\}$, two group operations are incurred for each $y_i \in \{B+1, \ldots, B^2\}$, and in general c group operations are required for each $y_i \in \{B^{c-1}+1, \ldots, B^d\}$. In other words, committing to a $c\log(n)$-bit value y_i requires about c group operations.

Accordingly, in order to understand the cost of committing to a vector, it is not enough to analyze the *length* of the committed vector, one must also analyze how big or small the vector's values are.

Notice in particular that if $y_i = p(x_i) = 0$, then $g_i^{y_i} = 1$, and hence the multi-exponentiation in Equation (13) is unaffected by the i'th term. This means that committing to 0s is free for the prover, in the sense that any 0-values that are committed do not alter the commitment whatsoever.

We refer to the number of non-zero entries of y as the *sparsity* of p and of y. The above means that the time to commit to p and y depend only on the sparsity, and not the ambient dimension $n = 2^\ell$. The ambient dimension of y does factor into other prover costs, discussed shortly, but not into the time required to compute the commitment.

Another important special case is an exponent $y_i = 1$. The total contribution to the commitment of all such exponents can be computed with *exactly* one group operation per entry i with $y_i = 1$, as this contribution is:

$$\prod_{i:\ y_i = 1} g_i.$$

Pippenger's bucketing algorithm achieves an amortized cost of *very close to* one group operation per committed value in $\{2, \ldots, B\}$, but not exactly one. The smaller B is relative to the length n of the committed vector y (or more precisely, the number of entries of y outside of $\{0, 1\}$), the closer the amortized cost of commitment is to one group operation per value y_i in $\{2, \ldots, B\}$.

Not only is it fast to commit to small positive values via Equation (13), it is also fast to commit to "small negative values", i.e., values in $\{-B^d, \ldots, -1\}$. To do this, one can in preprocessing invert each element of the original commitment key, so that the commitment key becomes $(g_1, \ldots, g_n, h_1, \ldots, h_n)$, where $h_i = g_i^{-1}$. Then committing to a negative value y_i can be done by committing to the positive value $-y_i$ using group element h_i in place of g_i. The Twist prover will need this in order to commit to "increments" quickly, since increments can be negative.

The cost of computing evaluation proofs. While *commitment time* depends only on the sparsity of p when using elliptic-curve-based polynomial commitment schemes, committed 0s are not literally free. They do affect the time required to compute evaluation proofs.

Specifically, if the committed vector y has length n, then evaluation proofs for multilinear polynomial commitment schemes like HyperKZG and Zeromorph require the prover to compute commitments to (i.e., multi-exponentiations over) a vector v of length n with *random* entries (in fact, several such vectors). If y is sparse and only contains small values, committing to v as required in the evaluation proof is potentially orders of magnitude more expensive than merely committing to y. This is because v is *not* sparse, nor are its entries small.

Furthermore, if using HyperKZG or Zeromorph, committed 0s also affect the size of the "powers-of-tau" SRS [KZG10] used by these commitment schemes. Specifically, to commit to an ℓ-variate multilinear polynomial with these schemes, one needs an SRS of size 2^ℓ regardless of the sparsity of the committed polynomial.

Fortunately, both of the above issues (large evaluation proof computation time, and large SRS) can be addressed by the standard amortization technique described next.

3.1.1 Amortizing the cost of computing evaluation proofs via homomorphism

Elliptic curve commitment schemes automatically come with extremely powerful amortization properties for evaluation proofs that render this concern moot, at least so long as n is not *too* much bigger than m. For example, let $\ell = \ell_1 + \ell_2$. Rather than committing to p directly, instead commit to 2^{ℓ_1} different ℓ_2-variate multilinear polynomials, where for each $z \in \{0, 1\}^{\ell_1}$, the z'th committed polynomial is $g_z(x) = p(z, x)$.

Suppose the verifier requests the evaluation $p(r, r')$ for $r \in \mathbb{F}^{\ell_1}$ and $r' \in \mathbb{F}^{\ell_2}$. The verifier on its own can, via homomorphism, compute a commitment to the ℓ_2-variate polynomial $g(x) = p(r, x)$, and the prover merely needs to provide an evaluation proof for $g(r_2)$. Computing this evaluation proof is roughly 2^{ℓ_1} times cheaper than computing an evaluation proof if p had been committed directly.

The downside of this technique is that the commitment size grows by a factor of 2^{ℓ_1}, and the verifier has to perform 2^{ℓ_1} group exponentiations[25] to homomorphically compute the commitment to $g(x) = p(r, x)$. In many contexts, this is an acceptable increase in verifier costs even for 2^{ℓ_1} in the dozens or hundreds. For example, HyperKZG and Zeromorph evaluation proofs require the verifier to do a couple of dozen group exponentiations regardless, so having the verifier do an extra couple of dozen exponentiations at most doubles verifier costs. And SNARK verifiers do more than merely process commitments, so doubling the verifier's costs to process commitments may mean less than a doubling of total verifier costs.

Even more powerful amortization via folding. A related reason that the cost of polynomial evaluation proofs can often be ignored is the use of folding schemes. Folding schemes are closely related to the above amortization procedure, except that by invoking recursion, they avoid the 2^{ℓ_1}-factor increase in verifier costs.

[25] More precisely, a multi-exponentiation of length 2^{ℓ_1}.

Here is a sketch what folding schemes can accomplish in the context of zkVMs. Say a prover wants to prove it ran a RISC-V program correctly for C cycles. To bound the prover's memory usage, one can split the computation up into, say, C/S shards each consisting of S cycles. One can apply a zkVM to prove correct execution of each shard separately, and then use folding techniques to recursively aggregate the proofs. Specifically, represent the zkVM verifier (*minus* verification of polynomial evaluation proofs, as such claims can be "accumulated" similar to the above example, rather than verified directly) via a constraint system like R1CS. Then proving that one knows a valid proof for each of the C/S shards is equivalent to proving that one knows satisfying assignments for C/S instances of the constraint system, one per shard. One can apply an efficient folding procedure such as Nova [KST22] to obtain a single "folded" instance of the constraint system such that knowledge of a satisfying assignment to the folded instance is equivalent to knowledge of a satisfying assignment of all C/S original instances. Section 2.9.3 provides more details.

The key point for our purposes is that in this setting, only a single polynomial evaluation needs to be provided, for a polynomial whose size is proportional to the size of a *single* shard.[26] That is, by breaking the CPU's execution into C/S shards and applying folding, one obtains a roughly (C/S)-fold reduction in the cost of computing an evaluation proof, compared to if the entire C-cycle CPU execution was proved in a monolithic fashion.

For the reasons above, it is often reasonable to ignore the computation of polynomial evaluation proofs when analyzing SNARK prover costs. Henceforth in this work, we do this.

Comparison to sparse polynomial commitment schemes. Readers may wonder about the relationship between the discussion above and the notion of *sparse polynomial commitment schemes* such as Spark [Set20]. These are polynomial commitment schemes that enable the prover to efficiently commit to an ℓ-variate multilinear polynomial p in time proportional to the number T of inputs $x \in \{0,1\}^\ell$ such that $p(x) \neq 0$. We refer to T as the *sparsity* of p. Haven't we just asserted that a simple Pedersen vector commitment achieves exactly this property, and also that the cost of evaluation proofs can be ignored due to amortization techniques? Doesn't that mean that any elliptic-curve based polynomial commitment scheme immediately gives a sparse polynomial commitment scheme?

The answer is that sparse polynomial commitment schemes like Spark are generally targeted at settings where $n = 2^\ell$ is so much larger than the sparsity T that the cost of computing evaluation proofs *cannot* be ignored, even given the excellent amortization properties described above.

For example, the situation where $n = T^2$ naturally arises in SNARKs for R1CS or circuit satisfiability, and this is the setting that motivated the development of Spark (and also arises in our study of SNARKs for non-uniform constraint systems, see Section 9). In the zkVM context, n turns out to equal $K \cdot T$ where K is the size of the memory being checked. There are two potential differences between our setting and Spark's:

- We are interested in settings where T is in the millions or billions or larger (in a zkVM context, T corresponds roughly to the number of cycles in the VM's execution), but where K may (or may not) be much smaller. For example, in the setting of RISC-V registers, K is just 32. This is a setting where the cost of evaluation proofs can definitely be ignored due to amortization, but this this may not be the case if $n \geq T^2$ as considered in Spark.
- Spark is a polynomial commitment scheme for *arbitrary* sparse polynomials. Whereas, Twist and Shout do not need to commit to arbitrary sparse polynomials, but rather to polynomials whose evaluations can be viewed as a $K \times T$ matrix that have *exactly one non-zero entry per row*. Nonetheless, we show in Section 9.3.1 that Shout indirectly yields a general sparse polynomial commitment scheme, a major improvement over Spark that we call Spark++.

[26] This ability to avoid having the prover produce a polynomial evaluation proof for each shard, and avoid recursively proving that it verified such a proof, is a key benefit of folding schemes compared to the "brute force recursion" [Tea22, COS20] that is used today by all zkVMs that avoid elliptic curves.

3.1.2 An analytic cost model: relating group operations to field operations

When committing to a multilinear polynomial p defined over field $\mathbb{F}$ using an elliptic curve group $\mathbb{G}$, $\mathbb{F}$ will always be the *scalar field* of $\mathbb{G}$, which is why we can interpret evaluations of p as *exponents* of group elements in Equation (13). Elements of $\mathbb{G}$ correspond to pairs of elements (x, y) in a *different* field $\mathbb{B}$ called the *base field* of $\mathbb{G}$. And group operations are naturally defined in terms of base field operations. Specifically, multiplying two group elements together typically requires about 6 base-field multiplications [GW20a].

Despite the fact that $\mathbb{B}$ and $\mathbb{F}$ are different fields, often operations in $\mathbb{B}$ and $\mathbb{F}$ have the same cost. For example, if the elliptic curve is BN254, both field implementations involve 256-bit Montgomery arithmetic [Mon85]. This allows us to directly relate the cost of group operations to $\mathbb{F}$-operations: a single operation in $\mathbb{G}$ has roughly the same cost as 6 multiplications in $\mathbb{F}$. We ignore the cost of field additions, as in 256-bit fields they are an order of magnitude cheaper than field multiplications.

Some pairing-friendly curves, like BLS12-381, have a larger base field than scalar field (e.g., the base field is 381 bits for BLS12-381 while the scalar field is only 256 bits). For these fields and curves, the method of increments is still attractive relative to memory-checking techniques. This is because the method of increments has substantially lower commitment costs compared to memory-checking techniques (by commitment costs, we mean the number of group operations required to commit to data). And the bigger the base field of the elliptic curve group, the more expensive those group operations are.

Summary of our analytic cost model. On CPUs, we consider a 256-bit field multiplication to be 40-80 times more expensive than a native multiplication of two 64-bit data types [MvOV01, Chapter 14]. We consider a group operation (often called a group addition, using additive-group terminology) to be roughly 6 times as expensive as a field operation [GW20a]. We treat small committed values as costing exactly one group operation (see Section 2.1, especially Footnote 7, and Section 3.1 for discussion). We treat random values as costing about 11 group operations to commit to (as this is an optimistic bound on the runtime of Pippenger's algorithm at realistic input sizes [EHB22]). We ignore field additions, as well as multiplications by small constants such as 2, since $2x$ can be computed with a single addition: $2x = x + x$.

A scalar multiplication (aka group exponentiation) by an arbitrary field element is equivalent in cost to roughly 400 group operations via the standard double-and-add algorithm, and a pairing evaluation is equivalent in cost to roughly 10 scalar multiplications.

Caveats. Of course, any such cost model will only approximately match actual system performance. Different cryptographic groups have different base field sizes, and real hardware is complicated (e.g., some operations are memory-bound rather than compute bound). However, the above approximate costs are carefully informed by examination of how field operations and group operations are actually computed, as well as by microbenchmarks.

Any inaccuracies in our cost model generally understate our improvements over prior works. For example, we treat all small committed values as costing exactly one group operation, when in fact committing to a 1-value truly requires one group operation, and committing to larger-but-still-small values is somewhat more expensive. Since our protocols commit to more 1-values and fewer larger-but-still-small values than prior work, this cost model understates our speedups.

3.2 Multilinear extensions

Our treatment of multilinear extension polynomials, polynomial commitment schemes, and SNARKs is standard and taken verbatim from Setty and Thaler [STW23].

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

A basic fact about multilinear polynomials is the following.

Fact 3.1. *Let $p\colon \mathbb{F}^n \to \mathbb{F}$ be any multilinear polynomial. Then for any $c \in \mathbb{F}$ and any $x' \in \{0,1\}^{n-1}$,*

$$p(c, x') = (1 - c) \cdot p(0, x') + c \cdot p(1, x'). \tag{14}$$

Proof. The right and left hand sides of Equation (14) are both multilinear polynomials and are easily seen to evaluate to the same value whenever $(c, x') \in \{0,1\} \times \{0,1\}^{n-1}$. Since $\{0,1\}^n$ is an interpolating set for multilinear polynomials, the left hand side and right hand side of Equation (14) must be equal for all $c \in \mathbb{F}$ (and any $x' \in \mathbb{F}^{n-1}$ as well). □

The right hand side of Equation (14) can be computed with a single field multiplication, as it equals $p(0, x') + c \cdot (p(1, x') - p(0, x'))$.

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

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

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

$$\widetilde{\mathsf{eq}}(x, e) = \prod_{i=1}^{s} \left(x_i e_i + (1 - x_i)(1 - e_i)\right). \tag{15}$$

Indeed, one can easily check that the right hand side of Equation (15) is a multilinear polynomial, and that if evaluated at any input $(x, e) \in \{0,1\}^s \times \{0,1\}^s$, it outputs 1 if $x = e$ and 0 otherwise. Hence, the right hand side of Equation (15) is the unique multilinear polynomial extending eq. Equation (15) implies that $\widetilde{\mathsf{eq}}(r_1, r_2)$ can be evaluated at any point $(r_1, r_2) \in \mathbb{F}^s \times \mathbb{F}^s$ in $O(s)$ time.[27]

Another multilinear extension that we will make use of is the MLE of the *less-than* function. Given any vector $j = (j_1, \ldots, j_{\log T}) \in \{0,1\}^{\log T}$, define

$$\mathrm{int}(j) = \sum_{i=1}^{\log T} 2^{i-1} \cdot j_i.$$

That is, $\mathrm{int}(j)$ is the integer whose binary representation is j. Then define

$$\mathsf{LT}(j', j) = \begin{cases} 1 & \text{if } \mathrm{int}(j') < \mathrm{int}(j) \\ 0 & \text{otherwise.} \end{cases}$$

It was shown in [STW24, Appendix G, see Equation 58 on Page 56] that $\widetilde{\mathsf{LT}}$ can be evaluated at any input $(r', r) \in \mathbb{F}^{\log T} \times \mathbb{F}^{\log T}$ with $O(\log T)$ field operations. See also Appendix B.

Multilinear extensions of vectors. Given a vector $u \in \mathbb{F}^m$, we will often refer to the *multilinear extension of* u and denote this multilinear polynomial by $\widetilde{u}$. $\widetilde{u}$ is obtained by viewing u as a function mapping $\{0,1\}^{\log m} \to \mathbb{F}$ in the natural way[28]: the function interprets its $(\log m)$-bit input $(i_0, \ldots, i_{\log m - 1})$ as the binary representation of an integer i between 0 and $m - 1$, and outputs u_i. $\widetilde{u}$ is defined to be the multilinear extension of this function.

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

[28] All logarithms in this paper are to base 2.

Multilinear Lagrange interpolation. An explicit expression for the MLE of a vector $u \in \mathbb{F}^m$ is as follows:

$$\widetilde{u}(x) = \sum_{y \in \{0,1\}^{\log m}} u_y \cdot \widetilde{\mathsf{eq}}(y, x).$$

Indeed, it is easily checked that the right hand side of the above equality is multilinear in x, and for any $x \in \{0,1\}^{\log m}$ the right hand side equals u_x. Hence, the right hand side is the unique multilinear extension of u.

For each $y \in \{0,1\}^{\log m}$, the multilinear polynomial

$$x \mapsto \widetilde{\mathsf{eq}}(y, x) = \prod_{i=1}^{\log m} (y_i x_i + (1 - y_i)(1 - x_i))$$

is called the *yth multilinear Lagrange basis polynomial.*

The following lemma is well-known [VSBW13].

Lemma 1 (Vu et al. [VSBW13])**.** *Given as input a point $r \in \mathbb{F}^{\log m}$, it is possible to evaluate all Lagrange basis polynomials at r using only m field multiplications. Given a vector $u \in \mathbb{F}^m$, it is possible to compute $\widetilde{u}(r)$ with $2m$ field multiplications.*

Proof. For $i = 1, \ldots, m$, define A_i to be the array of length 2^i, with entries indexed by $x \in \{0,1\}^i$, such that $A_i[x] = \widetilde{\mathsf{eq}}(x, r_1, \ldots, r_i)$. Then

$$A_{i+1}[x, 1] = r_i \cdot A_i[x]$$

and

$$A_{i+1}[x, 0] = (1 - r_i) \cdot A_i[x] = A_i[x] - A_{i+1}[x, 1].$$

$A_{\log m}$ contains the desired evaluations. The total cost to compute $A_{\log m}$ is $1 + 2 + \cdots + m/2 = m$ multiplications. See [Rot24] for an alternative algorithm.

Given a vector $u \in \mathbb{F}^m$, $\widetilde{u}(r)$ is simply the inner product of u with $A_{\log m}$. This inner product can be computed with m field multiplications (on top of the m required to compute $A_{\log m}$). □

3.3 The sum-check protocol

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

$$H \coloneqq \sum_{b \in \{0,1\}^\ell} g(b). \tag{16}$$

To compute H unaided, the verifier would have to evaluate g at all 2^ℓ points in $\{0,1\}^\ell$ and sum the results. The sum-check protocol allows the verifier to offload this hard work to the prover. It consists of ℓ rounds, one per variable of g. In round i, the prover sends a message consisting of d_i field elements, where d_i is the degree of g in its i'th variable, and the verifier responds with a single (randomly chosen) field element. If the prover is honest, this polynomial (in the single variable X_i) is

$$\sum_{(b_{i+1}, \ldots, b_{\ell-1}) \in \{0,1\}^{\ell - i}} g(r_0, \ldots, r_{i-1}, X_i, b_{i+1}, \ldots, b_{\ell-1}). \tag{17}$$

Here, we are indexing both the rounds of the sum-check protocol and the variables of g starting from zero (i.e., indexing them by $\{0, 1, \ldots, \ell - 1\}$), and $r_0, \ldots, r_{i-1}$ are random field elements chosen by the verifier across rounds $0, \ldots, i - 1$ of the protocol.

The protocol has perfect completeness, and soundness error $\left(\sum_{i=1}^{\ell} d_i\right) / |\mathbb{F}|$. The verifier's runtime is $O\left(\sum_{i=1}^{\ell} d_i\right)$, plus the time required to evaluate g at a single point $r \in \mathbb{F}^\ell$. In the typical case that $d_i = O(1)$

for each round i, this means the total verifier time is $O(\ell)$, plus the time required to evaluate g at a single point $r \in \mathbb{F}^\ell$. This is exponentially faster than the 2^ℓ time that would generally be required for the verifier to compute H. See [AB09, Chapter 8] or [Tha22, §4.1] for details.

Review of linear-time sum-check proving. Suppose the sum-check protocol is applied to compute $\sum_{x\in\{0,1\}^n} \widetilde{\mathsf{eq}}(r', x) \cdot g(x)$ for some fixed $r' \in \mathbb{F}^n$, where

$$g(x) = \prod_{i=1}^{\ell} p_i(x), \tag{18}$$

and where each p_i is multilinear. Let $N = 2^n$. The standard linear-time sum-check prover algorithm [CTY11, Tha13] has the prover begin by storing arrays $A_1, \ldots, A_\ell$ each of size N with entries indexed by $\{0,1\}^n$, where $A_i(x)$ stores $p_i(x)$. The prover also stores an array B containing all evaluations $\widetilde{\mathsf{eq}}(r, x)$ as x ranges over $\{0,1\}^n$. It is known how to build this array with N field multiplications in total.

In round m of the sum-check protocol, for each $c' \in \{0, 1, 2, \ldots, \ell + 1\}$, the must compute

$$s_m(c') = \sum_{y\in\{0,1\}^{n-m}} \mathsf{eq}(r', r_1, \ldots, r_{m-1}, c', y) g(r_1, \ldots, r_{m-1}, c, y), \tag{19}$$

where s_m is the degree-$(\ell + 1)$ univariate polynomial sent by the sum-check prover in round ℓ.

Accordingly, the sum-check prover at the start of each round m makes sure that A_i stores $p_i(r_1, \ldots, r_{m-1}, y)$ as y ranges over $\{0,1\}^{n-(m-1)}$. By Fact 3.1, the following procedure suffices to ensure this. At the end of each round m, when the verifier selects that round's random challenge $r_m \in \mathbb{F}$, the sum-check prover updates each entry $A_i(y)$ for $y \in \{0,1\}^{n-m}$ to:

$$\begin{aligned} A_i(y) \leftarrow p_i(r_1, \ldots, r_m, y) &= (1 - r_m) \cdot p_i(r_1, \ldots, r_{m-1}, 0, y) + r_m \cdot p_i(r_1, \ldots, r_{m-1}, 1, y) \\ &= A_i(0, y) + r_m \cdot (A_i(y, 1) - A_i(y, 0)) \,. \end{aligned} \tag{20}$$

We refer to this procedure as the prover *binding* the m'th variable to r_m. A similar update is performed on the array B of evaluations $\widetilde{\mathsf{eq}}(r', r_1, \ldots, r_m, y)$.

Given the A_i arrays at each round m, the prover can evaluate $g(r_1, \ldots, r_{m-1}, c', y)$ for the $\ell + 2$ relevant points c' and all $y \in \{0,1\}^{n-m}$ with $\ell - 1$ field multiplications per point. Multiplying $g(r_1, \ldots, r_{m-1}, c', y)$ by the value $\widetilde{\mathsf{eq}}(r', r_1, \ldots, r_{m-1}, c', y)$ (which is easily derivable from the array B without additional general field multiplications) costs an additional field multiplication per point, and this counts all field operations needed to compute the quantities appearing in Equation (19).

Hence, the prover's total work across all rounds of the sum-check protocol is $\ell \cdot N$ field multiplications across all binding operations for the arrays $A_1, \ldots, A_\ell$, plus $2N$ operations to compute and then bind the B array, plus $\ell \cdot (\ell + 2) \cdot N$ field multiplications across all rounds to compute the prover's messages given the arrays $A_1, \ldots, A_\ell$. That is

$$(\ell^2 + 3\ell + 2)N$$

field multiplications in total.

Recent optimizations have brought this cost down. Dao and Thaler [DT24] show how to effectively eliminate the $2N$ operations required to build and bind the B array of $\widetilde{\mathsf{eq}}$ evaluations. And it's well-known that the prover need not consider the evaluation point $c' = 1$ in each round because when the prover is honest, $s_m(1) = s_{m-1}(r_{m-1}) - s_m(0)$. On top of this, Gruen [Gru24] effectively showed that the evaluation point $\ell + 1$ can be omitted as well (see Section 3.6.1 for details). Combining these optimizations, the number of field multiplications performed by the prover falls to roughly

$$(\ell^2 + \ell) \cdot N.$$

3.4 SNARKs and commitment schemes

SNARKs We adapt the definition provided in [KST22].

Definition 3.1. *Consider a relation $\mathcal{R}$ over public parameters, structure, instance, and witness tuples. A non-interactive argument of knowledge for $\mathcal{R}$ consists of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$, denoting the generator, the prover, the verifier and the encoder respectively with the following interface.*

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

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

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

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

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

A non-interactive argument of knowledge is succinct if the verifier's time to check the proof π and the size of the proof π are at most polylogarithmic in the size of the statement proven.

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

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

Definition 3.2. *A tuple of four protocols* $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ *is an extractable polynomial commitment scheme for multilinear polynomials over a finite field $\mathbb{F}$ if the following conditions hold.*

- ***Completeness.*** *For any ℓ-variate multilinear polynomial $g \in \mathbb{F}[\ell]$,*

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

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

$$\Pr\left\{\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \ell);\ (\mathcal{C}, g_0, g_1) = \mathcal{A}(\mathsf{pp}); \\ b_0 \leftarrow \mathit{Open}(\mathsf{pp}, \mathcal{C}, g_0);\ b_1 \leftarrow \mathit{Open}(\mathsf{pp}, \mathcal{C}, g_1): \\ b_0 = b_1 \neq 0 \wedge g_0 \neq g_1 \end{array}\right\} \leq \mathit{negl}(\lambda)$$

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

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

3.5 Polynomial IOPs and polynomial commitments

Modern SNARKs are constructed by combining a type of interactive protocol called a *polynomial IOP* [BFS20] with a cryptographic primitive called a *polynomial commitment scheme* [KZG10]. The combination yields a succinct *interactive* argument, which can then be rendered non-interactive via the Fiat-Shamir transformation [FS86b], yielding a SNARK.

Roughly, a polynomial IOP is an interactive protocol where, in one or more rounds, the prover may "send" to the verifier a very large polynomial g. Because g is so large, one does not wish for the verifier to read a complete description of g. Instead, in any efficient polynomial IOP, the verifier only "queries" g at one point (or a handful of points). This means that the only information the verifier needs about g to check that the prover is behaving honestly is one (or a few) evaluations of g.

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

Costs of some specific polynomial commitment schemes. The following elliptic-curve-based polynomial commitments are of particular interest to us. Two are HyperKZG and Zeromorph [ZSC24, KT23]. These are homomorphic commitment schemes for multilinear polynomials. The commitment consists of one group element in a pairing friendly group $\mathbb{G}_1$, and committing to an ℓ-variate multilinear polynomial p consists of applying an MSM to the vector of evaluations of p across all inputs in $\{0,1\}^\ell$. The commitment key is the powers-of-tau SRS (also used in KZG commitments [KZG10]) of size $N = 2^\ell$. Evaluation proofs have logarithmic size and verifying them requires performing two or three pairings and an MSM of logarithmic size. Computing the evaluation proof requires, for each $i = 1, \ldots, \ell$, a constant number of MSMs of length 2^i (the scalars in this MSM are random field elements, if the evaluation point is random).

Dory [Lee21] is a transparent elliptic-curve-based polynomial commitment scheme that also uses pairings. An attractive property of Dory is that its commitment key is sublinear size, consisting of only $\sqrt{N}$ elements of $\mathbb{G}_1$ and $\sqrt{N}$ elements of $\mathbb{G}_2$. Committing to an ℓ-variate multilinear polynomial entails performing roughly $\sqrt{N}$ MSMs each of length $\sqrt{N}$ (as with HyperKZG and Zeromorph, the scalars the MSMs are applied to comprise evaluations of p across all inputs in $\{0,1\}^\ell$). In addition, committing requires computing a *multi-pairing* of size $\sqrt{N}$.[29] Dory has a transparent linear-time pre-processing phase, which produces a verification key of size just $O(\log N)$. Evaluation proofs consist of $6 \log N$ target group elements and verifying them involves logarithmically many scalar multiplications in the target group.

3.6 Zero-check PIOP

Let g be an ℓ-variate polynomial of, say, constant degree in each variable. The following standard interactive proof establishes that that $g(x) = 0$ for all $x \in \{0,1\}^\ell$, while requiring the verifier to merely evaluate g at a single point in $\mathbb{F}^\ell$ (after processing a proof consisting of $O(\ell)$ field elements).

[29] A multi-pairing of size S is an expression of the form $\prod_{i=1}^{S} e(a_i, b_i)$ where each $a_i \in \mathbb{G}_1$ and $b_i \in \mathbb{G}_2$.

The verifier picks an input $r \in \mathbb{F}^\ell$ at random. The prover and verifier apply the sum-check protocol to confirm that

$$0 = \sum_{x \in \{0,1\}^\ell} \widetilde{\mathsf{eq}}(r, x) \cdot g(x). \tag{21}$$

This PIOP is perfectly complete. By a direct application of the Schwartz-Zippel lemma, the soundness error is $\ell/|\mathbb{F}|$ (plus the soundness error of the sum-check protocol itself).

3.6.1 An optimization of Gruen

Suppose the polynomial $g(x)$ in Equation (21) has degree d in each variable of x. Then $\widetilde{\mathsf{eq}}(r, x) \cdot g(x)$ has degree $d+1$ in each variable of x. As a consequence, when applying the sum-check protocol to $\widetilde{\mathsf{eq}}(r, x) \cdot g(x)$, the prover's message $s_i(X)$ in each round i is a univariate polynomial of degree $d+1$. This has implications for prover time: the prover must evaluate s_i at $d+2$ points, say $\{0, 1, \ldots, d+1\}$ (in fact, the evaluation $s_i(1)$ can be omitted and/or had "for free" as it is quickly derivable from $s_i(0)$ and s_{i-1}. See Section 3.3 for details).

Gruen [Gru24, Section 3] shows that the sum-check protocol can be modified so that in each round i the prover only has to compute a degree-d polynomial s_i', not the degree $d+1$ polynomial s_i (and s_i can then be derived from s_i', in time that depends only on d). This saves the prover the work of evaluating the relevant polynomial at one point (say, point $d+1$). (Gruen describes his optimization as having the prover send s_i' instead of s_i, but this requires modifying the standard sum-check verifier so as to process s_i' rather than s_i. We prefer to leave the standard sum-check verification procedure unchanged).

The idea is roughly to define s_i' to "leave out" the contribution of $\widetilde{\mathsf{eq}}(r, x)$ to s_i when defining s_i'. This reduces the degree of s_i by one. More precisely, since $\widetilde{\mathsf{eq}}(r, x)$ factors into a product of terms where each term depends on a single variable, s_i' leaves out the contribution of variable i to $\widetilde{\mathsf{eq}}(r, x)$. This contribution is independent of the other variables still being summed over in round i, so the prover can "add this contribution back in" (i.e., compute s_i from s_i') in time independent of the number of terms being summed.

Specifically, if $(\mathsf{r}_1, \ldots, \mathsf{r}_{i-1})$ denotes the randomness chosen by the sum-check verifier in rounds $1, \ldots, i-1$, then recall that

$$\begin{aligned} s_i(c) &= \sum_{x' \in \{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(r, \mathsf{r}_1, \ldots, \mathsf{r}_{i-1}, c, x') \cdot g(\mathsf{r}_1, \ldots, \mathsf{r}_{i-1}, c, x') \\ &= \underbrace{\left(\prod_{j=1}^{i-1} (r_j \mathsf{r}_j + (1 - r_j)(1 - \mathsf{r}_j)) \right)}_{\text{call this factor } A} \cdot \underbrace{(r_i c + (1 - r_i)(1 - c))}_{\text{call this factor } B(c)} \underbrace{\sum_{x' \in \{0,1\}^{\ell-i}} \widetilde{\mathsf{eq}}(r_{i+1}, \ldots, r_\ell, x') g(\mathsf{r}_1, \ldots, \mathsf{r}_{i-1}, c, x')}_{\text{call this factor } C(c)}. \end{aligned}$$

The modified polynomial $s_i'(X)$ for round i is

$$s_i'(c) = A \cdot C(c).$$

Crucially, for each $c \in \{0, \ldots, d+1\}$, $s_i(c) = s_i'(c) \cdot B(c)$. Effectively, this modification has removed from the prover's message $s_i(c)$ the degree-1 factor $B(c)$ that the i'th variable of x contributed to the term $\widetilde{\mathsf{eq}}(r, x)$.

The prover can compute $s_i'(c)$ for $d+1$ points via the standard linear-time sum-check proving algorithm (Section 3.3). These $d+1$ evaluations fully specify s_i' since s_i' has degree d rather than $d+1$ (note that, per a standard observation, the evaluation $s_i'(0)$ can be computed in constant time from $s_i'(1)$ and $s_i(0)$).

So once the prover has computed s_i', in time that depends only on d it can compute $s_i(c)$ at the $d+2$ points $c \in \{0, \ldots, d+1\}$, thereby fully specifying s_i.

1. $\mathcal{P}$ and $\mathcal{V}$ have agreed upon a size-K lookup table whose multilinear extension is given by $\widetilde{\mathsf{Val}}$ (which we assume to be evaluable in $O(\log K)$ time), and $\mathcal{P}$ has already committed to a multilinear polynomial $\widetilde{\mathsf{ra}}\colon \mathbb{F}^{\log K} \times \mathbb{F}^{\log T} \to \mathbb{F}$. $\mathcal{P}$ wishes to give the verifier query access to the virtual polynomial $\widetilde{\mathsf{rv}}\colon \mathbb{F}^{\log T} \to \mathbb{F}$, defined as the unique multilinear polynomial satisfying that: for all $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{rv}}(j) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j) \cdot \widetilde{\mathsf{Val}}(k)$.
2. $\mathcal{V} \to \mathcal{P}$: pick the desired evaluation point $r_{\text{cycle}} \in \mathbb{F}^{\log T}$ send it to $\mathcal{P}$.
3. $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to compute

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r_{\text{cycle}}) \cdot \widetilde{\mathsf{Val}}(k).$$

4. Let r_{address} denote the randomness chosen over the course of the sum-check protocol. To perform $\mathcal{V}'s$ check in the final round of the sum-check protocol, $\mathcal{V}$ evaluates the committed polynomial $\widetilde{\mathsf{ra}}$ respectively at the random point $(r_{\text{address}}, r_{\text{cycle}})$, and $\mathcal{V}$ evaluates $\widetilde{\mathsf{Val}}(r_{\text{address}})$ directly in $O(\log K)$ time.

Figure 5: The core Shout PIOP when $d = 1$, assuming the lookup table is MLE-structured (i.e., $\mathcal{V}$ can evaluate $\widetilde{\mathsf{Val}}$ at a random input in $O(\log K)$ time).

3.7 One-hot encodings

Fix K and let $z \in \{0, 1, \ldots K-1\} \subseteq \mathbb{F}$. The (one-dimensional) *one-hot encoding* of z is the unit vector $e_z \in \mathbb{F}^K$ whose z'th entry is 1 and all other entries of which are 0.

Next, let us assume for simplicity that $K^{1/d}$ is an integer. For $d > 1$, the d-dimensional one-hot encoding of z refers to the collection of d unit vectors, say $v_1, \ldots, v_d$, each of length $K^{1/d}$, whose tensor product $v_1 \otimes v_2 \cdots \otimes v_d$ equals e_z. In other words, if we index the K entries of e_z by $k = (k_1, \ldots, k_d) \in \{0, \ldots, K^{1/d} - 1\}^d$, then $v_1, \ldots, v_d$ are the unique unit vectors such that the k'th entry of e_z equals

$$\prod_{i=1}^{d} v_i(k_i).$$

This implies that, if z itself corresponds to $(z_1, \ldots, z_d)$ via the natural bijection between $\{0, \ldots, K-1\}$ and $\{0, \ldots, K^{1/d} - 1\}^d$, then $v_i = e_{z_i} \in \{0,1\}^{K^{1/d}}$ for $i = 1, \ldots, d$.

4 The Shout PIOP

This section gives a complete description of the Shout PIOP (both the special case when $d = 1$ and the case of general d). We defer until Section 6 an explanation of how to quickly implement the prover in this PIOP.

4.1 A special case: $d = 1$

4.1.1 Core Shout PIOP for $d = 1$

Figure 5 contains the core Shout PIOP when $d = 1$. It is identical to (the first sum-check invocation) in the Generalized-Lasso protocol [STW24].

Theorem 1. *Consider an instance of read-only memory-checking, and assume that for each $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(\cdot, j)$ is the one-hot representation of the address read at cycle j. Then the PIOP in Figure 5 has perfect completeness and soundness error $(2\log(K) + \log(T))/|\mathbb{F}|$.*

Proof. Since $\widetilde{\mathsf{ra}}(\cdot, j)$ is the one-hot representation of the address read at cycle j, $\widetilde{\mathsf{rv}}(j)$ is indeed the table

value stored at that address if and only if

$$\widetilde{\mathsf{rv}}(j) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j) \cdot \widetilde{\mathsf{Val}}(k). \tag{22}$$

Both the left hand side and right hand side of Equation (22) are multilinear polynomials in j. Hence they are equal as formal polynomials if and only if this equality holds for all $j \in \{0,1\}^{\log T}$. By the Schwartz-Zippel lemma, up to soundness error $\log(T)/|\mathbb{F}|$, in order to check that the left hand side and right hand side are the same multilinear polynomial it suffices for $\mathcal{V}$ to pick r_{cycle} at random from $\mathbb{F}^{\log T}$ and confirm that

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r_{\text{cycle}}) \cdot \widetilde{\mathsf{Val}}(k). \tag{23}$$

Figure 5 applies the sum-check protocol to confirm this. The claim now follows from the completeness and soundness properties of the sum-check protocol (Section 3.3). □

According to our formulation of the memory-checking problem (Section 2.2), one should separately check that for each $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(\cdot, j)$ is indeed the one-hot representation of some address $\mathsf{raf}(j) \in \{0, 1, \ldots, K-1\}$ that is read at cycle j (and query access to the multilinear polynomial $\widetilde{\mathsf{raf}}$ should be granted to the verifier). A PIOP for establishing this is given in Section 4.1.2 below.

We describe in Section 6.1 how to quickly implement the prover in core Shout PIOP when $d = 1$.

4.1.2 Checking correctness of one-hot encodings

Suppose the prover has committed to (the multilinear extension of) a vector $\mathsf{ra} \in \left(\mathbb{F}^K\right)^T$ and is also prepared to grant the verifier query access to a $\log(T)$-variate multilinear virtual polynomial $\widetilde{\mathsf{raf}}$. The prover claims that for each $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{raf}}(j) \in \{0, 1, \ldots, K-1\}$ and the vector $\widetilde{\mathsf{ra}}(\cdot, j)$ as $\cdot$ ranges over $\{0,1\}^{\log K}$, equals the one-hot encoding of $\widetilde{\mathsf{raf}}(j)$. We describe a PIOP to check these claims. In other words, the PIOP allows the verifier to:

- Confirm that $\widetilde{\mathsf{ra}}(k, j) \in \{0, 1\}$ for all $k \in \{0,1\}^{\log K}$.
- Confirm that $\widetilde{\mathsf{ra}}(k, j)$ equals 1 for exactly one register $k \in \{0,1\}^{\log K}$.
- Obtain query access to $\widetilde{\mathsf{raf}}(j)$, where the $k \in \{0,1\}^{\log K}$ from the previous bullet is the binary representation of $\widetilde{\mathsf{raf}}(j)$.

Checking the first bullet. To check the first bullet point, simply apply the zero-check PIOP to confirm that for all $(k, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$,

$$\widetilde{\mathsf{ra}}(k, j)^2 - \widetilde{\mathsf{ra}}(k, j) = 0. \tag{24}$$

We will show later (Section 6.3) that the prover in this zero-check PIOP performs only $O(K) + 2T$ field multiplications.

Checking the second bullet. The second bullet point is equivalent to the following constraint holding for all $j \in \{0,1\}^{\log T}$:

$$1 = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j). \tag{25}$$

Since the left hand side and right hand side of Equation (25) are both multilinear polynomials in j (in fact, the left hand side has degree 0), satisfaction of this constraint system implies that $\widetilde{a}$ equals the right hand side

of Equation (27) as formal polynomials (and vice versa). Hence, the entire constraint system can be checked (with soundness error $\log(T)/|\mathbb{F}|$) by having the verifier pick a random $r' \in \mathbb{F}^{\log T}$ and confirming that

$$1 = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r'). \tag{26}$$

Since $\widetilde{b}$ is multilinear, it is easy to see that, so long as the field characteristic is greater than 2, this sum is in fact equal to

$$K \cdot \widetilde{\mathsf{ra}}(2^{-1}, \ldots, 2^{-1}, r'),$$

(see for example [AW09, End of Section 1.4] for reference). And so Equation (26) can be confirmed to hold by the verifier with a single evaluation query to $\widetilde{\mathsf{ra}}$, at point $(2^{-1} \ldots, 2^{-1}, r')$.

This technique does not work over binary fields (where 2^{-1} is undefined). In that case, one can simply apply the sum-check protocol directly to compute Equation (26). This has the additional benefit that it lowers the number of evaluation queries to $\widetilde{\mathsf{ra}}$ performed by the verifier (i.e., by running this sum-check in parallel with the one used to check the third bullet below, we can ensure that both sum-checks wind up evaluating $\widetilde{\mathsf{ra}}$ at the same point).

Checking the third bullet. Given the first two bullet points hold, the third bullet point is equivalent to to the following constraint holding for all $j \in \{0,1\}^{\log T}$:

$$\widetilde{a}(j) = \sum_{k \in \{0,1\}^{\log K}} \left(\sum_{i=0}^{\log(K)-1} 2^i k_i \right) \cdot \widetilde{\mathsf{ra}}(k, j). \tag{27}$$

Since the left hand side and right hand side of Equation (27) are both multilinear polynomials in j, satisfaction of this constraint system implies that $\widetilde{a}$ equals the right hand side of Equation (27) as formal polynomials (and vice versa). Hence, the entire constraint system can be checked (with soundness error $\log(T)/|\mathbb{F}|$) by having the verifier pick a random $r' \in \mathbb{F}^{\log T}$ and confirming that

$$\widetilde{\mathsf{raf}}(r') = \sum_{k \in \{0,1\}^{\log K}} \left(\sum_{i=0}^{\log(K)-1} 2^i k_i \right) \cdot \widetilde{\mathsf{ra}}(k, r').$$

The verifier can obtain the evaluation $\widetilde{\mathsf{raf}}(r)$ from the commitment to $\widetilde{\mathsf{raf}}$. To compute

$$\sum_{k \in \{0,1\}^{\log K}} \left(\sum_{i=0}^{\log(K)-1} 2^i k_i \right) \cdot \widetilde{\mathsf{ra}}(k, r'),$$

the prover and verifier can apply the sum-check protocol, at the end of which the verifier needs to evaluate $\widetilde{\mathsf{ra}}(r, r')$ for a random point $(r, r') \in \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$. This evaluation can be obtained by the verifier from the commitment to $\widetilde{\mathsf{ra}}$.

Figure 6 provides the entire PIOP. The following theorem is immediate from completeness and soundness of the sum-check protocol and the discussion above.

Theorem 2. *Figure 6 satisfies perfect completeness and has soundness error* $(6\log(K) + 4\log(T))/|\mathbb{F}|$. *That is, if* $\mathsf{ra}(j)$ *is not the one-hot encoding of some value* $\mathsf{raf}(j) \in \{0, 1, \ldots, K-1\}$ *for all* $j \in \{0,1\}^{\log T}$, *or if the claim that* $\widetilde{\mathsf{raf}}(r') = y$ *is false, the verifier will accept with probability at most* $(2\log(K) + \log(T))/|\mathbb{F}|$.

Proof. First, suppose that for some $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(\cdot, j)$ is not the correct one-hot-representation of some value in $\{0, 1, \ldots, K-1\}$. Then either $\widetilde{\mathsf{ra}}(k, j) \notin \{0,1\}$ for some $k \in \{0,1\}^{\log K}$, or $\sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j) \neq 1$. We claim this implies that the verifier in Figure 6 rejects with sufficiently high probability.

Observe that the following polynomial $g(r, r')$ is multilinear:

$$g(r, r') = \sum_{k \in \{0,1\}^{\log K},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, k)\widetilde{\mathsf{eq}}(r', j)\left(\widetilde{\mathsf{ra}}(k, j)^2 - \widetilde{\mathsf{ra}}(k, j)\right).$$

Furthermore, if there is any $(k, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$ such that $\widetilde{\mathsf{ra}}(k, j) \notin \{0, 1\}$, g is not the identically-zero polynomial. Hence, by the Schwartz-Zippel lemma, with probability at least $1 - (\log(K) + \log(T))/|\mathbb{F}|$ over the random choice of r, r', Equation (29) fails to hold. In this event, the soundness guarantee of the sum-check protocol ensures that the verifier rejects within the Booleanity-checking sum-check with probability at least $1 - 3(\log(K) + \log(T))/|\mathbb{F}|$.

Let $h(j) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j)$. h is multilinear in j, and if there is any $j \in \{0,1\}^{\log T}$ for which $h(j) \neq 1$, then h is not the identically-1 polynomial. In this case, by the Schwartz-Zippel lemma, with probability at least $1 - \log(T)/|\mathbb{F}|$ over the random choice of r_{cycle}, $1 = h(r_{\text{cycle}})$ will fail to hold. In this event, the soundness guarantee of the sum-check protocol ensures that the verifier will reject in the Hamming weight 1 check will probability at least $1 - \log(K)/|\mathbb{F}|$. This completes the proof of the claim.

Finally, suppose that $\widetilde{\mathsf{raf}}(r_{\text{cycle}}) \neq y$. We claim that $\widetilde{\mathsf{raf}}(r_{\text{cycle}})$ satisfies the the following equality:

$$\widetilde{\mathsf{raf}}(r_{\text{cycle}}) = \sum_{k \in \{0,1\}^{\log K}} \left(\sum_{i=0}^{\log(K)-1} 2^i \cdot k_i \right) \cdot \widetilde{\mathsf{ra}}(k, r_{\text{cycle}}), \tag{28}$$

Indeed, the right hand side of this equation is multilinear in r_{cycle} and agrees with raf whenever $r_{\text{cycle}} \in \{0,1\}^{\log T}$. Since $\{0,1\}^{\log T}$ is an interpolating set for multilinear polynomials, the left hand side and right hand side must be equal as formal polynomials in r_{cycle}.

Hence, if $\widetilde{\mathsf{raf}}(r_{\text{cycle}}) \neq y$, soundness of the sum-check protocol guarantees that the verifier will reject in the $\widetilde{\mathsf{raf}}$-evaluation sum-check invocation with probability at least $1 - 2\log(K)/|\mathbb{F}|$. This completes the proof of the soundness property stated in the theorem.

Perfect completeness is immediate from perfect completeness of the sum-check protocol, combined with the following facts:

- If every $\widetilde{\mathsf{ra}}(\cdot, j)$ is a valid one-hot encoding for all $j \in \{0,1\}^{\log T}$, then $\widetilde{\mathsf{ra}}(k, j) \in \{0, 1\}$ for all $(k, j) \in \{0,1\}^{\log(K)} \times \{0,1\}^{\log T}$, and hence $\widetilde{\mathsf{ra}}(k, j)^2 - \widetilde{\mathsf{ra}}(k, j) = 0$.
- If every $\widetilde{\mathsf{ra}}(\cdot, j)$ is a valid one-hot encoding for all $j \in \{0,1\}^{\log T}$, then $\sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j) = 1$ for all $j \in \{0,1\}^{\log T}$, and since the left hand side is a multilinear polynomial in j and $\{0,1\}^{\log T}$ is an interpolating set for such polynomials, this implies $\sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, j) = 1$ holds for all $j \in \mathbb{F}^{\log T}$.
- Equation (28) holds.

This completes the proof of the theorem. □

4.2 Shout for general d

Figure 7 has pseudocode for the core Shout PIOP for a general parameter d, and Figure 8 specifies the PIOP for proving correctness of d-dimensional one-hot encodings, for general d.

Theorem 3. *Figures 7 and 8 satisfy perfect completeness. Assuming each address $\widetilde{\mathsf{ra}}(\cdot, j)$ for $j \in \{0,1\}^{\log T}$ is indeed the d-dimensional one-hot encodings of some value $\widetilde{\mathsf{raf}}(j) \in \{0, 1, \ldots, K-1\}$, Figure 7 has soundness error at most*

$$\left((d+2)\log T + 2\log K\right)/|\mathbb{F}|.$$

Figure 8 has soundness error at most

$$\left(4d\log T + 6\log K\right)/|\mathbb{F}|.$$

1. As input, $\mathcal{P}$ has already committed to a multilinear polynomial $\widetilde{\mathsf{ra}}\colon \mathbb{F}^{\log K} \times \mathbb{F}^{\log T} \to \mathbb{F}$. $\mathcal{P}$ claims that for each $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(\cdot, j)$ is the one-hot representation of some value $\widetilde{\mathsf{raf}}(j) \in \{0, 1, \ldots, K-1\}$, and (optionally) that $\widetilde{\mathsf{raf}}(r_{\text{cycle}}) = y$.
2. $\mathcal{V}$ picks $r \in \mathbb{F}^{\log K}$ and $r' \in \mathbb{F}^{\log T}$ at random and sends (r, r') to $\mathcal{P}$.
3. **(Booleanity check)**: $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that

$$0 = \sum_{k \in \{0,1\}^{\log K},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j)\left(\widetilde{\mathsf{ra}}(k,j)^2 - \widetilde{\mathsf{ra}}(k,j)\right). \tag{29}$$

4. Let $(r'_{\text{address}}, r'_{\text{cycle}}) \in \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$ be the randomness chosen over the course of the sum-check protocol invoked in Line 3.
5. **(Hamming weight 1 check)**: $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that

$$1 = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r'_{\text{cycle}}).$$

6. **($\widetilde{\mathsf{raf}}$-evaluation sum-check)**: In parallel with the sum-check invocation in Line 5, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that

$$y = \sum_{k \in \{0,1\}^{\log K}} \left(\sum_{i=0}^{\log(K)-1} 2^i \cdot k_i \right) \cdot \widetilde{\mathsf{ra}}(k, r_{\text{cycle}}).$$

   Let r''_{address} be the randomness chosen by the verifier over the course of this sum-check instance.
7. To perform $\mathcal{V}$'s check in the final round of the two sum-checks (Lines 3 and 6), and $\mathcal{V}$'s check in Line 5, $\mathcal{V}$ evaluates $\widetilde{\mathsf{eq}}(r, r'_{\text{address}})$ and $\widetilde{\mathsf{eq}}(r', r'_{\text{cycle}})$ on its own, and evaluates $\widetilde{\mathsf{ra}}(r'_{\text{address}}, r_{\text{cycle}})$, and $\widetilde{\mathsf{ra}}(r''_{\text{address}}, r'_{\text{cycle}})$ with two queries to $\widetilde{\mathsf{ra}}$. Alternatively, standard techniques can reduce these two evaluations to a single evaluation of $\widetilde{\mathsf{ra}}$.

Figure 6: A PIOP for checking that for each $j \in \{0,1\}^T$, $\widetilde{\mathsf{ra}}(\cdot, j)$ is the (one-dimensional) one-hot representation of some value $\widetilde{\mathsf{raf}}(j) \in \{0, 1, \ldots, K-1\}$. This PIOP also grants the verifier query access to the virtual polynomial $\widetilde{\mathsf{raf}}$ (i.e., when only $\widetilde{\mathsf{ra}}$, not $\widetilde{\mathsf{raf}}$, is sent/committed by the prover).

That is, the verifier in Figure 7 will reject with overwhelming probability if there is any $j \in \{0,1\}^{\log T}$ such that $\widetilde{\mathsf{rv}}(j) \neq \widetilde{\mathsf{Val}}(k)$, where $k \in \{0,1\}^{\log K}$ is the binary representation of the address whose d-dimensional one-hot encoding is given by $(\widetilde{\mathsf{ra}}_1(j), \ldots, \widetilde{\mathsf{ra}}_d(j))$.

Proof. The completeness and soundness of Figure 8 is nearly identical to Theorem 2. The main difference is that in place of Equation (28), we now invoke that:

$$\widetilde{\mathsf{raf}}(r'_{\text{cycle}}) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j) \left(\sum_{i=1}^{d} \sum_{\ell=0}^{\log(K)/d-1} 2^{i \cdot \log(K)/d + \ell} \cdot k_{i,\ell} \right) \cdot \prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j).$$

This equality holds by the following reasoning. The right hand side is multilinear in r'_{cycle}. Since $\{0,1\}^{\log T}$ is an interpolating set for multilinear polynomials, we can conclude that the right hand side and left hand side are equal as a formal polynomials (and thus agree at all $r'_{\text{cycle}} \in \mathbb{F}^{\log T}$) so long as they agree whenever whenever $r'_{\text{cycle}} \in \{0,1\}^{\log T}$. To see that this is the case, observe that if r'_{cycle} and j are both in $\{0,1\}^{\log T}$,

1. $\mathcal{P}$ and $\mathcal{V}$ have agreed upon a size-K lookup table whose multilinear extension is given by $\widetilde{\mathsf{Val}}$ (which we assume to be evaluable in $O(\log K)$ time). $\mathcal{P}$ has already committed to multilinear polynomials $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d \colon \mathbb{F}^{\log(K)/d} \times \mathbb{F}^{\log T} \to \mathbb{F}$. $\mathcal{P}$ wishes to give the verifier query access to the virtual polynomial $\widetilde{\mathsf{rv}} \colon \mathbb{F}^{\log T} \to \mathbb{F}$, defined as the unique multilinear polynomial satisfying that: for all $j \in \{0,1\}^{\log T}$,

$$\widetilde{\mathsf{rv}}(j) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d} \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k).$$

2. $\mathcal{V} \to \mathcal{P}$: pick the desired evaluation point $r_{\text{cycle}} \in \mathbb{F}^{\log T}$ at random and send it to $\mathcal{P}$.
3. **(Read-checking for Shout)**: $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k). \tag{30}$$

4. Let $r_{\text{address}} = (r^{(1)}_{\text{address}}, \ldots, r^{(d)}_{\text{address}}) \in \left(\mathbb{F}^{\log(K)/d}\right)^d$ denote the randomness chosen over the first $\log(K)$ rounds of the sum-check protocol and r'_{cycle} the randomness over the final $\log T$ rounds. To perform $\mathcal{V}'s$ check in the final round of the sum-check protocol, $\mathcal{V}$ queries the committed polynomials $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$ respectively at $(r^{(i)}_{\text{address}}, r'_{\text{cycle}})$, and evaluates $\widetilde{\mathsf{Val}}$ at r_{address} in $O(\log K)$ time.

Figure 7: The core Shout PIOP with integer parameter $d > 1$, assuming the lookup table is MLE-structured (i.e., $\mathcal{V}$ can evaluate $\widetilde{\mathsf{Val}}$ at a random input in $O(\log K)$ time).

then $\widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j) = 0$ unless $j = r'_{\text{cycle}}$. In this case, $\widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j) = 1$, while

$$\mathsf{raf}(j) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d} \left(\sum_{i=1}^{d} \sum_{\ell=0}^{\log(K)/d-1} 2^{i \cdot \log(K)/d + \ell} \cdot k_{i,\ell}\right) \cdot \prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)$$

is immediate from the definition of d-dimensional one-hot encodings (Section 3.7).

Similarly, completeness and soundness of Figure 7 is nearly identical to Theorem 1, except that we replace Equation (23) with

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k). \tag{31}$$

To see that Equation (31) holds, observe that the right hand side is multilinear in r_{cycle}. Since $\{0,1\}^{\log T}$ is an interpolating set for multilinear polynomials, to show the equality holds for all $r_{\text{cycle}} \in \mathbb{F}^{\log T}$, it suffices to show the right hand side and left hand side agree whenever $r_{\text{cycle}} \in \{0,1\}^{\log T}$. To see this, observe that $\widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) = 0$ if $j \in \{0,1\}^{\log T}$ unless $j = r_{\text{cycle}}$, in which case $\widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) = 1$. Meanwhile, by the definition of d-dimensional one-hot encoding of addresses (Section 3.7), $\widetilde{\mathsf{rv}}(j) = \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k)$. □

4.2.1 A standard optimization for verifier costs: Batching parallel sum-check instances

A standard technique to control proof size when running $t > 1$ parallel sum-check instances is to instead apply a single instance of the sum-check protocol to a random linear combination of the t claims. In other words, the verifier picks $z \in \mathbb{F}$ at random, sends z to the prover, and then the prover applies the sum-check protocol a single time, to prove that:

1. As input, $\mathcal{P}$ has already committed to multilinear polynomials $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d \colon \mathbb{F}^{\log K} \times \mathbb{F}^{\log T} \to \mathbb{F}$. $\mathcal{P}$ claims that for each $j \in \{0,1\}^T$, $\widetilde{\mathsf{ra}}_1(\cdot, j), \ldots, \widetilde{\mathsf{ra}}_d(j)$ is the correct d-dimensional one-hot representation of some value $\widetilde{\mathsf{raf}}(j) \in \{0, 1, \ldots, K-1\}$, and also (optionally) that $\widetilde{\mathsf{raf}}(r'_{\text{cycle}}) = y$.
2. $\mathcal{V}$ picks $r \in \mathbb{F}^{\log(K)/d}$ and $r' \in \mathbb{F}^{\log T}$ at random and sends (r, r') to $\mathcal{P}$. Assuming that r'_{cycle} was chosen by the verifier at random after $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$ and $\widetilde{\mathsf{rv}}$ were committed, then $\mathcal{V}$ can set $r' = r'_{\text{cycle}}$.
3. **(Booleanity check)**: $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol d times in parallel to confirm that for $i = 1, \ldots, d$,
$$0 = \sum_{k \in \{0,1\}^{\log(K)/d},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, k)\widetilde{\mathsf{eq}}(r', j)\left(\widetilde{\mathsf{ra}}_i(k, j)^2 - \widetilde{\mathsf{ra}}_i(k, j)\right).$$
4. **(Hamming weight 1 check)**: Let $r''_{\text{cycle}} \in \mathbb{F}^{\log T}$ be any value chosen at random by $\mathcal{V}$ after $\widetilde{\mathsf{ra}}_i$ was committed. For each $i = 1, \ldots, d$, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that
$$1 = \sum_{k_i \in \{0,1\}^{\log(K)/d}} \widetilde{\mathsf{ra}}_i(k_i, r''_{\text{cycle}}).$$
5. **($\widetilde{\mathsf{raf}}$-evaluation sum-check)**: In parallel with the Booleanity check (and the read-checking sum-check from the core Shout PIOP, see Line 3 of Figure 7), $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that
$$y = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j)\left(\sum_{i=1}^{d} \sum_{\ell=0}^{\log(K)/d-1} 2^{i \cdot \log(K)/d + \ell} \cdot k_{i,\ell}\right) \cdot \prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j).$$

Figure 8: A PIOP for checking that for each $j \in \{0,1\}^T$, $\widetilde{\mathsf{ra}}_1(\cdot, j), \ldots, \widetilde{\mathsf{ra}}_d(j)$ is the correct d-dimensional one-hot representation of some value $\widetilde{\mathsf{raf}}(j) \in \{0, 1, \ldots, K-1\}$, and also granting the verifier query access to the virtual polynomial $\widetilde{\mathsf{raf}}$ (i.e., enabling only $\widetilde{\mathsf{ra}}$, not $\widetilde{\mathsf{raf}}$, to be sent/committed by the prover). Some hashing-based commitment schemes directly ensure Booleanity of committed values, in which case the Booleanity check (Line 3) can be omitted.

$$0 = \sum_{i=1}^{t} z^{\ell-1} \cdot \sum_{k \in \{0,1\}^{\log(K)/d},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, k)\widetilde{\mathsf{eq}}(r', j)\left(\widetilde{\mathsf{ra}}_i(k, j)^2 - \widetilde{\mathsf{ra}}_i(k, j)\right). \tag{32}$$

This adds at most $t/|\mathbb{F}|$ to the soundness error, and avoids a t-fold increase in proof size. In the context of Twist and Shout, it can also have prover time benefits (see for example Section 6.3).

5 The Twist PIOP

The core Twist PIOP is given in full in Figure 9. The PIOP in this figure is sound assuming that all one-hot encodings are provided correctly, i.e., that for all $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(j)$ are $\widetilde{\mathsf{wa}}(j)$ are the correct d-dimensional one-hot encodings of some values $\widetilde{\mathsf{raf}}(j)$ and $\widetilde{\mathsf{waf}}(j)$ in $\{0, 1, \ldots, K-1\}$. To confirm this, one can invoke the one-hot-encoding-checking PIOP of Figure 8 (replacing $\widetilde{\mathsf{ra}}_i$ and $\widetilde{\mathsf{raf}}$ with $\widetilde{\mathsf{wa}}_i$ and $\widetilde{\mathsf{waf}}$ as appropriate).

Theorem 4. *Figure 9 has perfect completeness and soundness error at most*
$$\left((2d+3)\log T + 3\log K\right)/|\mathbb{F}|.$$
That is, the verifier in Figure 9 will reject with overwhelming probability if the claimed value of $\widetilde{\mathsf{rv}}(r')$ does

1. As input, $\mathcal{P}$ has already committed to the multilinear polynomials $\widetilde{\mathsf{Inc}}\colon \mathbb{F}^{\log K} \times \mathbb{F}^{\log T} \to \mathbb{F}$, $\widetilde{\mathsf{wv}}\colon \mathbb{F}^{\log T} \to \mathbb{F}$, and $\widetilde{\mathsf{ra}}_1, \widetilde{\mathsf{wa}}_1, \ldots, \widetilde{\mathsf{ra}}_d, \widetilde{\mathsf{wa}}_d\colon \mathbb{F}^{\log(K)/d} \times \mathbb{F}^{\log T} \to \mathbb{F}$. If the prover is honest, then $\widetilde{\mathsf{Inc}}$ is the MLE of the vector Inc defined via Equation (9). $\widetilde{\mathsf{Val}}\colon \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$ is a virtual polynomial that is defined as
$$\widetilde{\mathsf{Val}}(k, j) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(k, j)\widetilde{\mathsf{LT}}(j', j).$$
The prover wishes to give the verifier query access to the virtual polynomial $\widetilde{\mathsf{rv}}$, defined as the unique multilinear polynomial satisfying that for all $j \in \{0,1\}^{\log T}$,
$$\widetilde{\mathsf{rv}}(j) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K/d)}\right)^d} \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k, j),$$
where $\widetilde{\mathsf{Val}}(k, j)$ is the value stored in register k during cycle j according to $\widetilde{\mathsf{wa}}$ and $\widetilde{\mathsf{wv}}$.
2. $\mathcal{V} \to \mathcal{P}$: pick a desired evaluation point $r' \in \mathbb{F}^{\log T}$ for $\widetilde{\mathsf{rv}}$ and send it to $\mathcal{P}$. Also pick $r \in \mathbb{F}^{\log K}$ at random and send it to $\mathcal{P}$. Then $\mathcal{V}$ and $\mathcal{P}$ run the following two instances of the sum-check protocol in parallel:
3. **Read-checking sum-check.** $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to compute
$$\widetilde{\mathsf{rv}}(r') = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', j) \cdot \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k, j) \tag{33}$$
4. **Write-checking sum-check**. In parallel with the read-checking sum-check, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that $\widetilde{\mathsf{Inc}}(r, r')$ equals:
$$\sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, k) \cdot \widetilde{\mathsf{eq}}(r', j) \cdot \left(\left(\prod_{i=1}^{d} \widetilde{\mathsf{wa}}_i(k_i, j)\right) \cdot \left(\widetilde{\mathsf{wv}}(j) - \widetilde{\mathsf{Val}}(k, j)\right)\right). \tag{34}$$
5. At the end of the above instances of the sum-check protocol, for random values $r_{\text{address}}, r_{\text{cycle}}$, the verifier has to evaluate $\widetilde{\mathsf{wa}}(r_{\text{address}}, r_{\text{cycle}})$, $\widetilde{\mathsf{wv}}(r_{\text{cycle}})$, $\widetilde{\mathsf{ra}}(r_{\text{address}}, r_{\text{cycle}})$ and $\widetilde{\mathsf{Val}}(r_{\text{address}}, r_{\text{cycle}})$. The $\widetilde{\mathsf{wa}}$, $\widetilde{\mathsf{wv}}$, and $\widetilde{\mathsf{ra}}$ evaluations are obtained directly from the commitments to these polynomials. The $\widetilde{\mathsf{Val}}$ evaluation is obtained from the $\widetilde{\mathsf{Val}}$-evaluation sum-check below.
6. $\widetilde{\mathsf{Val}}$**-evaluation sum-check.** $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to compute
$$\widetilde{\mathsf{Val}}(r_{\text{address}}, r_{\text{cycle}}) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(r_{\text{address}}, j') \cdot \widetilde{\mathsf{LT}}(j', r_{\text{cycle}}).$$
7. To perform $\mathcal{V}'s$ check in the final round of this sum-check protocol, the verifier has to evaluate $\widetilde{\mathsf{Inc}}$ at a random point and $\widetilde{\mathsf{LT}}$. The verifier computes the $\widetilde{\mathsf{LT}}$ evaluation on its own in $O(\log T)$ time. The evaluation of $\widetilde{\mathsf{Inc}}$ is obtained via the commitment to $\widetilde{\mathsf{Inc}}$.

Figure 9: The core Twist PIOP for parameter $d \geq 1$, ignoring checking correctness of one-hot decompositions (which is covered in Figure 8).

not actually equal $\widetilde{\mathsf{rv}}(r')$ *for the unique multilinear polynomial satisfying that for all* $j \in \{0,1\}^{\log T}$,
$$\widetilde{\mathsf{rv}}(j) \neq \widetilde{\mathsf{Val}}(k, j), \tag{35}$$
where $k \in \{0,1\}^{\log K}$ *is the binary representation of the address whose* d*-dimensional one-hot encoding is given by* $(\widetilde{\mathsf{ra}}_1(j), \ldots, \widetilde{\mathsf{ra}}_d(j))$*, and* $\widetilde{\mathsf{Val}}(k, j)$ *is the value most recently written to address* k *prior to the* j*'th*

write operation.

Proof. We begin with the soundness analysis.

Defining some polynomials. Let $\widetilde{\mathsf{cInc}}$ denote the actual MLE of the "correct" increments implies by the write operations specified by $\widetilde{\mathsf{wa}}$ and $\widetilde{\mathsf{wv}}$ (as opposed to $\widetilde{\mathsf{Inc}}$, which denotes the committed multilinear polynomial that is *claimed* by the prover to equal $\widetilde{\mathsf{cInc}}$). Similarly, let $\widetilde{\mathsf{crv}}$ denote the MLE of the "correct" values returned by each read operation. Finally, let $\widetilde{\mathsf{cVal}}(k,j)$ denote the MLE of the "correct" values stored in register k after $j-1$ writes have occurred, while $\widetilde{\mathsf{Val}}(k,j)$ denotes the multilinear polynomial defined via:

$$\widetilde{\mathsf{Val}}(k,j) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(k,j') \cdot \widetilde{\mathsf{LT}}(j',j). \tag{36}$$

Relationships between these polynomials. Note that for all $(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$,

$$\widetilde{\mathsf{cInc}}(k,j) = \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k,j)\right)\left(\widetilde{\mathsf{wv}}(j) - \widetilde{\mathsf{cVal}}(k,j)\right). \tag{37}$$

(This is not an equality of formal polynomials, since the right hand side is not multilinear in k and j, but the equality does hold at all inputs in $\{0,1\}^{\log K} \times \{0,1\}^{\log T}$).

In addition,

$$\widetilde{\mathsf{cVal}}(k,j) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{cInc}}(k,j') \cdot \widetilde{\mathsf{LT}}(j',j). \tag{38}$$

This is an equality of formal polynomials, since the right hand side is multilinear in k and j and the agrees with the left hand side pointwise over the interpolating set $\{0,1\}^{\log K} \times \{0,1\}^{\log T}$.

Finally, note that

$$\widetilde{\mathsf{crv}}(r') = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r',j) \cdot \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i,j)\right) \cdot \widetilde{\mathsf{cVal}}(k,j). \tag{39}$$

This too is an equality of formal polynomials. Indeed, the left and side and right hand side are both multilinear in r', so we need only confirm that they agree at all r' in the interpolating set $\{0,1\}^{\log T}$. When r' is in this set, then $\widetilde{\mathsf{eq}}(r',j)$ equals 0 unless $j = j'$, in which case $\widetilde{\mathsf{eq}}(r',j) = 1$. And for $j = r'$, it indeed holds that

$$\widetilde{\mathsf{crv}}(r') = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d} \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i,j)\right) \cdot \widetilde{\mathsf{cVal}}(k,j).$$

Overview of the soundness analysis. Informally, the soundness analysis proceeds via a three-step argument:

- The read-checking sum-check correctly grants query access to $\widetilde{\mathsf{crv}}$ *assuming* $\widetilde{\mathsf{Val}} = \widetilde{\mathsf{cVal}}$.
- The write-checking sum-check makes sure that $\widetilde{\mathsf{Inc}} = \widetilde{\mathsf{cInc}}$, also *assuming* $\widetilde{\mathsf{Val}} = \widetilde{\mathsf{cVal}}$.
- The $\widetilde{\mathsf{Val}}$-evaluation sum-check makes sure that $\widetilde{\mathsf{Val}} = \widetilde{\mathsf{cVal}}$ *assuming* that $\widetilde{\mathsf{Inc}} = \widetilde{\mathsf{cInc}}$.

As written, the second and third bulletpoints introduce circular assumptions, so the soundness analysis cannot actually proceed in this manner. Fortunately, the assumption in the third bulletpoint is stronger than what is actually needed. In order for the $\widetilde{\mathsf{Val}}$-evaluation sum-check to ensure that $\widetilde{\mathsf{Val}}(k,j) = \widetilde{\mathsf{cVal}}(k,j)$ for a given $k \in \{0,1\}^{\log K}$ and $j \in \{0,1\}^{\log T}$, one need only assume that $\widetilde{\mathsf{Inc}}(k,j') = \widetilde{\mathsf{cInc}}(k,j')$ for all j' such that $\mathsf{LT}(j',j) = 1$ (which we'll henceforth denote by $j' < j$ as shorthand). This means that we can avoid circular assumptions in our soundness analysis by focusing on the smallest such j for which $\widetilde{\mathsf{Val}}(k,j) \neq \widetilde{\mathsf{cVal}}(k,j)$.

Formal soundness analysis. First, suppose that $\widetilde{\mathsf{Inc}}$ and $\widetilde{\mathsf{cInc}}$ are not the same polynomial. Since they are both multilinear and $\{0,1\}^{\log K} \times \{0,1\}^{\log T}$ is an interpolating set for multilinear polynomials, this means there is at least one input $(k, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$ such that $\widetilde{\mathsf{Inc}}(k, j) \neq \widetilde{\mathsf{cInc}}(k, j)$. Let (k^*, j^*) be one such input with the smallest possible value of j^*. That is, for all $j' < j^*$ and all $k \in \{0,1\}^{\log K}$, we assume that $\widetilde{\mathsf{Inc}}(k, j') = \widetilde{\mathsf{cInc}}(k, j')$. It follows then, by the definition of $\widetilde{\mathsf{Val}}$ (Equation (36)), that $\widetilde{\mathsf{Val}}(k^*, j^*) = \widetilde{\mathsf{cVal}}(k^*, j^*)$. But this means that

$$\widetilde{\mathsf{cInc}}(k^*, j^*) = \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k^*, j^*)\right)\left(\widetilde{\mathsf{wv}}(j^*) - \widetilde{\mathsf{cVal}}(k^*, j^*)\right) = \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k^*, j^*)\right)\left(\widetilde{\mathsf{wv}}(j^*) - \widetilde{\mathsf{Val}}(k^*, j^*)\right),$$

where the first equality holds by Equation (37) and the final equality holds because $\widetilde{\mathsf{Val}}(k^*, j^*) = \widetilde{\mathsf{cVal}}(k^*, j^*)$. Since $\widetilde{\mathsf{Inc}}(k^*, j^*) \neq \widetilde{\mathsf{cInc}}(k^*, j^*)$ by assumption, we conclude that

$$\widetilde{\mathsf{Inc}}(k^*, j^*) \neq \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k^*, j^*)\right)\left(\widetilde{\mathsf{wv}}(j^*) - \widetilde{\mathsf{Val}}(k^*, j^*)\right). \tag{40}$$

Let

$$h(k^*, j^*) = \sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(k^*, k) \cdot \widetilde{\mathsf{eq}}(j^*, j) \cdot \left(\prod_{i=1}^{d} \widetilde{\mathsf{wa}}_i(k_i, j)\right) \cdot \left(\widetilde{\mathsf{wv}}(j) - \widetilde{\mathsf{Val}}(k, j)\right).$$

It is easily seen that $h(k^*, j^*)$ equals the right hand side of Expression (40), and hence $\widetilde{\mathsf{Inc}}(k^*, j^*) \neq h(k^*, j^*)$. Since h and $\widetilde{\mathsf{Inc}}$ are both multilinear polynomials, and they are distinct, the Schwartz-Zippel lemma implies that with probability at least

$$1 - (\log(K) + \log(T))/|\mathbb{F}|,$$

$\widetilde{\mathsf{Inc}}(r, r') \neq h(r, r')$ when (r, r') is chosen at random from $\mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$. By soundness of the sum-check protocol, in this event the write-checking sum-check verifier rejects with probability at least

$$1 - (d\log(T) + \log(K))/|\mathbb{F}|.$$

Thus, if the verifier accepts with probability more than $((d+1)\log(T) + 2\log(K))/|\mathbb{F}|$, then $\widetilde{\mathsf{Inc}} = \widetilde{\mathsf{cInc}}$. We assume henceforth that indeed $\widetilde{\mathsf{Inc}} = \widetilde{\mathsf{cInc}}$. Then by Equation (38) and soundness of the sum-check protocol as invoked, we conclude (up to an additional soundness error of $2\log(T)/|\mathbb{F}|$) that for whatever evaluation point $(r_{\text{address}}, r_{\text{cycle}})$ on which the $\widetilde{\mathsf{Val}}$-evaluation sum-check is invoked, it holds that

$$\widetilde{\mathsf{Val}}(r_{\text{address}}, r_{\text{cycle}}) = \widetilde{\mathsf{cVal}}(r_{\text{address}}, r_{\text{cycle}}). \tag{41}$$

Absorbing $2\log(T)/|\mathbb{F}|$ into the soundness error, we assume henceforth that $\widetilde{\mathsf{Val}}$ and $\widetilde{\mathsf{cVal}}$ agree at the point $(r_{\text{address}}, r_{\text{cycle}})$ at which the $\widetilde{\mathsf{Val}}$-evaluation sum-check is applied.

It follows by Equation (39) and soundness of the sum-check protocol, the verifier will reject during the read-checking sum-check with probability at least $(d\log(T) + \log(K))/|\mathbb{F}|)$. This completes the soundness analysis.

The total soundness error according to the above analysis is at most $((2d+3)\log(T) + 3\log(K))/|\mathbb{F}|$.

Perfect completeness is an easy consequence of completeness of the sum-check protocol and Equations (37) and (38). Equation (37) ensures that the write-checking sum-check is perfectly complete and Equation (38) ensures that the $\widetilde{\mathsf{Val}}$-evaluation sum-check is perfectly complete.

□

6 Fast Shout prover implementation (small memories)

This section explains how to implement the prover in the various invocations of the sum-check protocol in Shout. Our treatment assumes familiarity with the standard linear-time sum-check prover implementation [CTY11, Tha13] covered in the preliminaries (Section 3.3). In this section, we are generally focused upon the case that $K = o(T)$, and hence we do not seek to avoid additive terms in the prover time of $O(K)$. This is what we mean by "small" memories in the section title.

Because Shout with $d = 1$ is a special case, we present the fast prover for $d = 1$ in Section 6.1. before turning to the general case (Figures 7 and 8) in Sections 6.2 and 6.3.

6.1 Core Shout prover for $d = 1$

An algorithm for small or unstructured memories. Recall that the core Shout PIOP prover with $d = 1$ (Figure 5) applies the sum-check protocol to prove that

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r_{\text{cycle}}) \cdot \widetilde{\mathsf{Val}}(k). \tag{42}$$

With T field multiplications, the prover can compute a vector E storing $\widetilde{\mathsf{eq}}(j, r_{\text{cycle}})$ as j ranges over $\{0,1\}^{\log T}$ [VSBW13] (see Lemma 1). Given E, computing a vector F storing all $\widetilde{\mathsf{ra}}(k, r_{\text{cycle}})$ evaluations for all $k \in \{0,1\}^{\log K}$ can be done with only additions and lookups into E. Specifically, $\widetilde{\mathsf{ra}}(k, r_{\text{cycle}}) = \sum_{j\colon \widetilde{\mathsf{ra}}(k,j)=1} \widetilde{\mathsf{eq}}(j, r_{\text{cycle}})$. Given this array, the standard linear-time sum-check proving algorithm (Section 3.3) requires $3K$ field multiplications.[30] Hence, the total number of field multiplications done by the core Shout PIOP prover is: $T + 3K$.

Theorem 5. *The prover in the core Shout PIOP with $d = 1$ (Figure 5) can be implemented in $3K + T$ field multiplications.*

An algorithm for large, structured memories. If $K \gg T$, we would not be happy with the additive $2K$ factor in the prover runtime above. However, we generally do not expect Shout to be applied with d set to 1 in this case, as it would either lead to a very large commitment key (in the case of an elliptic curve commitment) or high time to compute commitments (in the case of a hashing-based commitment scheme, where 0s are not free to commit to). Still, for completeness, we outline a prover implementation that avoids this linear-in-K term. This implementation is an immediate consequence of the sparse-dense sum-check protocol from [STW24, Appendix G].

Indeed, if $T \gg K$ the vector F above is *sparse*: only (at most) T of its entries are non-zero. Suppose that $K \leq T^c$ for some constant $c > 0$. The sparse-dense sum-check protocol of [STW24, Appendix G] identifies conditions on $\widetilde{\mathsf{Val}}$ that guarantee that the prover (in the sum-check protocol invoked to establish Equation (42) holds) runs in time $O(c \cdot T)$. [STW24, Appendix G] also showed that important lookup tables of size 2^{64} arising in Jolt satisfy the necessary structural conditions for this runtime bound to hold. See Section 7.1 for further details.

6.2 Core Shout prover (general d, small memories)

Recall that the core Shout PIOP for any $d \geq 1$ (Figure 7) applies the sum-check protocol to check that

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j) \right) \cdot \widetilde{\mathsf{Val}}(k). \tag{43}$$

[30] Section 3.3 states a time bound of $2K$ field multiplications, but this does not account for the cost of computing the claimed answer, in this case $\widetilde{\mathsf{ra}}(k, r_{\text{cycle}})$. Given the array F, $\widetilde{\mathsf{ra}}(k, r_{\text{cycle}})$ can be computed with K additional field operations.

Before the sum-check protocol even begins, the prover can compute the following array E^* with one entry per $j \in \{0,1\}^{\log T}$.

$$E^* \text{ stores } \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) \text{ for all } j \in \{0,1\}^{\log T}. \tag{44}$$

This array E^* is useful later in the protocol.

Turning to the sum-check invocation, we bind the variables of the register $k \in \{0,1\}^{\log K}$ before the variables of the cycle $j \in \{0,1\}^{\log T}$.

6.2.1 The first $\log K$ rounds of sum-check

The prover's message in the first $\log K$ rounds is independent of d, in the following sense: the prover messages sent in the first $\log K$ rounds is the same if the term

$$\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j) \tag{45}$$

in Expression (43) is replaced with the multilinear polynomial $\widetilde{\mathsf{ra}}(k_1, \ldots, k_d, j)$ that takes the same values as Expression (45) over inputs in $\{0,1\}^{\log K} \times \{0,1\}^{\log T}$. This is because the first $\log(K)$ rounds of the sum-check protocol bind variables only the variables in $k = (k_1, \ldots, k_d)$, and Expression (45) has degree one in the variables of k—it only has degree more than one in the variables of j. So changing the degree of Expression (43) in j without changing its evaluations over the relevant domain does not alter the sum-check prover's message in the first $\log K$ rounds.

Hence, for simplicity of notation, let us focus on the case the $d = 1$ for the first $\log K$ rounds.

Our first key observation is that for the first $\log K$ rounds, and each register $k \in \{0,1\}^{\log K}$, each cycle at which register k is read "contributes identically" to Expression (43). Accordingly, for each register k, we can "aggregate" together at the start of the protocol all cycles j at which register k is read. Details follow.

A key fact. The following is a key consequence of the fact that $\mathsf{ra}(k, j)$ is a unit vector for each cycle j. Consider the set of values $\widetilde{\mathsf{ra}}(r_1, \ldots, r_m, y)$ as $y = (y'_{\mathsf{mem}}, y_{\mathsf{cycle}})$ ranges over $\{0,1\}^{\log(K)-m} \times \{0,1\}^{\log T}$. For any round $m \leq \log K$, for each $y_{\mathsf{cycle}} \in \{0,1\}^{\log T}$, this set of values is highly structured: there is exactly one $y'_{\mathsf{mem}} \in \{0,1\}^{\log(K)-m}$ for which $\widetilde{\mathsf{ra}}(r_1, \ldots, r_m, y'_{\mathsf{mem}}, y_{\mathsf{cycle}})$ is non-zero, and if register $k \in \{0,1\}^{\log K}$ was read in cycle y_{cycle}, then this non-zero value equals

$$\widetilde{\mathsf{eq}}((k_1, \ldots, k_m), (r_1, \ldots, r_m)). \tag{46}$$

The prover's data structures for the first $\log K$ rounds. The prover maintains two arrays that initially store K values and halve in size each round. Let's call these two arrays A and C.

$A[k]$ initially stores $\widetilde{\mathsf{Val}}(k)$. And $C[k]$ initially stores

$$v_k \coloneqq \sum_{j \in \{0,1\}^{\log T} \colon \widetilde{\mathsf{ra}}(k,j)=1} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j). \tag{47}$$

Initializing C costs no field multiplications since each entry of C is a sum of entries of the already-computed array E^* (Expression (44)).

At the end of round m, when the verifier sends the prover the random challenge r_m, the prover binds the A and C arrays as in Section 3.3. That is, for each $k \in \{0,1\}^{\log(K)-m}$, the prover sets

$$A[k] \leftarrow (1 - r_m)A[k, 0] + r_m A[k, 1] = A[k, 0] + r_m \cdot (A[k, 1] - A[k, 0])$$

$$C[k] \leftarrow (1 - r_m)C[k, 0] + r_m C[k, 1] = C[k, 0] + r_m \cdot (C[k, 1] - C[k, 0]).$$

By Fact 3.1, this ensures that after round m, $A[k]$ stores $\widetilde{\mathsf{Val}}(r_1, \ldots, r_m, k)$ and

$$C[k] = \sum_{k^* = (k', k) \in \{0,1\}^m \times \{0,1\}^{\log(K)-m}} v_{k^*} \cdot \widetilde{\mathsf{eq}}(r_1, \ldots, r_m, k'). \tag{48}$$

Leveraging the data structures. In the middle of round m, for any $k \in \{0,1\}^{\log(K)-m}$, let us use

$$A[c', k]$$

as shorthand for

$$(1 - c') \cdot A[0, k] + c' \cdot A[1, k], \tag{49}$$

and similarly with C in place of A. Then with the above data structures, the following holds. For each $c' \in \{0, 2\}$, the prescribed prover message s_m in round m includes $s_m(c')$, which equals:

$$s_m(c') = \sum_{k \in \{0,1\}^{\log(K)-m}} \widetilde{\mathsf{Val}}(r_1, \ldots, r_{m-1}, c', k) \left(\sum_{j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r'', j) \widetilde{\mathsf{ra}}(r_1, \ldots, r_{m-1}, c', k, j) \right) \tag{50}$$

$$= \sum_{k \in \{0,1\}^{\log(K)-m}} A[c', k] \cdot C[c', k]. \tag{51}$$

This last equality exploits Fact 3.1, as well as the key fact above, that for each $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(r_1, \ldots, r_{m-1}, c', k, j)$ is non-zero for exactly one value of $k \in \{0,1\}^{\log(K)-m}$, and per Equation (46), at this k its value is precisely $\widetilde{\mathsf{eq}}(k', r_1, \ldots, r_{m-1}, c')$ where k' is the first m coordinates of the register read at cycle j. By Equation (48), this means that the quantity in parenthesis in Expression (50) precisely equals $C[c', k]$.

Thus, we have shown that the prover in the first $\log K$ rounds of the sum-check protocol applied to Expression (43), the prover performs at most $4K$ multiplications (not counting the construction of E^* which we already charged for in the evaluation of $\widetilde{\mathsf{rv}}(r_{\text{cycle}})$). Here is the accounting:

- K multiplications to bind the A array.
- K to bind the C array.
- $3 \cdot K$ across all rounds m to evaluate Expression (51) given these arrays (see Footnote 30 for an explanation of why this cost is $3K$ and not $2K$).

6.2.2 The final $\log T$ rounds of sum-check

The degree of the univariate polynomial the prover sends in each of the last d rounds is $d + 1$, so the prover time in the last d rounds does grow with d. Nonetheless, for simplicity let us begin by considering the case $d = 1$. Let $\tau = (r_1, \ldots, r_{\log K}) \in \mathbb{F}^{\log K}$ denote the random challenges chosen during the first $\log K$ rounds of Shout's sum-check protocol invocation. Per Equation (43), the final $\log T$ rounds are intended to compute:

$$\widetilde{\mathsf{Val}}(\tau) \sum_{j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) \cdot \widetilde{\mathsf{ra}}(\tau, j). \tag{52}$$

With K field multiplications, the prover can compute an array E of length T whose j'th entry stores $\widetilde{\mathsf{ra}}(\tau, j)$ (as j ranges over $\{0,1\}^{\log T}$). Indeed, per Equation (46), $\widetilde{\mathsf{ra}}(\tau, j)$ simply equals $\widetilde{\mathsf{eq}}(\tau, k)$ where $k \in \{0,1\}^{\log K}$ is the register read at cycle j. And it's known how to compute a size-K array storing all evaluations $\widetilde{\mathsf{eq}}(\tau, k)$ for $k \in \{0,1\}^{\log K}$ with K field multiplications (see Lemma 1). Each entry of E can be computed with one lookup into this size-K array.

With this array E in hand, as well as the array E^* computed before round one (see Expression (44)), the standard linear-time sum-check proving algorithm (Section 3.3) implements the sum-check prover applied to compute Expression (52) with only about $4T$ field multiplications: T for binding the array E round-over-round, T for binding the array E^* round-over-round, and $2T$ more for evaluating $s_i(0)$ and $s_i(2)$ in each round $i = \log(K) + 1, \ldots, \log(K) + \log(T)$. In fact, now-standard optimizations [DT24, Gru24] reduce this to $2T$, as the optimization of [Gru24, Section 3] covered in Section 3.6.1 eliminates the need to evaluate $s_i(2)$ and [DT24] eliminates the cost of binding the array E^*. We even discuss below a final optimization that effectively eliminates the cost of binding E when $K = o(T)$. That brings the total number of prover multiplications for the final $\log T$ rounds down to only about T when $d = 1$ and $K = o(T)$.

An optimization leveraging small memory size. When $K = o(T)$, the following additional optimization applies. Since $\widetilde{\mathsf{ra}}(\tau, j)$ takes on at most K distinct values across all $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}(\tau, r_{\log(K)+1}, \ldots, r_m, j)$ takes on at most K^{2^m} distinct values as j ranges over $\{0,1\}^{\log(T)-m}$. In fact, each of these K^{2^m} values is a size-2^m *sum* of values from a set $\mathcal{S}$ of size $K \cdot 2^m$ quantities that can all be computed in $O(K2^m)$ time. Namely, $\mathcal{S}$ consists of each of the K distinct values in $\widetilde{\mathsf{ra}}(\tau, j)$, times a value of the form $\widetilde{\mathsf{eq}}(r_{\log(K)+1}, \ldots, r_m, j')$ as j' ranges over $\{0,1\}^m$. This means that as long as $K \cdot 2^m \ll T$, we can speed up the task of binding $\widetilde{\mathsf{ra}}$ for each of the first m rounds. Rather than costing $T/2^j$ field multiplications at the end of round j, it can instead cost $O(K \cdot 2^m)$ field multiplications.

General d. For general $d > 1$, the prover in each of the final $\log T$ rounds sends a univariate polynomial of degree $d+1$. There are also d arrays tracking evaluations of $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$, rather than the single array E considered above when we focused on the case that $d = 1$. As long as $K^{1/d} = o(T)$, the optimization above that nearly eliminates the cost of binding E applies to binding each of these d arrays.

Thus, when $K = o(T)$, the prover cost of the final $\log(T)$ rounds for general d is, up to low-order terms, d^2T field multiplications. This simply amounts to applying the standard linear-time sum-check prover algorithm from Section 3.3, combined with the various optimizations described above.

To summarize the costs:

- T field multiplications to compute the array E^*.
- $5K$ additional multiplications to implement the first $\log K$ rounds.
- dT multiplications to bind the arrays of $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$ evaluations over the final $\log T$ rounds. This cost falls to $o(T)$ if $K^{1/d} = o(T)$.
- d^2T multiplications given the above arrays to evaluate the prover's messages over the final $\log T$ rounds.

We have established the following theorem.

Theorem 6. *For $d > 1$, the core Shout PIOP prover (Figure 7) can be implemented with $(d^2 + d + 1)T + 5K + o(T)$ field multiplications. If $K^{1/d} = o(T)$, this falls to $(d^2+1)T + 5K + o(T)$.*

6.3 Booleanity-checking and one-hot-encoding-checking

Here is how to efficiently implement the prover in the PIOP of Figure 8 when an additive $O(K^{1/d}\log(K))$ term in the prover cost is acceptable. When $K^{1/d}\log(K)$ is bigger than T, one should instead use our generalization of the sparse-dense sum-check protocol given later, in Section 7 (that protocol has a much better dependence on K, but a worse dependence on T, by a significant constant factor).

Let us start with the Booleanity-checking sum-check invocation (Line 3 of Figure 8), which for each $i = 1, \ldots, d$ applies the sum-check protocol to confirm that

$$0 = \sum_{k \in \{0,1\}^{\log(K)/d},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j)\left(\widetilde{\mathsf{ra}}_i(k,j)^2 - \widetilde{\mathsf{ra}}_i(k,j)\right).$$

We bind the $\log(K)/d$ variables of k first, followed by the $\log T$ variables of j.

First $\log(K)/d$ rounds. Before the first round, the prover builds arrays B and D of size $K^{1/d}$ and T respectively, storing the following values:

$$B \text{ stores } \mathsf{eq}(r,k) \text{ for all } k \in \{0,1\}^{\log(K)/d} \tag{53}$$

and

$$D \text{ stores } \mathsf{eq}(r',j) \text{ for all } j \in \{0,1\}^{\log T}. \tag{54}$$

By Lemma 1, computing B requires $K^{1/d}$ field multiplications, while computing D requires T field multiplications.

At the end of round m, let $\mathsf{r} = (\mathsf{r}_1, \ldots, \mathsf{r}_m)$ denote the random values chosen by the sum-check prover in the first m rounds. Via standard binding techniques (Section 3.3), the prover can ensure that at the end of round m, the array B has shrunk to length $K^{1/d}/2^m$, and for every $k' \in \{0,1\}^{\log(K)/d-m}$, $B[k']$ stores $\widetilde{\mathsf{eq}}(r, \mathsf{r}_1, \ldots, \mathsf{r}_m, k')$.

Recall from Equation (46) that for at the end of any round $m \leq \log(K)/d$ when r_m is selected, for each $y_{\mathsf{cycle}} \in \{0,1\}^{\log T}$, there is exactly one $y'_{\mathsf{mem}} \in \{0,1\}^{\log(K)/d-m}$ for which $\widetilde{\mathsf{ra}}_i(\mathsf{r}_1, \ldots, \mathsf{r}_m, y'_{\mathsf{mem}}, y_{\mathsf{cycle}})$ is non-zero, and if register $k \in \{0,1\}^{\log K}$ was read in cycle y_{cycle}, then this non-zero value equals

$$\widetilde{\mathsf{eq}}((k_1, \ldots, k_m), (\mathsf{r}_1, \ldots, \mathsf{r}_m)).$$

The prover can maintain an array F that at the end of round m has size 2^m and stores all 2^m such values, i.e.,

$$F \text{ stores } \widetilde{\mathsf{eq}}((k_1, \ldots, k_m), (\mathsf{r}_1, \ldots, \mathsf{r}_m)) \text{ for all } (k_1, \ldots, k_m) \in \{0,1\}^m \tag{55}$$

Via standard techniques (see Lemma 1), maintaining F costs $K^{1/d}$ field multiplications across the first $\log(K)/d$ rounds.

Next, consider a specific i from the set $\{1, \ldots, d\}$. The prover can, with only additions, before the protocol begins, compute a size-$K^{1/d}$ array G_i that for $k \in \{0,1\}^{\log(K)/d}$ stores

$$G_i[k] = \sum_{j \in \{0,1\}^{\log T} :\, \widetilde{\mathsf{ra}}_i(k,j)=1} D[j].$$

In the middle of round m, as per Expression (49), let us use $B[c', k']$ as shorthand for

$$(1 - c') \cdot B[0, k'] + c' \cdot B[1, k'].$$

and $F[k_1, \ldots, k_{m-1}, c']$ as shorthand for

$$F[k_1, \ldots, k_{m-1}] \cdot \widetilde{\mathsf{eq}}(k_m, c'),$$

and where in the middle of round $m = 1$, the factor $F[k_1, \ldots, k_{m-1}]$ is simply 1.

The prover's message s_m in round $m \leq \log(K)/d$ of sum-check for the i'th Booleanity check satisfies that $s_m(c)$ equals:

$$\sum_{k=(k_1,\ldots,k_m,k') \in \{0,1\}^m \times \{0,1\}^{\log(K)/d-m}} G_i[k] \cdot B[c, k'] \cdot \left(F[k_1, \ldots, k_{m-1}, c]^2 - F[k_1, \ldots, k_{m-1}, c]\right). \tag{56}$$

In round m, the prover has to evaluate $s_m(c)$ for all relevant values of c. Using Gruen's technique (Section 3.6.1), there are 2 relevant values of c, say $c \in \{0, 2\}$. Given the arrays G_i, B, and F, both $s_m(0)$ and $s_m(2)$ can be computed via Expression (56) with the following total number of multiplications:

$$2^m + 2K^{1/d}.$$

Here, the 2^m multiplications are needed to compute $F[k_1, \ldots, k_{m-1}, c]^2$ from $F[k_1, \ldots, k_{m-1}, c]$ for all 2^m relevant values of $(k_1, \ldots, k_{m-1}, c)$.

The above description refers to implementing the Booleanity-checking sum-check for a single i. That sum-check in fact must be applied d times (once for each $i = 1, \ldots, d$). However, a full d-fold cost increase does not quite occur, since several arrays can be reused between for all $i = 1, \ldots, d$. This holds in particular for arrays B, D, and F.

In summary, across the first $\log(K)/d$ rounds of the sum-check protocol, the prover's costs are as follows.

- $K^{1/d}$ field multiplications to build the array B.

- T multiplications to build D.
- $K^{1/d}$ multiplications to bind the array B across all rounds.
- $K^{1/d}$ multiplications to build up the array F across all $\log(K)/d$ rounds.
- $K^{1/d}$ multiplications to $F[k_1, \ldots, k_{m-1}, c]^2$ for all relevant values of $(k_1, \ldots, k_{m-1}, c)$ across all rounds m.
- For each $i = 1, \ldots, d$, $2K^{1/d}\log(K^{1/d})$ field multiplications to evaluate $s_1(c), \ldots, s_m(c)$ across all $\log(K)/d$ rounds for $c \in \{0, 2\}$ given the above arrays. Across all $i = 1, \ldots, d$, this is $2K^{1/d} \cdot \log(K)$ multiplications total.

Summing the above yields $T + (2\log(K) + 4) \cdot K^{1/d}$ field multiplications in total for the first $\log(K)/d$ rounds for all d Booleanity-checking sum-check invocations across all $i = 1, \ldots, d$.

Last $\log T$ rounds. The last $\log T$ rounds of sum-check are used to compute

$$\begin{aligned}
&\widetilde{\mathsf{eq}}(r, \mathsf{r}) \cdot \sum_{j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j) \left(\widetilde{\mathsf{ra}}_i(\mathsf{r}, j)^2 - \widetilde{\mathsf{ra}}_i(\mathsf{r}, j)\right) \\
&= \widetilde{\mathsf{eq}}(r, \mathsf{r}) \cdot \sum_{j \in \{0,1\}^{\log T}} D[j] \cdot \left(\widetilde{\mathsf{ra}}_i(\mathsf{r}, j)^2 - \widetilde{\mathsf{ra}}_i(\mathsf{r}, j)\right) \qquad (57)
\end{aligned}$$

Given the contents of the size-$K^{1/d}$ array F (Expression (55)) at the end of round $\log(K)/d$, the prover can, with no further multiplications, construct for each $i = 1, \ldots, d$, an array H_i of length T such that

$$H_i[j] = \widetilde{\mathsf{ra}}_i(\mathsf{r}, j).$$

Indeed by Equation (46), if k is the register read at cycle j then

$$\widetilde{\mathsf{ra}}_i(\mathsf{r}, j) = \widetilde{\mathsf{eq}}(k, \mathsf{r}),$$

and at the end of round $\log(K)/d$ this value is simply equal to $F[k]$. Then Expression (57) equals:

$$\widetilde{\mathsf{eq}}(r, \mathsf{r}) \cdot \sum_{j \in \{0,1\}^{\log T}} D[j] \cdot \left(H_i[j] \cdot H_i[j] - H_i[j]\right).$$

Given the arrays H_i and D, the standard linear-time sum-check proving algorithm (Section 3.3) implements the final $\log(T)$ rounds of the sum-check protocol (i.e., applies the sum-check protocol to compute Expression (57)) with the following costs, for each $i = 1, \ldots, d$.

- T field multiplications to bind the array D across all $\log T$ rounds.
- dT field multiplications to bind the arrays $H_1, \ldots H_d$ across all $\log T$ rounds.
- $2 \cdot 2 \cdot d \cdot T$ field multiplications suffice to complete the prover's work given the above arrays.

Hence, without further optimization, the total cost for the prover of the last $\log T$ rounds is $(5d + 1)T$ field multiplications. Added to the cost of the first $\log K$ rounds, this is $(5d + 2)T + O(K^{1/d}\log(K))$ field multiplications. However, further significant further optimizations are possible.

Further optimizations.

- So long as $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$ are committed before the core Shout PIOP (Figure 7) commences, one can set r' in Figure 8 to r_{cycle}, i.e., share verifier-chosen randomness between the core Shout PIOP and the one-hot-encoding-checking PIOP. This makes the array D (Expression (54)) built by the Booleanity-checking prover the same as the array E^* (Expression (44)) built by the core Shout PIOP prover. This eliminates T field multiplications from the cost of the Booleanity-checking prover (and thereby renders the number of prover field multiplications for the first $\log(K/d)$ rounds independent of T).

- The techniques of Dao and Thaler [DT24] directly apply to reduce the cost of binding the array $D = E^*$ to a low-order number of field multiplications (i.e., $O(\sqrt{T})$ of them). Ignoring low-order terms, this eliminates another T field multiplications from the prover's cost.
- Exactly as in the final $\log(T)$ rounds of the core Shout PIOP (Section 6.2.2), in round $\log(K)/d + m$ of sum-check, the number of distinct values in each array $H_1, \ldots, H_d$ is at most K^{2^m}, and each of these values is a sum of at most 2^m elements of a set of size $2^m \cdot K$ (and all quantities in this set can be computed in time $O(K2^m)$). This can be exploited to save most of the T multiplications devoted to binding each of the arrays $H_1, \ldots, H_d$. It also saves the work of squaring $H_1[c, x'], \ldots, H_d[c, x']$ for each $x' \in \{0,1\}^{\log(T)-t}$ in each sum-check round, for each evaluation point c. This is a total of $2dT$ multiplications saved in total.

This brings to the total number of prover multiplications for Booleanity-checking down from $(5d+2)T + O(K^{1/d} \log K)$ to essentially $3dT + O(K^{1/d} \log K)$.

$\widetilde{\mathsf{raf}}$-evaluation sum-check when $d = 1$ (Line 6 of Figure 6). This is a $\log(K)$-round sum-check protocol. The prover sends a degree-2 univariate polynomial in each round. It's easy to make the prover run in time $O(K)$ once given an E array of length K, that stores at cell $k' \in \{0,1\}^{\log(K)}$ the value $\widetilde{\mathsf{ra}}(k', r'_{\text{cycle}})$. Observe that

$$\widetilde{\mathsf{ra}}(k', r'_{\text{cycle}}) = \sum_{j \in \{0,1\}^{\log T} \,:\, \widetilde{\mathsf{ra}}(k', j) = 1} \widetilde{\mathsf{eq}}(j, r'_{\text{cycle}}). \tag{58}$$

Hence, if given an array D' of size T storing all values of them form $\widetilde{\mathsf{eq}}(j, r'_{\text{cycle}})$ as j ranges over $\{0,1\}^{\log T}$, the array E can be computed with no additional multiplications. D' can be computed with T field multiplications.

However, if $K = o(T)$, this procedure can be optimized further to avoid even the T multiplications needed to build D', using techniques similar to those in [DT24] that exploit multiplicative structure in the definition of $\widetilde{\mathsf{eq}}$ (Equation (15)). This is done by avoiding building the "full" size-T array D', whose sole purpose is to help initialize the array E. Instead, the prover builds a smaller array D'' of size $T/2^\ell$ whose definition is identical to D' but "leaves out" the last ℓ coordinates of j and r'_{cycle}. Specifically, for each $j' \in \{0,1\}^{\log(T)-\ell}$, letting $r = r'_{\text{cycle}}$,

$$D''[j'] \leftarrow \prod_{m=1}^{\log(T)-\ell} \widetilde{\mathsf{eq}}(r_m, j'_m).$$

The prover also builds an array D''' of size 2^ℓ the "incorporates" the last ℓ coordinates that were left out of the definition of D''. That is, for all $j'' \in \{0,1\}^i$,

$$D'''[j''] \leftarrow \prod_{m=1}^{i} \widetilde{\mathsf{eq}}(r_{\log(T)-m+\ell}, j''_\ell).$$

Then, for each register address $k' \in \{0,1\}^{K/d}$, the prover computes 2^ℓ quantities, one per $j'' \in \{0,1\}^\ell$, namely:

$$v_{k', j''} \coloneqq \sum_{j = (j', j'') \in \{0,1\}^{\log(T)-\ell} \times \{0,1\}^\ell \,:\, :\, \widetilde{\mathsf{ra}}(k', j) = 1} D''[j'].$$

Then recalling from Equation (58) that

$$D'[k'] = \sum_{j \in \{0,1\}^{\log T} \,:\, \widetilde{\mathsf{ra}}(k', j) = 1} \widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j),$$

it follows that

$$D'[k'] = \sum_{j'' \in \{0,1\}^\ell} D'''[j''] \cdot v_{k', j''}.$$

The cost of this protocol for building E_i is $T/2^\ell + 2^{\ell+1}$ field multiplications. Minimizing this quantity, the cost to compute the array E is $O(\sqrt{TK}) = o(T)$ multiplications.

Hamming-weight-one check for general d (Line 4 of Figure 8). This sum-check (for general d) is similar to the $\widetilde{\mathsf{raf}}$-evaluation sum-check for $d = 1$, but even simpler, as the prover only sends a degree-1 polynomial in each round. Given the arrays E_i computed for the $\widetilde{\mathsf{raf}}$-evaluation sum-check, the Hamming-weight-one sum-check prover can be implemented with $O(dK^{1/d})$ field multiplications if $dK^{1/d} = o(T)$.

$\widetilde{\mathsf{raf}}$-evaluation sum-check for $d > 1$ (Line 5 of Figure 8). Recall that here the sum-check protocol is applied to compute:

$$\sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r'_{\text{cycle}}, j)\left(\sum_{i=1}^{d}\sum_{\ell=0}^{\log(K)/d-1} 2^{i\cdot\log(K)/d+\ell}\cdot k_{i,\ell}\right)\cdot\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i, j).$$

This is similar to the read-checking sum-check in Shout, which was applied to compute:

$$\sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j)\left(\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i, j)\right)\cdot\widetilde{\mathsf{Val}}(k).$$

The only difference is that the $\widetilde{\mathsf{raf}}$-evaluation sum-check replaces $\widetilde{\mathsf{Val}}(k)$ with

$$\sum_{i=1}^{d}\sum_{\ell=0}^{\log(K)/d-1} 2^{i\cdot\log(K)/d+\ell}\cdot k_{i,\ell},$$

which is the MLE of the function that maps k to $\text{int}(k)$.

In applications, one can set the random value r_{cycle} chosen by the verifier before the read-checking PIOP to equal r'_{cycle}. In this case, the prover in the $\widetilde{\mathsf{raf}}$-evaluation sum-check can be implemented with only K field multiplications of additional work, by batching Shout's core read-checking sum-check instance with the $\widetilde{\mathsf{raf}}$-evaluation sum-check per Section 4.2.1. Effectively, at the start of both sum-checks, the prover simply replaces $\widetilde{\mathsf{Val}}(k)$ with $\widetilde{\mathsf{Val}}(k) + z\cdot\widetilde{\text{int}}(k)$, where z is randomness used in the standard batching technique of Section 4.2.1.

6.4 Cost summary for the combined Shout prover

Suppose $K = o(T)$. In this case, recall Section 6.2 gave an upper bound for the core Shout PIOP prover (Figure 5) of roughly the following number of field multiplications:

$$(d^2 + 2)T.$$

And Section 6.3 gave an upper bound for the one-hot-encoding-checking PIOP prover (Figure 8) of $O(K^{1/d}\log K) + 3dT$ when $K^{1/d}\log K = o(T)$. Summing the two, when $K = o(T)$ and $K^{1/d}\log K = o(T)$, the total prover work is at most

$$(d^2 + 3d + 2)\cdot T.$$

7 Fast Shout prover for large, structured memories

The sparse-dense sum-check protocol from (Generalized-)Lasso [STW24, Appendix G] is the key tool allowing the Shout prover to efficiently prove T lookups into (structured) memories of size vastly larger than T.

7.1 Sparse-dense sum-check protocol

7.1.1 Informal Overview

Consider two very large vectors $a, b \in \mathbb{F}^N$ and suppose we want to compute their inner product

$$\langle a, b \rangle = \sum_{i=1}^{N} a_i \cdot b_i. \tag{59}$$

Say we want to outsource this work to an untrusted prover using the sum-check protocol. A standard application would apply the sum-check protocol to the polynomial $\widetilde{a} \cdot \widetilde{b}$ where $\widetilde{a}$ and $\widetilde{b}$ are the multilinear extensions of a and b respectively. Note that $\widetilde{a} \cdot \widetilde{b}$ is a polynomial in $\log N$ variables with degree two in each variable. The standard linear-time prover implementation of the sum-check protocol in this context (Section 3.3) would require about $4N$ field operations. However, the prover can be implemented far faster if a and b are structured in certain ways. [STW24, Appendix G] refers to such a fast prover implementation as the *sparse-dense sum-check protocol*.[31]

Suppose we use two (not necessarily multilinear) extensions $\hat{a}$ and $\hat{b}$ of a and b. At the end of round 1, when the first variable gets bound to $r_1 \in \mathbb{F}$, both $\hat{a}$ and $\hat{b}$ become $(\log(N) - 1)$-variate polynomials $\hat{a}(r_1, x_2, \ldots, x_{\log N})$ and $\hat{b}(r_1, x_2, \ldots, x_{\log N})$. The sparse-dense sum-check protocol asks: how do these "new" polynomials relate to the "old" polynomials $\hat{a}(x_1, \ldots, x_{\log N})$ and $\hat{b}(x_1, \ldots, x_{\log N})$? In particular, is the relationship between $\hat{a}(x_1, \ldots, x_{\log N})$ and $\hat{a}(r_1, x_2, \ldots, x_{\log N})$ largely independent of $x_2, \ldots, x_{\log N}$, and similarly for $\hat{b}(x_1, \ldots, x_{\log N})$ and $\hat{b}(r_1, x_2, \ldots, x_{\log N})$? If so, then major optimizations are possible.

Let r_i denote the random field element chosen by the sum-check verifier in round i. Suppose that we can chunk the entries of a and b into two groups, $H(0)$ and $H(1)$, such that when variable one gets bound to r_1, each evaluation $\hat{a}(i)$ for $i \in H(0)$ gets multiplied by the same value (say m_{0,r_1}) and similarly each term a_i of a in $H(1)$ gets multiplied by m_{1,r_1}. Suppose further that each term b_i of b in $H(0)$ (respectively, $H(1)$) gets multiplied by the same value, say v_{0,r_1} and v_{1,r_1}.

Illustrative example. A key example to have in mind is

$$\hat{b}(x) = \widetilde{\mathsf{eq}}(r', x),$$

for some random vector $r' \in \mathbb{F}^{\log N}$. Then for any $x = (k, y)$ with $k \in \{0,1\}$ and $y \in \{0,1\}^{\log(N)-1}$,

$$\hat{b}(r_1, y)/\hat{b}(k, y) = \widetilde{\mathsf{eq}}(r'_1, r_1)/\widetilde{\mathsf{eq}}(r'_1, k).$$

In other words, changing the first coordinate fed to $\hat{b}$ from k to r_1 results in a multiplicative update that depends only on k, r_1, and r'_1.

Hence, letting H_0 and H_1 denote points in $\{0,1\}^{\log(N)}$ of the form $(0, y)$ and $(1, y)$ respectively, $\hat{b}$ satisfies the informal condition above if we define

$$v_{0,r_1} = \widetilde{\mathsf{eq}}(r'_1, r_1)/(1 - r'_1) \tag{60}$$

and

$$v_{1,r_1} = \widetilde{\mathsf{eq}}(r'_1, r_1)/r'_1. \tag{61}$$

Meanwhile, if $\hat{a}$ is multilinear (i.e., $\hat{a} = \hat{a}$) then by Fact 3.1,

$$\hat{a}(r_1, y) = (1 - r_1)\hat{a}(0, y) + r_1\hat{a}(1, y). \tag{62}$$

So in this example, we have the following expression for the prescribed prover's message in round two of sum-check, where r_1 is the random challenge in round 1:

[31] To clarify, the term "sparse-dense sum-check protocol" used in [STW24, Appendix G] refers to a fast *prover implementation*. The protocol itself (i.e., the checks the verifier does on the prover's messages) is no different than in the standard sum-check protocol.

$$
\begin{aligned}
s_1(r_1) &= \sum_{y \in \{0,1\}^{\log(N)-1}} \widetilde{a}(r_1, y) \cdot \hat{b}(r_1, y) \\
&= \left(\sum_{(0,y)\colon\, y \in \{0,1\}^{\log(N)-1}} (1 - r_1) \hat{a}(0, y) \cdot \hat{b}(r_1, y) \right) + \left(\sum_{(1,y)\colon\, y \in \{0,1\}^{\log(N)-1}} r_1 \hat{a}(1, y) \cdot \hat{b}(r_1, y) \right) \\
&= \left(\sum_{i \in H(0)} a_i \cdot b_i \cdot (1 - r_1) \cdot v_{0,r_1} \right) + \left(\sum_{i \in H(1)} (1 - r_1) \cdot a_i \cdot b_i \cdot r_1 \cdot v_{1,r_1} \right),
\end{aligned}
$$

where v_{0,r_1} and v_{1,r_1} are defined in Equations (60) and (61).

This means that in moving from the sum defining the prover's claim at the start of round 1 (Expression (59)) to the sum defining the prover's claim at the start of round 2, every term $a_i b_i$ for an $i \in H(0)$ gets multiplied by the same factor, namely $(1 - r_1) \cdot v_{0,r_1}$, and similarly for every term for an $i \in H(1)$. Rather than performing this multiplication for each term individually, it is much faster to first *aggregate* (i.e., sum up) all the terms $a_i \cdot b_i$ in $H(0)$ and $H(1)$ respectively, and just multiply each aggregated value by the relevant factor. This costs only two multiplications *in total across all terms*, rather than one for each of the N terms.

This is the key idea in the sparse-dense sum-check protocol from [STW24, Appendix G]. In round i, there are 2^i relevant groups (one group for each $k \in \{0,1\}^i$, with the k'th group capturing all terms whose binary representations begin with k). At the end of round i, when variable i gets round to $r_i \in \mathbb{F}$, each term within a group gets multiplied by the same value.

If N is considered too large for linear-in-N prover time to be acceptable, say, because only $T \ll N$ terms $a_i \cdot b_i$ are non-zero, then the sparse-dense sum-check protocol is capable of running in time $O(CT)$ where C is any integer such that $N \leq T^C$.

The sparse-dense sum-check protocol also applies when binding the j'th variable to r_i causes each $\hat{b}$'s value to change by an additive rather than multiplicative factor that depends only on r_j. This is explained in detail in [STW24, Appendix G]. The canonical example to have in mind here is

$$
\hat{b}(x) = \sum_{i=1}^{\log N} 2^{i-1} x_i,
$$

which arises when applying Generalized-Lasso or Shout to a "range-check" table storing all field elements in $\{0, 1, \ldots, N-1\}$. In this case, changing x_i from c to r_i increases $\hat{b}(x)$ by an additive factor of $2^{i-1}(r_i - c)$.

7.2 Extension to the Booleanity-checking sum-check when $K^{1/d} \gg T$

Overview. Recall that in the Booleanity-checking sum-check (Line 3 of Figure 8), the sum-check protocol is applied to compute

$$
\sum_{k \in \{0,1\}^{\log(K)/d},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, k) \widetilde{\mathsf{eq}}(r', j) \left(\widetilde{\mathsf{ra}}_i(k, j)^2 - \widetilde{\mathsf{ra}}_i(k, j) \right).
$$

Let us first focus on the "sub-sum"

$$
\sum_{k \in \{0,1\}^{\log(K)/d},\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, k) \widetilde{\mathsf{eq}}(r', j) \widetilde{\mathsf{ra}}_i(k, j)^2,
$$

as this is the heart of the protocol.

Let

$$
\hat{a}(k, j) = \widetilde{\mathsf{ra}}_i(k, j)^2
$$

and

$$\hat{b}(k,j) = \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j).$$

The treatment of the sparse-dense sum-check protocol in [STW24, Appendix G] assumes $\hat{a}$ is multilinear, i.e., $\hat{a} = \widetilde{a}$. This is not the case here, as $\hat{a}$ is the *square* of a multilinear polynomial.

Fortunately, $\widetilde{\mathsf{ra}}_i(k,j)$ satisfies an "ultrastructured" sparseness property: for each $j \in \{0,1\}^{\log T}$, there is at most (in fact, exactly) one $k \in \{0,1\}^{\log(K)/d}$ such that $\widetilde{\mathsf{ra}}_i(k,j) \neq 0$. Exploiting this property ensures that the sparse-dense sum-check protocol still applies to the Booleanity-checking sum-check. At a high level, what's going on is the following.

The treatment in [STW24, Appendix G] exploited that when $\hat{a}(k,j)$ is multilinear, Equation (62) holds, which roughly states that binding the first variable of $\hat{a}$ to r_1 causes the resulting evaluations of $\hat{a}(r_1,y)$ to be a linear combination of $\hat{a}(0,y)$ and $\hat{a}(1,y)$. When $\hat{a}(k,j) = \widetilde{\mathsf{ra}}_i(k,j)^2$, Equation (62) does not hold, but something just as nice does: if k is the unique value in $\{0,1\}^{\log(K)/d}$ with $\widetilde{\mathsf{ra}}_i(k,j) \neq 0$, then assuming wlog that the first entry of k is 0, and letting y denote the remaining $\log(K)/d + \log(T) - 1$ entries of (k,j), it holds that:

$$\widetilde{\mathsf{ra}}_i^2(r_1,y) = ((1-r_1)\widetilde{\mathsf{ra}}_i(0,y) + r_1\widetilde{\mathsf{ra}}_i(1,y))^2 = (1-r_1)^2\widetilde{\mathsf{ra}}_i(0,y)^2 = (1-r_1)^2\hat{a}(0,y).$$

In other words, binding variable one to r_1 causes $\hat{a}(k,j)$ to get multiplied by a factor $(1-r_1)^2$ that only depends on the first coordinate of k and on r_1.

Algorithm details. As in the overview above, let

$$\hat{a}(k,j) = \widetilde{\mathsf{ra}}_i(k,j)^2$$

and

$$\hat{b}(k,j) = \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j)$$

and suppose we apply the sum-check protocol to compute

$$\sum_{k \in \{0,1\}^{\log(K)/d},\ j \in \{0,1\}^{\log T}} \hat{a}(k,j) \cdot \hat{b}(k,j). \tag{63}$$

Let $c \geq 1$ be a positive integer such that $cK^{1/(cd)} = o(T)$. For simplicity, assume $\log(K)/d$ is divisible by c. We will break each $k \in \{0,1\}^{\log(K)/d}$ into c chunks each of size $\log(K)/(cd)$, writing

$$k = (k_1, \ldots, k_c) \in \left(\{0,1\}^{\log(K)/(dc)}\right)^c.$$

Similarly, write

$$r = (r_1, \ldots, r_c) \in \left(\mathbb{F}^{\log(K)/(dc)}\right)^c.$$

Evaluating all $\hat{b}(k,j)$ values. The first thing the prover needs to do is compute $\hat{b}(k,j)$ for all $(k,j) \in \{0,1\}^{\log(K)/d} \times \{0,1\}^{\log T}$ such that $\hat{a}(k,j) \neq 0$. In Booleanity-checking where $\hat{b}(k,j) = \widetilde{\mathsf{eq}}(k,r) \cdot \widetilde{\mathsf{eq}}(j,r')$, this can be done with

$$(c+1) \cdot K^{1/(cd)} + T$$

field multiplications as follows. First, the prover computes $c+1$ tables, each of size $K^{1/(dc)}$, storing for each $\ell \in \{1,\ldots,c\}$, the values $\widetilde{\mathsf{eq}}(r_\ell, x)$ as x ranges over $\{0,1\}^{\log(K)/(cd)}$, as well as a table storing $\widetilde{\mathsf{eq}}(r',x)$ as x ranges over the same domain $\{0,1\}^{\log T}$. This requires $c \cdot K^{1/(dc)} + T$ multiplications in total (by Lemma 1). Given these tables, for any desired $(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$, $\hat{b}(k,j)$ can be evaluated with one lookup into each of the $c+1$ tables followed by c multiplications.

Building a partial sum tree. Before the start of the protocol, the prover builds a "tree of partial sums". Specifically, for each $k_1 \in \{0,1\}^{\log(K)/(cd)}$, let

$$L(k_1) = \sum_{k=(k_1,k') \in \{0,1\}^{\log(K)/(cd)} \times \{0,1\}^{\log(K)/d - \log(K)/(cd)},\ j \in \{0,1\}^{\log T}} \hat{a}(k,j) \cdot \hat{b}(k,j).$$

The prover builds a binary tree over these leaf values, with each interior node storing the sum of its children. Note that the tree has $2^{\log(K)/(cd)} = K^{1/(cd)} = o(T)$ leaves. This step requires $(c+1)T$ field multiplications in total, because there are T non-zero entries of $\hat{a}(k,j)$ and for each such entry, $\hat{b}(k,j)$ can be computed with c multiplications and then multiplied by $\hat{a}(k,j)$ with a single additional multiplication.

Let us index the internal nodes at distance i from the root by $\{0,1\}^i$ (we will label the root by $\emptyset$). This ensures that the root of the tree, $L(\emptyset)$ computes the entire sum in Expression (63).

Round one message. The round-1 message of the prover can computed via a constant number of field operations (and inversions) given the values stored at the two children of the root. Indeed,

$$s_1(0) = \sum_{k=(0,k') \in \{0,1\} \times \{0,1\}^{\log(K)/d-1},\ j \in \{0,1\}^{\log T}} \hat{a}(k,j) \cdot \hat{b}(k,j),$$

and this quantity is equal to $L(0)$ (the left child of the root). Similarly, $s_1(2)$ is a constant-time-computable linear combination of $L(0)$ and $L(1)$:

$$s_1(2) = \left(\widetilde{\mathsf{eq}}(2,0)^2 \cdot \widetilde{\mathsf{eq}}(2,r_1) \cdot (1-r_1)^{-1}\right) \cdot L(0) + \left(\widetilde{\mathsf{eq}}(2,1)^2 \widetilde{\mathsf{eq}}(2,r_1) \cdot r_1^{-1}\right) \cdot L(1).$$

This equality holds by the following reasoning. For each $k = (0,k')$ and each $j \in \{0,1\}^{\log T}$ satisfying $\widetilde{\mathsf{ra}}_i(k,j) \neq 0$, it holds that $\widetilde{\mathsf{ra}}_i(k,j) = 1$ and $\widetilde{\mathsf{ra}}_i(2,k',j) = \widetilde{\mathsf{eq}}(2,0)^2$. Meanwhile,

$$\hat{b}(2,k',j) = \widetilde{\mathsf{eq}}((2,k'),r) \cdot \widetilde{\mathsf{eq}}(r',j) = \widetilde{\mathsf{eq}}(2,r_1) \cdot (1-r_1)^{-1} \cdot \hat{b}(k,j).$$

Similarly, for each $k = (1,k')$ and each $j \in \{0,1\}^{\log T}$ satisfying $\widetilde{\mathsf{ra}}_i(k,j) \neq 0$, it holds that $\widetilde{\mathsf{ra}}_i(k,j) = 1$ and $\widetilde{\mathsf{ra}}_i(2,k',j) = \widetilde{\mathsf{eq}}(2,1)^2$, and

$$\hat{b}(2,k',j) = \widetilde{\mathsf{eq}}(2,k',r) \cdot \widetilde{\mathsf{eq}}(r',j) = \widetilde{\mathsf{eq}}(2,r_1) \cdot r_1^{-1} \cdot \hat{b}(k,j).$$

(Even though each round's sum-check prover message s_i is a degree-3 univariate polynomial, Gruen [Gru24, Section 3] shows how a slight modification of the protocol allows the prover to only send $s_i(0)$ and $s_i(2)$ in each round, see Section 3.6.1).

Round two message. At the end of round 1, the verifier sends the random value r_1. Similar to round 1, the prover's round-2 message is a quickly-computable linear combination of the four values stored at level two of the tree (grandchildren of the root). Indeed,

$$s_2(0) = \sum_{k' \in \{0,1\}^{\log(K)/d-2},\ j \in \{0,1\}^{\log T}} \hat{a}(\mathsf{r}_1,0,k',j) \cdot \hat{b}(\mathsf{r}_1,0,k',j).$$

Recall that for each $j \in \{0,1\}^{\log T}$, the prover can quickly ascertain the unique $k'' \in \{0,1\}^{\log(T)-1}$ for which $\hat{a}(\mathsf{r}_1,k'',j) \neq 0$, and quickly compute the factor $m_{k_1'}$ (which depends only only r_1 and k_1') such that

$$\hat{a}(\mathsf{r}_1,k'',j) = m \cdot \hat{a}(k',j),$$

where $k' \in \{0,1\}^{\log K}$ is the unique vector such that $\hat{a}(k', j) \neq 0$. Similarly, we have assumed that the prover can quickly compute the factors v_0, v_1 (which depend on only r_1) such that $\hat{b}(\mathsf{r}_1, 0, k', j) = v_0 \cdot \hat{b}(0, 0, k', j)$ and $\hat{b}(\mathsf{r}_1, 0, k', j) = v_1 \cdot \hat{b}(0, 1, k', j)$. Hence,

$$s_2(0) = m_0 v_0 L(0,0) + m_0 v_1 L(0,1).$$

Similarly, $s_2(2)$ is a quickly-computable linear combination of the four nodes at distance two from the root: $L(0,0)$, $L(0,1)$, $L(1,0)$, $L(1,1)$:

$$s_2(2) = c_{0,0} \cdot L(0,0) + c_{0,1} L(0,1) + c_{1,0} L(1,0) + c_{1,1} L(1,1),$$

where for each $k_1, k_2 \in \{0,1\}$,

$$c_{k_1,k_2} = m_{k_1,k_2} v_{k_1,k_2},$$

where

$$m_{k_1,k_2} = \widetilde{\mathsf{eq}}(\mathsf{r}_1, k_1)^2 \cdot \widetilde{\mathsf{eq}}(2, k_2)^2$$

and

$$v_{k_1,k_2} = \widetilde{\mathsf{eq}}(k_1, r_1)^{-1} \cdot \widetilde{\mathsf{eq}}(\mathsf{r}_1, r_1) \cdot \widetilde{\mathsf{eq}}(2, r_2) \cdot \widetilde{\mathsf{eq}}(k_2, r_2)^{-1}.$$

Round $i \leq \log(K)/(cd)$. In general, $s_i(0)$ and $s_i(2)$ are each quickly-computable linear combinations of the nodes at distance i from the root. For example, when $\hat{a} = \widetilde{\mathsf{ra}}_i^2$ and $\hat{b}(k, j) = \widetilde{\mathsf{eq}}(r, k) \cdot \widetilde{\mathsf{eq}}(r', j)$, then:

$$s_2(2) = \sum_{k \in \{0,1\}^i} c_k \cdot L(k),$$

where $c_k = m_k \cdot v_k$, defined as follows:

$$m_k = \widetilde{\mathsf{eq}}(\mathsf{r}_1, \ldots, \mathsf{r}_{i-1}, 2, k)^2$$

and

$$v_k = \widetilde{\mathsf{eq}}(k, (r_1, \ldots, r_i))^{-1} \cdot \widetilde{\mathsf{eq}}(\mathsf{r}_1, \ldots, \mathsf{r}_i, r_1, \ldots, r_i).$$

Each round $i \leq \log(K)/(cd)$ can be implemented in roughly $O(2^i)$ field operations.[32]

Tree-recomputation procedure. The prover performs a tree-recomputation procedure at the end of round $\log(K)/(cd)$. Specifically, the prover computes, for every $j \in \{0,1\}^{\log T}$, the unique $k = (k_2, \ldots, k_c) \in \{0,1\}^{(c-1)\cdot\log(K)/(cd)}$ such that $\hat{a}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K)/c}, k, j) \neq 0$, and moreover for this k, the prover can compute both $\hat{a}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K)/c}, k, j)$ and $\hat{b}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K/c)}, k, j)$.

Indeed, by Lemma 1, with $K^{1/(cd)}$ multiplications the the prover can compute $\widetilde{\mathsf{eq}}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K)/(cd)}, k')$ for all $k' \in \{0,1\}^{\log(K)/(cd)}$. Since $\hat{a} = \widetilde{\mathsf{ra}}_i^2$, each quantity $\hat{a}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K)/(cd)}, k, j)$ can be identified with a single lookup into this table followed by a squaring operation. Similarly, for each non-zero $\hat{a}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K)/(cd)}, k, j)$, the value $\hat{b}(\mathsf{r}_1, \ldots, \mathsf{r}_{\log(K)/(cd)}, k, j)$ can also be computed with one multiplication given partial products computed and stored during the course of the procedure computing $\hat{b}(k, j)$ for all relevant values of (k, j) while building the partial sum tree prior to the start of round 1. This is $2T + K^{1/(cd)}$ field multiplications in total.

[32] More precisely, by using a batch-inversion procedure, it's possible to implement round i in with $O(2^i)$ field multiplications and one inversion.

Given these quantities, the prover can construct, with T more multiplications, a new partial-sum tree with T leaves, such that leaf k' stores

$$\sum_{k=(k_1,k',k_3,\ldots,k_c)\in\{0,1\}^{\log(K)/d},\ j\in\{0,1\}^{\log T}} \hat{a}(\mathsf{r}_1,\ldots,\mathsf{r}_{\log(K)/c},k',k_2,\ldots,k_c)\cdot\hat{b}(\mathsf{r}_1,\ldots,\mathsf{r}_{\log(K)/(cd)},k',k_2,\ldots,k_c).$$

The next $\log(K)/(cd)$ rounds proceed analogously to the first $\log(K)/(cd)$ rounds, followed by another tree-recomputation step, and so on until the first $\log(K)/d$ rounds are complete.

The final $\log T$ rounds can then be implemented exactly as in the case of small memory sizes (Section 6.3). The total cost for these final $\log T$ rounds is $5dT$ field operations (Section 6.3 implemented those rounds with only $3dT$ multiplications, but $2dT$ multiplications were saved there by exploiting that $K^{1/d} = o(T)$, which does not hold here if $c > 1$).

The full Booleanity-checking sum-check. Recall that the Booleanity-checking sum-check actually applies the sum-check protocol to compute

$$\sum_{k\in\{0,1\}^{\log(K)/d},\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j)\left(\widetilde{\mathsf{ra}}_i(k,j)^2-\widetilde{\mathsf{ra}}_i(k,j)\right).$$

We have explained above how to apply the sum-check protocol to compute

$$\sum_{k\in\{0,1\}^{\log(K)/d},\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j)\widetilde{\mathsf{ra}}_i(k,j)^2. \tag{64}$$

It suffices to now explain how to apply the sum-check prover to compute

$$\sum_{k\in\{0,1\}^{\log(K)/d},\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k)\widetilde{\mathsf{eq}}(r',j)\widetilde{\mathsf{ra}}_i(k,j), \tag{65}$$

since in the Booleanity-checking sum-check the prover's message in each round is just the difference between the corresponding message for Expressions (64) and (65).

Walking through the algorithm described above for proving Expression (64) and modifying it appropriately for Expression (65), it is easy to see that the sum-check prover for Expression (65) only needs to do $O(cK^{1/(cd)})$ additional field multiplications on top of what the prover for Expression (65) does. In other words, "handling" Expression (65) in addition to Expression (64) does not substantially change the prover cost.

Cost summary for Booleanity-checking when $K^{1/d} \gg T$. In total, across all $i = 1,\ldots,d$, the Booleanity-checking prover performs the following field multiplications:

- $c\cdot K^{1/(cd)} + T$ multiplications in pre-computation to evaluate $\widetilde{\mathsf{eq}}$ polynomials at appropriate points.
- $d(c+1)T$ more multiplications to build the d partial-sum trees prior to round one.
- $O(dK^{1/(cd)})$ to compute the first $\log(K)/(dc)$ rounds worth of messages given these trees.
- $3dT + O(dK^{1/(cd)})$ field multiplications every time the partial-sum tree is rebuilt and the next $\log(K)/(cd)$ rounds of messages are computed.
- $5dT$ field multiplications in the final $\log(T)$ rounds.

Since the partial sum tree for each $i = 1,\ldots,d$ is rebuilt $c-1$ times, this means the prover performs

$$O(dcK^{1/(dc)}) + (d(c+1) + 3d(c-1) + 5d + 1)T = O(dcK^{1/(dc)}) + (4dc + 3d + 1)\,T$$

field multiplications in total.

For example, if $c = 2$ and $d = 2$, this is $O(K^{1/4}) + 23T$ field multiplications. This is an appropriate setting of parameters when $K = 2^{64}$ as arises in Jolt (see Application 4 in Section 2.8), as in this case $K^{1/4} = 2^{16}$, which is substantially less than typical values of T.

7.3 Shout's read-checking sum-check

Recall that the read-checking sum-check within Shout (Line 3 in Figure 7) applies the sum-check protocol to confirm that $\widetilde{\mathsf{rv}}(r_{\text{cycle}})$ equals:

$$\sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}},j)\left(\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i,j)\right)\cdot\widetilde{\mathsf{Val}}(k). \tag{66}$$

[STW24, Appendix G] showed that the $\widetilde{\mathsf{Val}}$ polynomials corresponding to the lookup tables arising in the Jolt RISC-V zkVM satisfy the conditions necessary to apply the sparse-dense sum-check protocol to compute

$$\sum_{k\in\{0,1\}^{\log K}}\hat{a}(k)\cdot\widetilde{\mathsf{Val}}(k) \tag{67}$$

for any sparse, multilinear polynomial $\hat{a}$. It extends trivially to also handle the case that $\hat{a}$ takes an additional $\log T$ variables and the sum is over KT terms rather than K, i.e.,

$$\sum_{(k,j)\in\{0,1\}^{\log K}\times\{0,1\}^{\log T}}\hat{a}(k,j)\cdot\widetilde{\mathsf{Val}}(k). \tag{68}$$

To implement the read-checking sum-check prover quickly, we would like to apply the sparse-dense sum-check protocol to compute

$$\sum_{(k,j)\in\{0,1\}^{\log K}\times\{0,1\}^{\log T},\ k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d}\hat{a}(k,j)\cdot\hat{b}(k,j),$$

with $\hat{a}(k,j)=\widetilde{\mathsf{eq}}(r_{\text{cycle}},j)\cdot\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i,j)$ and $\hat{b}(k,j)=\widetilde{\mathsf{Val}}(k)$. However, this does not *quite* map into the formulation of Expression (68), because $\left(\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i,j)\right)$ does not have degree one in each variable of j (though it *does* have degree one in each variable of $k_1,\ldots,k_d$). Fortunately, this issue is not substantive. Since $\left(\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i,j)\right)$ is multilinear with respect to the variables of $k_1,\ldots,k_d$, we can still apply the sparse-dense sum-check protocol for the first $\log K$ rounds of the read-checking sum-check, and then "switch over" to the standard linear-time prover algorithm (Section 3.3) for the final $\log T$ rounds. If C is an integer such that $K^{1/C}=o(T)$, then the resulting total prover cost for the first $\log K$ rounds is, up to low-order terms, $2CT$ field multiplications.

The final $\log T$ rounds of Shout proceed identically to the case of small memories (Section 6.2), requiring d^2T multiplications up to low-order terms.

Remark 2. *[STW24, Appendix G] also describes two prover algorithms that are simpler but slower than the one that uses just $2dcT$ field multiplications to implement the first $\log K$ rounds above. These simpler algorithms replace $2dc$ with an $O(\log^2 K)$ and $O(\log K)$ factor respectively. Even these simpler algorithms may be fast enough for Shout to improve on Lasso as applied in Jolt.*

7.4 The $\widetilde{\mathsf{raf}}$-evaluation sum-check and Hamming-weight-one check when $K \gg T$

As pointed out in Section 6.3, the $\widetilde{\mathsf{raf}}$-evaluation sum-check of Figure 8 is identical to the read-checking sum-check in the core Shout PIOP, just with $\widetilde{\mathsf{Val}}(k)$ replaced with

$$\sum_{i=1}^{d}\sum_{\ell=0}^{\log(K)/d-1}2^{i\cdot\log(K)/d+\ell}\cdot k_{i,\ell},$$

which is the MLE of the function that maps k to $\text{int}(k)$. In other words, the $\widetilde{\mathsf{raf}}$-evaluation sum-check is equivalent to checking reads into the table whose k'th entry is $\int(k)$. As further observed in Section 6.3, in

applications the prover can set r_{cycle} in the core Shout PIOP equal to r'_{cycle} (the random point at which the verifier wishes to evaluate $\widetilde{\mathsf{raf}}$ via the $\widetilde{\mathsf{raf}}$-evaluation sum-check). Then running the $\widetilde{\mathsf{raf}}$-evaluation sum-check and the read-checking sum-check in a batched manner per Section 4.2.1 is equivalent to replacing the lookup table $\widetilde{\mathsf{Val}}(k)$ with $\widetilde{\mathsf{Val}}(k) + z \cdot \widetilde{\mathrm{int}}(k)$. The first $\log K$ rounds of the batched sum-check can be implemented by separately applying the read-checking sum-check to the two tables with MLEs given by $\widetilde{\mathsf{Val}}$ and $\widetilde{\mathrm{int}}$, with the latter costing $CT + o(T)$ multiplications for the prover. The final $\log T$ rounds of the batched protocol are no more expensive than the read-checking sum-check alone, as the value $\widetilde{\mathsf{Val}}(r''')$ in the read-checking sum-check (where r''' is the verifier's random challenges chosen over the course of the first $\log K$ rounds) is simply replaced in the batched sum-check with $\widetilde{\mathsf{Val}}(r''') + z \cdot \widetilde{\mathrm{int}}(r''')$.

Hamming weight one check. Checking that each committed address has Hamming weight one (Line 4 of Figure 8) can be directly implemented with $O(dK^{1/d})$ field multiplications after the prover computes an array storing $\widetilde{\mathsf{eq}}(j, r''_{\text{cycle}})$ for all $j \in \{0,1\}^{\log T}$ (which, for appropriate choice of r''_{cycle}, will have already been computed elsewhere within Shout). If $c > 1$ so $O(dK^{1/d})$ time is not acceptable, the original sparse-dense sum-check protocol applies with $\hat{a}(k) = \widetilde{\mathsf{ra}}_i(k, r''_{\text{cycle}})$ and $\hat{b} = 1$ and achieves cost $cT + O(CK^{1/C})$ field multiplications, where $C = cd$.

7.5 Cost summary for Shout when $K \gg T$

In summary, for gigantic structured memories where $K^{1/C} = o(T)$, the Shout prover incurs the following costs up to low-order terms:

- $2T$ multiplications for evaluating $\widetilde{\mathsf{rv}}(r_{\text{cycle}})$ before applying the read-checking sum-check.
- $\left(2C + d^2\right) T$ multiplications for the read-checking sum-check.
- $(4C + 3d)\, T$ for Booleanity checking.[33]
- CT for the $\widetilde{\mathsf{raf}}$-evaluation sum-check.
- cT for the Hamming-weight-one-checking sum-check if $c > 1$.

In total, this is at most $\left(7C + d^2 + 3d + c + 2\right) T$ multiplications.

8 Fast Twist prover implementation

Recall from Figure 9 that the core Twist PIOP consists of three sum-check invocations: the read-checking sum-check, the write-checking sum-check, and the $\widetilde{\mathsf{Val}}$-evaluation sum-check.

8.1 $\widetilde{\mathsf{Val}}$-evaluation sum-check

Let us start by considering the $\widetilde{\mathsf{Val}}$-evaluation sum-check, as it is the simplest. This invocation applies the sum-check protocol to compute:

$$\sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(r_{\text{address}}, j') \cdot \widetilde{\mathsf{LT}}(j', r_{\text{cycle}}).$$

We explain in Appendix B that, with $3T/2$ field multiplications, the prover can compute a table storing all evaluations of $\widetilde{\mathsf{LT}}(j', r_{\text{cycle}})$ as j' ranges over $\{0,1\}^{\log T}$. Similarly, (but more straightforwardly) with $2K$ field multiplications, the prover can compute a table storing all $\widetilde{\mathsf{Inc}}(r_{\text{address}}, j')$ evaluations. To do this, one first builds a size-K table storing all $\widetilde{\mathsf{eq}}(r_{\text{address}}, k)$ evaluations for $k \in \{0,1\}^{\log K}$, which requires K field

[33] Here, we avoid T multiplications by setting $r' = r_{\text{cycle}}$ within the Booleanity-checking sum-check, and thereby reusing a table of $\widetilde{\mathsf{eq}}$ evaluations between the first bullet and this bullet, same as in Section 6.3.

multiplications (see Lemma 1). Given this table, each $\widetilde{\mathsf{Inc}}(r_{\text{address}}, j')$ evaluation is simply the product of an increment with an entry of the size-K table.

Given the above two tables, the standard linear-time sum-check prover algorithm (Section 3.3) performs $4T$ additional field multiplications. Hence, total prover time in a straightforward implementation of the $\widetilde{\mathsf{Val}}$-evaluation sum-check is $2K + 5.5 \cdot T$ field multiplications. In Appendix B.2, we explain how to lower this to $2K + 4T$ field multiplications.

8.2 Read-checking and write-checking sum-checks

Achieving a fast prover in the read-checking and write-checking sum-checks requires new techniques. Let us begin by focusing on the read-checking sum-check, which in the case $d = 1$ confirms that $\widetilde{\mathsf{rv}}(r')$ equals:

$$\sum_{(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', j) \cdot \widetilde{\mathsf{ra}}(k, j) \cdot \widetilde{\mathsf{Val}}(k, j). \tag{69}$$

8.2.1 Overview

We give two different prover algorithms. One, which we present in Section 8.2.5, achieves $O(T \log(K) + K + d^2 T)$ prover time. The other, which we call our "local algorithm", has a worse dependence on d (though the same worst-case runtime when $d = 1$) but the following major advantage. Call a memory operation 2^i-*local* if it accesses for a memory cell that was previously accessed within the last 2^i cycles. For our local algorithm, any 2^i-local memory access only contributes $O(i)$ field operations to the prover's work. This is potentially much less than the worst-case bound of $O(\log K)$ field operations per memory operation. Due to the potentially poor concrete runtime when $d > 1$, we restrict our description of the local algorithm to the $d = 1$ case (though the generalization to $d > 1$ is straightforward).

The main difference between our two prover algorithms is as follows. In our local algorithm, we bind the $\log T$ variables of the cycle j first, starting with the "low-order" bit j_1 of j and proceeding towards the high-order bit $j_{\log T}$. Intuitively, this allows the prover to benefit from local memory accesses during the first $\log K$ rounds of sum-check because round i "coalesces" the values at any given register k across a contiguous chunk of 2^i time steps. So, if fewer than 2^i distinct registers were read or written during these 2^i time steps, then the prover does less than 2^i work to process those time steps in round i. This leads to less than T total prover work in round i, resulting in a prover time that is less than $T \log K$ across those $\log K$ rounds. After round $\log K$, there are only T terms to be summed, and the standard linear-time sum-check achieves $O(T)$ *total* time for those rounds (as opposed to $O(T)$ time *per round*).

The other algorithm binds the $\log(K)$ variables of the register address k first, rather than the $\log T$ variables of j.

8.2.2 Details of the local algorithm for read-checking

Consider the standard linear-time sum-check prover implementation (Section 3.3). Our local-algorithm prover can be viewed as simply running this standard procedure, but optimized for the specific structure of this sum-check instance. Recall that for the local algorithm, we restrict our attention to the case that $d = 1$.

Tracking increments instead of values. Naively implemented, the standard algorithm requires $K \cdot T$ space, with the bottleneck being storing all $K \cdot T$ evaluations of $\mathsf{Val}(k, j)$ for $k \in \{0,1\}^{\log K}$ and $j \in \{0,1\}^{\log T}$. However, we can avoid this by exploiting the same observation that leads to fast commitments: rather than explicitly storing all $\widetilde{\mathsf{Val}}(k, j)$ values for $(k, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$, the prover can instead just store, for each $j \in \{0,1\}^{\log T}$, the unique register k that was written at cycle j, and the non-zero increment $\mathsf{Inc}(k, j)$ (see Definition 9).

Under this definition, for all $k \in \{0,1\}^{\log K}$ and $j \in \{0,1\}^{\log T}$,

$$\widetilde{\mathsf{Val}}(k,j) = \sum_{j' < j} \widetilde{\mathsf{Inc}}(k,j'), \tag{70}$$

where $j' < j$ is shorthand for $\mathsf{LT}(j',j) = 1$, i.e., for j' being the binary representation of an integer $\mathrm{int}(j')$ that is strictly less than $\mathrm{int}(j)$. Note that Equation (70) is *not* an equality of formal polynomials, but rather an equality that holds pointwise at all $(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$. Nonetheless, Equation (70) implies that for any $r_1, \ldots, r_i \in \mathbb{F}$ and any $j \in \{0,1\}^{\log(T)-i}$,

$$\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(k, r_1, \ldots, r_i, j) \cdot \widetilde{\mathsf{LT}}(j', r_1, \ldots, r_i, j). \tag{71}$$

Indeed, the right hand side of Equation (71) is a multilinear polynomial in the $\log(K) + \log(T)$ variables $(k, r_1, \ldots, r_i, j)$, and by Equation (70) and the definition of LT, the right hand side and left hand side agree whenever $(k, r_1, \ldots, r_i, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$.

Simplifying Expression (71)**.** Break the T cycles $j \in \{0,1\}^{\log T}$ into $T/2^i$ "chunks" each of size 2^i. For each $j \in \{0,1\}^{\log(T)-i}$, chunk j refers to the collection of all cycles in $\{0,1\}^{\log T}$ whose last $\log(T) - i$ bits equal j, i.e., all cycles of the form $(\hat{j}, j)$ for some $\hat{j} \in \{0,1\}^i$.

We say that chunk $\bar{j}$ is strictly less than chunk j (denoted $\bar{j} < j$) if $\widetilde{\mathsf{LT}}(\bar{j}, j) = 0$, i.e., if $\bar{j}$ is the binary representation of an integer strictly less than the integer represented by j. Similarly, we say that chunk $\bar{j}$ is strictly greater than chunk j (denoted $\bar{j} > j$) if $\bar{j}$ is the binary representation of an integer strictly greater than j.

If $j', j'' \in \{0,1\}^{\log T}$ are respectively in chunks $\bar{j}$ and j such that $\bar{j} < j$, then $\widetilde{\mathsf{LT}}(j'', j') = 1$. Meanwhile, if $\bar{j} > j$ then $\widetilde{\mathsf{LT}}(j', j'') = 0$.

Let $j^* = (0^i, j)$ denote the first cycle in chunk j, where 0^i denotes the all-zeros vector of length i. Then a cycle $j' \in \{0,1\}^{\log T}$ is in a chunk $\bar{j} < j$ if and only if $j' < j^*$. Combining this with the preceding paragraph and Fact 3.1, we conclude that:

$$\begin{aligned}\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j) &= \left(\sum_{j' \in \{0,1\}^{\log T} :\, j' < j^*} \widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j) \right) + \sum_{\hat{j} \in \{0,1\}^i,\ j'' = (\hat{j}, j)} \widetilde{\mathsf{Inc}}(k, j'') \cdot \widetilde{\mathsf{LT}}(j'', r_1, \ldots, r_i, j) \\ &= \widetilde{\mathsf{Val}}(k, j^*) + \sum_{\hat{j} \in \{0,1\}^i,\ j'' = (\hat{j}, j)} \widetilde{\mathsf{Inc}}(k, j'') \cdot \widetilde{\mathsf{LT}}(j'', r_1, \ldots, r_i, j).\end{aligned}$$

It follows from the definition of $\widetilde{\mathsf{LT}}$ (Equation (87)) that for any $j'' = (\hat{j}, j)$, $\widetilde{\mathsf{LT}}(j'', r_1, \ldots, r_i, j) = \widetilde{\mathsf{LT}}(\hat{j}, r_1, \ldots, r_i)$. Hence, we conclude that

$$\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j) = \widetilde{\mathsf{Val}}(k, j^*) + \sum_{\hat{j} \in \{0,1\}^i :\, j'' = (\hat{j}, j)} \widetilde{\mathsf{Inc}}(k, j'') \cdot \widetilde{\mathsf{LT}}(\hat{j}, r_1, \ldots, r_i). \tag{72}$$

Ensuring the prover runs in $O(T)$ time per round. Since there is only one non-zero increment per cycle, the following fact follows easily from Fact 3.1.

Fact 8.1. *For any fixed $j'' \in \{0,1\}^{\log(T)-i}$, there at most 2^i registers $k \in \{0,1\}^{\log K}$ for which*

$$\widetilde{\mathsf{Inc}}(k, r_1, \ldots, r_i, j'') \neq 0.$$

In round i, for fixed $j' \in \{0,1\}^{\log(T)-(i-1)}$, let us refer to the vector

$$\left\{ \widetilde{\mathsf{Val}}(k, r_1, \ldots, r_{i-1}, j') \colon k \in \{0,1\}^{\log K} \right\}$$

as *row j' of $\widetilde{\mathsf{Val}}$ in round i*. The prover will proceed iteratively through the rows in lexicographic order, starting with row $0^{\log(T)-i}$ and proceeding to row $1^{\log(T)-i}$.

Per the above, the prover can maintain an array I that in round i stores the following values:

$$I \text{ in round } i \text{ stores all non-zero evaluations of the form } \widetilde{\mathsf{Inc}}(k, r_1, \ldots, r_i, j'') \text{ for } j'' \in \{0,1\}^{\log(T)-i}. \quad (73)$$

Given these increments, Equation (72) ensures that when processing row j', the prover can indeed store in its memory all K values $\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j')$ as k ranges over $\{0,1\}^{\log K}$. By Fact 8.1, updating the array I round over round requires at most T multiplications for each round $i = 1, \ldots, \log K$. For a given row j', the prover also has to computing the size-2^i sum in Expression (72) for each of the (at most) 2^i values of k for which $\widetilde{\mathsf{ra}}(k, r_1, \ldots, r_i, j') \neq 0$. This can be done with only 2^i multiplications per row, using the fact that for any $\hat{j} \in \{0,1\}^{i-1}$ and $b \in \{0,1\}$,

$$\widetilde{\mathsf{LT}}(\hat{j}, b, r_1, \ldots, r_i) = \widetilde{\mathsf{eq}}(b, r_i) \cdot \widetilde{\mathsf{LT}}(\hat{j}, r_1, \ldots, r_{i-1}) + (1-b) r_i.$$

In summary, for the first $\log(K)$ rounds of the read-checking sum-check, with $2T$ multiplications per round, the prover can ensure that in each round i, it can iterate over rows j' one-by-one and when processing row j' ensure that it has in its memory the K values

$$\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_{i-1}, j') \colon k \in \{0,1\}^{\log K}. \quad (74)$$

Computing other necessary arrays and worst-case cost accounting. Via standard techniques (see Lemma 1), with K field multiplications across all of the first $\log K$ rounds, the prover can maintain at each round i an array A storing all evaluations $\widetilde{\mathsf{eq}}(x, r_1, \ldots, r_i)$ as x ranges over $\{0,1\}^i$. Since only one register is read per cycle, at the end of round i, each row j'' of $\widetilde{\mathsf{ra}}(\cdot, r_1, \ldots, r_i, j'')$ has at most 2^i non-zero entries, and each such entry is an entry of this array (this holds because $\mathsf{ra}(k, j) \in \{0,1\}$ for all $(k, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}$, and is similar to Equation (46), which considered the case where m "register variables" rather than i "cycle variables" were bound to random field elements).

Similarly, with standard techniques, the prover can across all rounds i maintain an array storing all evaluations $\widetilde{\mathsf{eq}}(r', r_1, \ldots, r_i, j'')$ as j'' ranges over $\{0,1\}^{\log(T)-i}$. This costs $2T$ field multiplications in total (T to build the size-T array at the start of the protocol, and T more to bind it round-over-round over the course of the protocol). These are the arrays (along with I) needed to run the standard linear-time sum-check proving algorithm described in Section 3.3. However, a significant optimization is possible to avoid these $2T$ field multiplications, and more significantly, to also save one field multiplication per relevant evaluation point c for each of the first $\log K$ rounds.

An optimization. The optimizations of Dao and Thaler [DT24] can be adapted to speed up the prover further.

Specifically, assuming $K = o(T)$ so $\sqrt{KT} = o(T)$, the prover builds two different arrays of size at most $\sqrt{KT}$, defined as follows. Let $r'_L = (r'_1, \ldots, r'_{(\log(K)+\log(T))/2})$ and $r'_R = (r'_{(\log(K)+\log(T))/2+1}, \ldots, r'_{\log T})$. The first array E, at the end of round i, stores the following values:

$$E \text{ stores } \widetilde{\mathsf{eq}}(r'_L, r_1, \ldots, r_i, j'') \text{ as } j'' \text{ ranges over } \{0,1\}^{(\log(K)+\log(T))/2-i}. \quad (75)$$

Intuitively, $E(j'')$ captures the contribution of the "left part" j'' of a cycle index $j = (j'', j''') \in \{0,1\}^{\log(T)-i}$ to the value $\widetilde{\mathsf{eq}}(r', r_1, \ldots, r_i, j)$.

The second array E', in round i, stores the following values

$$E' \text{ stores } A(x) \cdot \widetilde{\mathsf{eq}}(r'_R, j''') \text{ as } x \text{ ranges over } \{0,1\}^i \text{ and } j''' \text{ ranges over } \{0,1\}^{\log(T)/2-\log(K)/2}. \quad (76)$$

This array captures the *combination* (i.e., product) of the contribution of j''' to $\mathsf{eq}(r', r_1, \ldots, r_i, j)$ and the value $\widetilde{\mathsf{ra}}(k, j)$ if $\widetilde{\mathsf{ra}}(k, r_1, \ldots, r_i, j) \neq 0$, namely $\widetilde{\mathsf{eq}}(r_1, \ldots, r_{i-1}, k_1, \ldots, k_{i-1})$.

For each index $j'' \in \{0,1\}^{(\log(K)+\log(T))/2-i}$, the prover maintains an "aggregated value" $\mathsf{agg}_{j''}$, initialized to 0. When the prover in round i wants to compute its prescribed message $s_i(c)$, it sums over all

$$(k, j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log(T)-i}$$

such that

$$\widetilde{\mathsf{ra}}(k, r_1, \ldots, r_{i-1}, c, j) \neq 0.$$

Recall that for each row $j \in \{0,1\}^{\log(T)-i}$, there are at most 2^i such values of k.

For each such pair (k, j), write $j = (j'', c, j''')$ with $j'' \in \{0,1\}^{(\log(K)+\log(T))/2-(i-1)}$ and $j''' \in \{0,1\}^{\log(T)/2-\log(K)/2}$. The prover uses a constant number of lookups into the arrays E' and I, followed by a single field multiplication, to compute

$$\widetilde{\mathsf{eq}}(r'_R, j''') \cdot \widetilde{\mathsf{eq}}(r_1, \ldots, r_{i-1}, c, k_1, \ldots, k_{i-1}, k_i) \cdot \widetilde{\mathsf{Val}}(k, r_1, \ldots, r_{i-1}, c, j),$$

and adds the result to $\mathsf{agg}_{j''}$. This ensures that when all relevant pairs (j, k) have been processed, the following holds:

$$s_i(c) = \sum_{j'' \in \{0,1\}^{(\log(K)+\log(T))/2-i}} \mathsf{agg}_{j''} \cdot \widetilde{\mathsf{eq}}(c, r'_i) \cdot E[j''].$$

Worst-case cost accounting. In total, maintaining the arrays E (Expression (75)) and E' (Expression (76)) across the first $\log K$ rounds requires $O(\sqrt{KT}) = o(T)$ field multiplications. Maintaining the array I (Expression (73)) costs at most $2^i \cdot (T/2^i) = T$ multiplications per round, as there are $T/2^i$ rows in round i and computing one row given the previous row requires at most 2^i multiplications. This yields a total of $T \log K$ multiplications to maintain the array I over the first $\log K$ rounds. Translating the values in the array I into necessary evaluations of $\widetilde{\mathsf{Val}}$ costs another T multiplications per round. We call these (at most) $2T \log K$ field multiplications *write-induced multiplications.*

Given these arrays, the prover in round $i = 1, \ldots, \log K$ performs at most $\left(T/2^i\right) \cdot 2^i + O(\sqrt{TK}) = T + o(T)$ field multiplications for each of the relevant evaluation points $s_i(c)$. Since $d = 1$, there are two evaluations points, namely $c \in \{0, 2\}$. This means there are $3\log(K)T + o(T)$ field multiplications, which we call *read-induced multiplications.*

In total, this means the first $\log K$ rounds of the sum-check protocol applied to Expression (69) incur at most the following number of field multiplications when $d = 1$:

$$4\log(K)T + o(T).$$

After round $\log K$, the number of terms being summed has fallen from KT down to T, and the standard linear-time sum-check proving algorithm (appropriately optimized [DT24, Gru24]) requires at most $6T$ total field multiplications to implement all subsequent rounds. In fact, when $K = o(T)$, this falls to roughly $5T$ due to binding of the E array taking much less than T field operations, as in each round i, there are two different algorithms the prover can choose between to bind E, one taking time 2^i, which leverages that at the start of the protocol E contains only values in $\{0, 1\}$, and the other taking time $T \cdot K/2^i$.

Hence, in total, for small memories the number of prover field multiplications when $d = 1$ is upper bounded by roughly:

$$(4\log(K) + 6)\, T + o(T).$$

8.2.3 A refined time bound for local memory accesses

The dominant term in the (worst-case) time bound for our main prover algorithm is $4\log(K)T$ field multiplications. This comes from performing 4 multiplications per cycle for each of the first $\log K$ rounds. One of these four multiplications is *write-induced*: it comes from updating the array I that stores $\widetilde{\mathsf{Inc}}$ evaluations: there are only $T/2^i$ "rows" of such evaluations stored in round i, but computing one row given the previous row can require 2^i field multiplications. This means that every write operation, in the worst case, contributes one field multiplication per round, for each of the first $\log K$ rounds.

The other (at most) three field multiplications per cycle per round are read-induced. In round $i+1$, one of these three comes from translating values in I into relevant evaluations of the form $\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j)$ for $j \in \{0,1\}^{\log(T)-i}$. This costs one multiplication per distinct register k that is actually read during cycles of the form (j'', j) with $j'' \in \{0,1\}^i$, and there are 2^i such registers in the worst case. The other two read-induced multiplications per cycle in round $i+1$ occur as follows. Each of the $T/2^i$ rows of E at the end of round i has at most 2^i distinct values, which means round $(i+1)$ incurs at most $2^i \cdot (T/2^i)$ field multiplications per evaluation point $c \in \{0,2\}$. This means that every read operation, in the worst case, contributes two field multiplications per round for the first $\log K$ rounds.

Refined cost analysis for write-induced operations. At the end of round i, updating the array I requires at most 2^i field multiplications. However, a tighter bound holds. For any $j' \in \{0,1\}^{\log(T)-i}$ consider cycles between $\text{int}(j') \cdot 2^i$ and $(\text{int}(j')+1) \cdot 2^i$. These are the cycles whose high-order $\log(T)-i$ bits equal j'. If at most Z distinct registers were written during these cycles, then in round i, computing row j' of I from the preceding row requires only Z field multiplications. In particular, if there were D duplicate writes during these cycles (meaning a write to a register that was already written during these cycles) then Z equals $2^i - D$.

Refined cost analysis for read-induced operations. Similarly, if there were D "duplicate reads" during these cycles, then the number of field multiplications in round i due to read operations performed during these cycles falls from a worst-case bound of $2 \cdot 2^i$ to $2 \cdot (2^i - D)$.

In summary, if a write operation is made to a register that was written 2^i cycles previously, then that write operation does not increase the number of distinct write operations in its chunk of write cycles in any round after round i. Hence, it only contributes i total multiplications to the prover's work across all rounds (vs. a worst-case bound of $\log K$ multiplications).

Similarly, if a read operation is made to a register that was read 2^i cycles previously, then that read operation only contributes $3i$ total multiplications to the prover's work across all rounds (vs. a worst-case bound of $3 \log K$ multiplications).

8.2.4 Write-checking sum-check via the local algorithm

Recall from Equation (34) that the write-checking sum-check is applied to compute

$$\sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k) \cdot \widetilde{\mathsf{eq}}(r',j) \cdot \left(\left(\prod_{i=1}^{d} \widetilde{\mathsf{wa}}_i(k_i,j)\right) \cdot \left(\widetilde{\mathsf{wv}}(j) - \widetilde{\mathsf{Val}}(k,j)\right)\right). \tag{77}$$

Let us focus on the case $d = 1$. We begin by considering a slightly simpler expression:

$$\sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,k) \cdot \widetilde{\mathsf{eq}}(r',j) \cdot \widetilde{\mathsf{wa}}(k,j) \cdot \widetilde{\mathsf{Val}}(k,j). \tag{78}$$

The sum-check invocation computing Expression (78) is almost identical to the one in the read-checking sum-check, with the main difference being the $\widetilde{\mathsf{eq}}(r,k)$ factor appearing in each term of the sum. If $K^2 = o(T)$, this factor can be handled with no major modifications to the read-checking sum-check nor any significant increase in prover cost, and if $K = o(T)$ incorporating this factor increases prover costs by at most T multiplications. Furthermore, since the sum-check for Expression (78) is run in parallel with the read-checking sum-check, the work to maintain the array I capturing evaluations of $\widetilde{\mathsf{Val}}$ can be reused within both the read-checking and writing-checking sum-check.

To apply the sum-check protocol to Expression (77) instead of the simpler Expression (78), simply run the prover for Expression (78), but every time the prover processes a value $\widetilde{\mathsf{Val}}(k, r_1, ..., r_i, j)$, replace this with $\widetilde{\mathsf{Val}}(k, r_1, \ldots, r_i, j) - \widetilde{\mathsf{wv}}(r_1, \ldots, r_i, j)$.

8.2.5 Alternative algorithm for read-checking and write-checking

We describe an alternative prover implementation for the read-checking sum-check in Twist that performs $O(T)$ field operations for each of the first $\log K$ rounds. Standard linear-time sum-check prover algorithms implement the final $\log T$ rounds with roughly d^2T field multiplications (see Section 6.2.2 for details).

Let us begin with the read-checking sum-check. Recall that for general $d > 1$, the read-checking sum-check within Twist is used to confirm that:

$$\widetilde{\mathsf{rv}}(r') = \sum_{k=(k_1,\ldots,k_d)\in\left(\{0,1\}^{\log(K)/d}\right)^d,\ j\in\{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r',j)\cdot\left(\prod_{i=1}^{d}\widetilde{\mathsf{ra}}_i(k_i,j)\right)\cdot\widetilde{\mathsf{Val}}(k,j).$$

We describe the algorithm as if it is sequential, but it is easy to see that a (T/K)-fold parallel speedup is achievable with at most a constant-factor increase in total prover work, by processing in parallel $\Theta(T/K)$ "chunks" of operations consisting of $\Theta(K)$ cycles in each.[34]

Our algorithm is essentially just the standard linear-time sum-check proving algorithm (Section 3.3), but optimized to leverage the specific structure of this particular sum-check instance. Specifically, before round one, the prover computes $d+2$ arrays, say B, C, and $A_1,\ldots,A_d$. B has size T, C has size $K\cdot T$, and and $A_1,\ldots,A_d$ have size $K^{1/d}\cdot T$. As we will see, the prover need not ever explicitly materialize C or $A_1,\ldots,A_d$ in their entirety, due to their structure in this particular sum-check instance. But for now let us consider a naive implementation of the standard algorithm, despite its unacceptably large runtime.

B initially stores all evaluations of $\widetilde{\mathsf{eq}}(r',j)$ as j ranges over $\{0,1\}^{\log T}$. B can be computed with T multiplications via standard techniques (Lemma 1). C initially stores $\widetilde{\mathsf{Val}}(k,j)$ for all $(k,j)\in\{0,1\}^{\log K}\times\{0,1\}^{\log T}$. And A_ℓ stores $\widetilde{\mathsf{ra}}_\ell(k_\ell,j)$ for all $k_\ell\in\{0,1\}^{\log(K)/d}$ and $j\in\{0,1\}^{\log T}$. By applying the binding procedure described in Section 3.3, the prover ensures that at the end of each round $\ell\le\log(K)$, C has size $(K\cdot T)/2^\ell$ and $C[k',j]=\widetilde{\mathsf{Val}}(r_1,\ldots,r_i,k',j)$ for all $(k',j)\in\{0,1\}^{\log(K)-i}\times\{0,1\}^{\log T}$. Similarly, at the end of each round $i\le\log(K)/d$, $A_1[k',j]=\widetilde{\mathsf{ra}}_1(r_1,\ldots,r_i,k',j)$ for all $k'\in\{0,1\}^{\log(K)/d-i}$ and $j\in\{0,1\}^{\log T}$, and similarly A_ℓ halves in size during each of rounds $(\ell-1)\cdot\log(K)/d+1,\ldots,\ell\log(K/d)$.

The above naive implementation of the standard linear-time algorithm is extremely wasteful because it ignores the following structure in the arrays C and $A_1,\ldots,A_d$.

- At the end of any given round i, for fixed $j\in\{0,1\}^{\log T}$, let us refer to all entries of C of the form $C[k',j]$ for some $k'\in\{0,1\}^{\log(K)-i}$ as the *j'th row of C*. Then any two adjacent rows differ in at most one entry. This is because at most one address gets written in each cycle.
- At the end of any given round i, any given row of A_ℓ has exactly one non-zero entry (see Equation (46) for details).

Avoiding spending $K^{1/d}\cdot T$ time and space processing A_ℓ. Per the second bullet, our prover will store, for each row of A_ℓ, merely the index k of the non-zero entry in that row, and the associated value of that entry. In fact, it's not even necessary to store that value: in round $(\ell-1)\cdot\log(K)/d+i$, there are only 2^i distinct values in the array A_i (see Equation (46) for what those values are), so the prover merely needs to store a lookup table of size 2^i storing those values and that lookup table can be shared amongst all the rows. The first i bits of the index k of the non-zero entry in a given row specify the appropriate index in the lookup table storing the associated non-zero value.

Avoiding spending KT time and space processing C. Per the first bullet, the prover will not explicitly store C. Rather, in each round $i+1$, the prover will iteratively generate each row of C and "process" that row's contributions to the prescribe round-i message.

[34] Bigger parallel speedups are possible at the cost of increased total prover work. Also, sharding described in Section 3.1.1 provides additional opportunities for parallelism beyond whatever parallelism is available when processing a single shard.

In more detail, via repeated application of Fact 3.1, one easily derives that for any $k' \in \{0,1\}^{\log(K)-i}$ and any $j \in \{0,1\}^{\log T}$,

$$C[k', j] = \sum_{k' \in \{0,1\}^i} \widetilde{\mathsf{Val}}(k'', k', j) \cdot \widetilde{\mathsf{eq}}(k'', r_1, \ldots, r_i). \tag{79}$$

Suppose that $k = (k'', k') \in \{0,1\}^i \times \{0,1\}^{\log(K)-i}$ is the (binary representation of) the address written at cycle k. Let $\mathsf{next}(j)$ be the binary representation of the "next" row, i.e., of $\mathrm{int}(j)+1$. Then by Equation (79), for any $k''' \neq k'$, $C[k''', j] = C[k''', \mathsf{next}(j)]$. That is, the only column that differs between rows j and $\mathsf{next}(j)$ is column k' (i.e., k' is the column whose trailing $\log(K) - i$ bits equal those of the binary representation of the register that was written at cycle j).

Moreover, recall from the Twist PIOP that the *increment* $\mathsf{Inc}(k, j)$ is the difference between the value stored at register k at cycle $j+1$ vs. cycle j. Then if $k = (k'', k')$ is the cell written at cycle j,

$$C[k', (j)] - C[k', j] = \mathsf{Inc}(k, j) \cdot \widetilde{\mathsf{eq}}(k'', r_1, \ldots, r_i). \tag{80}$$

And via standard techniques (see Lemma 1) the prover can, with K field operations across all of the first $\log K$ rounds, build an array that in round i stores all values of the form $\widetilde{\mathsf{eq}}(k'', r_1, \ldots, r_i)$ as k'' ranges over $\{0,1\}^i$.

Hence, putting all of the above together, the prover can, in each round $i = 1, \ldots, \log K$ of the read-checking sum-check, iterate one-by-one over all cycles $j \in \{0,1\}^{\log T}$, and ensure at all times that it has in its memory the j'th row of C, i.e., $C[k', j]$ as k' ranges over $\{0,1\}^{\log(K)-i+1}$. By Equation (80), the total number of field multiplications needed to accomplish this is T per round.

Total prover costs. To get through the first $\log K$ rounds of the read-checking sum-check, the prover incurs the following costs.

- T field multiplications to build the array B of $\widetilde{\mathsf{eq}}(r', j)$ evaluations.
- $O(dK^{1/d})$ multiplications to maintain the arrays $A_1, \ldots, A_d$.
- $T \log K + O(K)$ to maintain the array C throughout.
- If one directly applies the standard linear-time sum-check prover, then given the above arrays, the prover incurs 4 field multiplications per row $j \in \{0,1\}^{\log T}$ per round. Here, 4 comes from there being two relevant evaluation points $c \in \{0, 2\}$ in each of these rounds, and two multiplications needed for each of them (one to multiply an entry of C by an evaluation of the relevant array A_ℓ for that round, and another to multiply the result by the $\widetilde{\mathsf{eq}}(r', j)$ value for that row j).

  However, the optimization adapted from [DT24] that applied to the local algorithm also applies here. Roughly, this entails replacing the array B and the arrays $A_1, \ldots, A_d$ with $2d$ $\sqrt{KT}$-sized arrays (two per A_i). This eliminates the T multiplications needed to build B and one of the two multiplications per evaluation point c.

Hence, if $K = o(T)$, the first $\log K$ rounds of Twist's read-checking sum-check require $3\log(K)T + o(T)$ field multiplications of the prover. The last $\log T$ rounds require $(d^2 + 2d + 1)T$ field multiplications via the standard linear-time sum-check algorithm (Section 3.3), appropriately optimized [DT24, Gru24].

Write-checking sum-check. As with the local algorithm, implementing the write-checking sum-check prover is similar to the read-checking case. The situation is actually nicer in the case of the alternative algorithm, since for the alternative algorithm both the read-checking and write-checking sum-checks bind the $\log K$ variables of k before the $\log T$ variables of j. If $K = o(T)$, this ensures that the total number of prover multiplications incurred to implement the write-checking sum-check (on top of the work the prover does in the read-checking sum-check) is at most

$$(2\log(K) + d^2 + 2d + 2)T.$$

8.3 Cost summary

Local algorithm for $d = 1$. If $K = o(T)$, then using the local proving algorithm, the entire core Twist PIOP with $d = 1$ can be implemented with the following number of field multiplications, up to low-order terms:

- $4T$ from the $\widetilde{\mathsf{Val}}$-evaluation sum-check.
- $(4\log(K) + 6)T$ from the read-checking sum-check.
- An additional $(3\log(K) + 5)T$ from the write-checking sum-check.

This is at most $(7\log(K) + 15)\,T$ field operations to process T reads and T writes. For $K = 32 = 2^5$, this is $50T$ field multiplications. For $K = 2^{20}$, it is $155 \cdot T$.

However, the costs benefit significantly from locality. For large memories, the dominant term by far is the $7\log(K)T$ field operations, $4\log K$ of which can be attributed to each write and $3\log K$ to each read. If a write operation is 2^i-local, then $4\log K$ falls to $4i$ and similarly if a read operation is 2^i-local, then $3\log K$ falls to $3i$.

Checking correctness of one-hot encodings (Figure 8) for all $2T$ committed addresses must be done in addition to the above costs. Per Section 6.3, this costs $6dT + O(dK^{1/d})$ field multiplications (up to low-order terms, all of this extra cost is coming from Booleanity-checking).

The alternative algorithm. The alternative algorithm algorithm has similar (in fact, slightly better) costs in the worst-case to the local algorithm when $d = 1$, but generalizes more nicely to $d > 1$ because it binds the variables of k before the variables of j, and the degree in the variables of k is independent of d for all of the polynomials that the sum-check protocol is applied to within Twist. Specifically, the costs are:

- $4T$ from the $\widetilde{\mathsf{Val}}$-evaluation sum-check.
- $(3\log(K) + d^2 + 2d + 1)T$ from the read-checking sum-check.
- An additional $(2\log(K) + d^2 + 2d + 2)T$ from the write-checking sum-check.

In total, this is at most $(5\log(K) + 2d^2 + 4d + 4)T$ field multiplications. For $d = 1$, this is

$$(5\log(K) + 10)T,$$

slightly improving the worst-case bound of the local algorithm of $(5\log(K) + 19)\,T$.

9 Faster SNARKs for non-uniform computation

This section describes SpeedySpartan and Spartan++, fast SNARKs for non-uniform circuits. A distinguishing aspect of both is that they incur minimal commitment costs: the prover simply commits to its witness, and the rest of the prover's work is field operations in the sum-check protocol and the cost to prove a polynomial evaluation (which, for appropriate polynomial commitment schemes, can be sub-linear in the size of the circuit). This improves significantly on Spartan and BabySpartan (and other prior SNARKs) for non-uniform circuits.

9.1 Overview

In this overview, let us consider the simple and clean setting of proving satisfiability of an arithmetic circuit consisting of fan-in two addition and multiplication gates over field $\mathbb{F}$. This treatment extends easily to more general constraint systems.

Suppose that the circuit has size T and consider two $\log(T)$-variate multilinear polynomials $\widetilde{z}_A$ and $\widetilde{z}_B$ that respectively capture the left and right inputs to each gate of the circuit and a third polynomial $\widetilde{z}$ that captures the output of each gate. The first two polynomials, $\widetilde{z}_A$ and $\widetilde{z}_B$ are *virtual polynomials*: they are not explicitly committed/sent by the SpeedySpartan prover. The third polynomial $\widetilde{z}$ *is* explicitly committed/sent by the prover.

Overview of SpeedySpartan. SpeedySpartan applies the sum-check protocol once to confirm that $\widetilde{z}$ respects the operation of its gate. That is, if $j \in \{0,1\}^{\log(T)}$ is the label of a multiplication gate, then the sum-check confirms that $\widetilde{z}_C(j) = \widetilde{z}_A(j) \cdot \widetilde{z}_B(j)$, and similarly if j is an addition gate then the sum-check confirms that $\widetilde{z}_C(j) = \widetilde{z}_A(j) + \widetilde{z}_B(j)$.

At the end of this sum-check, the verifier has to evaluate each of $\widetilde{z}_A$, and $\widetilde{z}_B$ and $\widetilde{z}$ at a random point. Since $\widetilde{z}$ was explicitly committed by the prover, the evaluation of $\widetilde{z}$ can be obtained via the evaluation argument of the polynomial commitment scheme. Since $\widetilde{z}_A$ and $\widetilde{z}_B$ are virtual polynomials the verifier cannot obtain their evaluations directly. SpeedySpartan applies the sum-check protocol a second time to reduce the task of computing the necessary evaluations of $\widetilde{z}_A$ and $\widetilde{z}_B$ to the task of evaluating $\widetilde{z}$ at single point. In particular, this second sum-check invocation is simply the Shout read-checking sum-check (see Figure 7). That is, the left input $\widetilde{z}_A(j)$ to gate j is equal to the value *output* by some other gate j_L, and similarly the right input $\widetilde{z}_B(j)$ to j is the output of some gate j_R. This is the same as a *lookup* into the table whose entries are given by $\widetilde{z}$. The wires of the circuit (i.e., the identities of the left and right inputs to each gate) determine the addresses that are looked up. In the context of SpeedySpartan, these addresses depend only on the circuit wires and hence can be committed (as in Shout) by an honest party in pre-processing. This a complete overview of the SpeedySpartan protocol.

Overview of Spartan++. Recall that Spartan is a SNARK for R1CS and that SuperSpartan extends it to CCS [STW23], a generalization of Plonkish, R1CS, and AIR. Spartan++ is essentially just SuperSpartan [STW23], but with an improved sparse polynomial commitment scheme used to commit to "constraint matrices". Specifically, if the prover is claiming to know a solution vector z to an R1CS instance

$$Az \circ Bz = Cz,$$

(where, unlike in SpeedySpartan, A, B, and C, may have many non-zeros per row), then Spartan has the multilinear extension polynomials $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$ committed in pre-processing via Spark. In the online phase of Spartan, the sum-check protocol is applied several times, and at the end of these invocations, the Spartan verifier needs to evaluate $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$ at a random point r (in the case of CCS, the verifier needs evaluations of $\widetilde{M_0}, \ldots, \widetilde{M_{t-1}}$ for a chosen value of t). This evaluation in Spartan is provided by the prover along with a Spark evaluation proof. Spark uses the Lasso lookup argument internally, to perform lookups into a structured table storing all multilinear Lagrange basis polynomials evaluated at r. Spartan++ is the same, except we give a much faster sparse polynomial commitment scheme, based on Shout rather than Lasso. We call this scheme Spark++. Unlike Lasso, Shout allows the values returned by the to be virtual, which eliminates the primary bottleneck for the Spark prover (namely, committing to the values returned by the lookups; these values are random since r is random, and hence very slow to commit to).

9.2 Details of SpeedySpartan

9.2.1 Detailed protocol description

Our presentation below borrows text from BabySpartan [ST23]. Like BabySpartan [ST23], our focus here is on Plonkish constraint systems. For simplicity and easier adoption, we focus on degree-2 Plonkish, but the protocol readily applies to arbitrary degree Plonkish with custom gates. In particular, we use the definition from Plonk [GWC19, §6], which we state below.

Definition 9.1 (Plonkish)**.** *Consider a finite field $\mathbb{F}$. Let the public parameters consist of size bounds $m, n, \ell \in \mathbb{N}$, with $\ell < n$. The Plonkish structure consists of: (1) vectors $q_m, q_l, q_r, q_o, q_c \in \mathbb{F}^m$, and (2) a set of vectors $(a, b, c) \in [n]^m$ specifying the left, right, and output indices in a vector containing public IO and witness. An instance $x \in \mathbb{F}^\ell$ consists of public IO and is satisfied by a witness $w \in \mathbb{F}^{n-\ell}$ if the following holds for all $i \in [m]$. $(q_l)_i \cdot z_{a_i} + (q_r)_i \cdot z_{b_i} + (q_o)_i \cdot z_{c_i} + (q_m)_i \cdot (z_{a_i} \cdot z_{b_i}) + (q_c)_i = 0$, where $z = (w, x)$ and z_{a_i} denotes the entry of z at index provided by the ith entry of vector a.*

Note that Plonkish is a special case of CCS [STW23]. Observe that the vector z can be viewed as a lookup table and its entries are "looked up" at indices provided by vectors a, b, and c. In the language of CCS, we can represent these indices as three sparse matrices. Let $A, B, C \in \mathbb{F}^{m \times n}$ denote those matrices. In particular,

for all $i \in [m]$, the ith row of A (similarly B and C) is the unit vector that encodes the position specified by the ith entry of vector a (similarly b and c). In other words, each row of A, B, C contains a single non-zero entry of 1 so that the matrix-vector products Az, Bz, Cz "lookup" the correct entry of z. Thus, we can express the Plonkish satisfiability check as follows:

$$q_l \circ Az + q_r \circ Bz + q_o \circ Cz + q_m \circ Az \circ Bz + q_c = 0$$

SpeedySpartan can use any multilinear polynomial commitment scheme. In a preprocessing phase, the verifier (or other honest party) commits to polynomials that encode the circuit structure (i.e., sparse matrices A, B, C) using a multilinear polynomial commitment scheme. In the online phase, the prover begins by committing to its purported witness $\widetilde{w}$. The prover then reduces the circuit satisfiability check to claims about evaluations of $\widetilde{w}$ and polynomials committed in the preprocessing phase. Finally, the prover uses the polynomial evaluation argument of the polynomial commitment scheme to prove those claimed evaluations.

Figure 10 depicts the SpeedySpartan PIOP in full. Below, we describe via prose the resulting SNARK one obtains by combining the PIOP with a multilinear polynomial commitment scheme.

Preprocessing phase. In a preprocessing step, the verifier (or another trusted party) computes commitments to the multilinear extensions of the matrices $A, B, C \in \mathbb{F}^{m \times n}$. Each row in these matrices is a unit vector, i.e., each row contains a single entry equal to 1 (the rest are zeros). In other words, each matrix proves the one-hot encodings of m addresses into a table of size n.

For a chosen parameter $d \geq 2$, the commitment to each sparse matrix encodes m lookup addresses using d-dimensional one-hot encodings (Section 3.7). Since this commitment is created by the verifier (or another trusted party), there is no need to ensure, as part of Shout, that the underlying vectors are one-hot encodings. That is, Booleanity-checking and additional checks from Figure 8 are not needed. This is analogous to how in Spartan's preprocessing step [Set20], there is no need to prove the correctness of timestamps in its use of offline memory checking procedure.

Online phase. Without loss of generality, let m and n be powers of 2 and that $\ell = n/2$. Let $s = \log m, s' = \log n$. For a purported witness $w \in \mathbb{F}^{n-\ell}$ and public IO $x \in \mathbb{F}^{\ell}$, let $z = (w, x) \in \mathbb{F}^n$. We can view the vector z as a function with the signature $\{0,1\}^{s'} \to \mathbb{F}$. Similarly, we can view the sparse matrices A, B, C as functions with the following signature: $\{0,1\}^m \times \{0,1\}^n \to \mathbb{F}$. Crucially, in our target setting, each row of these sparse matrices is a unit vector (i.e., there is a single non-zero entry of 1 and the rest are zeros).

For all $x \in \{0,1\}^s$ and $M \in \{A, B, C\}$, let

$$\widetilde{z_M}(x) = \left(\sum_{y \in \{0,1\}^{s'}} \widetilde{M}(x, y) \cdot \widetilde{z}(y) \right)$$

Also, let

$$f(x) = (\widetilde{q_l}(x) \cdot \widetilde{z}_A(x) + \widetilde{q_r}(x) \cdot \widetilde{z}_B(x) + \widetilde{q_o}(x) \cdot \widetilde{z_C}(x) + \widetilde{q_m}(x) \cdot \widetilde{z}_A(x) \cdot \widetilde{z}_B(x) + \widetilde{q_c}(x)) \tag{82}$$

The prover begins by sending a commitment to its purported witness $\widetilde{w}$. To check satisfiability, the verifier selects a random $\tau \in \mathbb{F}^s$, and applies the sum-check protocol to the polynomial

$$g(x) = \tilde{\mathsf{eq}}(\tau, x) \cdot f(x),$$

using it to confirm that

$$0 = \sum_{x \in \{0,1\}^s} g(x).$$

If z satisfies the constraint system then this equality is guaranteed to hold, and if z does not satisfy the constraint system then this equality will fail to hold with probability at least $1 - O(s)/|\mathbb{F}|$.We call this the "Spartan-sum-check" since this sum-check invocation is exactly how Spartan begins [Set20].

Preprocessing phase. The input to this step includes sparse matrices $A, B, C \in \mathbb{F}^{m \times n}$, where each row of the sparse matrix is a unit vector, and vectors $q_l, q_r, q_o, q_m, q_c \in \mathbb{F}^m$. The preprocessing phase outputs a set of polynomials for which the verifier has a query access.

- Output $(\widetilde{q_l}, \widetilde{q_r}, \widetilde{q_o}, \widetilde{q_m}, \widetilde{q_c}, \widetilde{\mathsf{addr}}_A, \widetilde{\mathsf{addr}}_B, \widetilde{\mathsf{addr}}_C)$, where for $M \in \{A, B, C\}$, addr_M is the vector of size $m \cdot d \cdot n^{1/d}$ listing the d-dimensional one-hot encodings of addresses provided in M per Section 3.7 (i.e., viewing each row of M as the one-hot encoding of an address into a lookup table of size n). $\widetilde{\mathsf{addr}}_M$ is sent as d separate polynomials $\mathsf{addr}_{M,1}, \ldots, \mathsf{addr}_{M,d}$ each of size $m \cdot n^{1/d}$.

Online phase. The prover is given as input a purported witness vector $w \in \mathbb{F}^{n-\ell}$ and a public IO vector $x \in \mathbb{F}^\ell$. The verifier is given as input $x \in \mathbb{F}^\ell$.

1. $\mathcal{P} \to \mathcal{V}$: A purported witness polynomial $\widetilde{w}$.
2. $\mathcal{V} \to \mathcal{P}$: pick $\tau \in \mathbb{F}^s$ at random and send it to $\mathcal{P}$.
3. $\mathcal{V} \leftrightarrow \mathcal{P}$: Let f be defined as per Equation (82) (degree-2 Plonkish) or (83) (arithmetic circuits). $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to confirm that

$$0 = \sum_{x \in \{0,1\}^s} \mathsf{eq}(\tau, x) \cdot f(x).$$

   At the end of the sum-check protocol, the verifier is left with checking whether

$$c = \mathsf{eq}(\tau, r) \cdot f(r),$$

   where c and r are determined over the course of the sum-check protocol.
4. $\mathcal{P} \to \mathcal{V}$: Let $z = (w, x)$, and $z_A = A \cdot z$, $z_B = B \cdot z$ and $z_C = C \cdot z$. Send v_A, v_B, v_C defined as follows:

$$v_A \leftarrow \widetilde{z}_A(r),\ v_B \leftarrow \widetilde{z}_B(r),\ v_C \leftarrow \widetilde{z}_C(r).$$

5. $\mathcal{V}$: Let $e = \mathsf{eq}(\tau, r)$, and obtain the following values by querying polynomials committed during pre-processing: $v_l \leftarrow \widetilde{q_l}(r), v_r \leftarrow \widetilde{q_r}(r), v_o \leftarrow \widetilde{q_o}, v_c \leftarrow \widetilde{q_c}$. Reject if

$$c \neq e \cdot (v_l \cdot v_A + v_r \cdot v_B + v_o \cdot v_C + v_m \cdot v_A \cdot v_B + v_c).$$

6. $\mathcal{V} \leftrightarrow \mathcal{P}$: Apply the core Shout PIOP with addresses from the preprocessing phase $(\mathsf{addr}_A, \mathsf{addr}_B, \mathsf{addr}_C)$ to validate claims v_A, v_B, v_C. That is, for $M \in \{A, B, C\}$, apply the sum-check protocol to confirm that:

$$v_M = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, j) \left(\prod_{i=1}^{d} \widetilde{\mathsf{addr}}_{M,i}(k_i, j) \right) \cdot \widetilde{z}(k). \tag{81}$$

   This is equivalent to the Shout read-checking sum-check (Equation (30)) to check the claim that $v_M = \widetilde{z}_M(r)$ where $\widetilde{z}_M$ is the MLE of the purported read values (denoted $\widetilde{\mathsf{rv}}$ in Equation (30)), $\widetilde{z}$ is the MLE of the lookup table (denoted $\widetilde{\mathsf{Val}}$ in Equation (30)), and $\widetilde{\mathsf{addr}}_{M,1}, \ldots, \widetilde{\mathsf{addr}}_{M,d}$ together represent the d-dimensional one-hot encoding of the addresses (denoted $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$ in Equation (30)).

Figure 10: The description of SpeedySpartan PIOP. Let d denote the parameter chosen for Shout.

At the end of this invocation of the sum-check protocol, the verifier has to evaluate $\widetilde{z}_A(r)$, $\widetilde{z}_B(r)$ and $\widetilde{z_C}(r)$, where $r \in \mathbb{F}^s$ is a random point chosen over the course of the sum-check protocol. Hence, the invocation of the sum-check protocol reduced the task of checking that z satisfies the constraint system, to the task of evaluating the multilinear extensions of z_A, z_B, and z_C each at a random point $r \in \mathbb{F}^s$.

Recall that by design, each row of A, B, C has a single non-zero entry (i.e., viewing z as a lookup table, each row of A, B, and C is the one-hot encoding of some address of z). So, we invoke the Shout protocol to prove the desired evaluations of $\widetilde{z}_A, \widetilde{z}_B$, and $\widetilde{z_C}$ at r. In more detail, by using the Shout read-checking sum-check (Equation (30)), the prover reduces the claims about $\widetilde{z}_A(r), \widetilde{z}_B(r)$, and $\widetilde{z_C}(r)$ into evaluations claims about the lookup table $\widetilde{z}$ and polynomials committed in the preprocessing phase.[35] The prover then proves the evaluations of these committed polynomials using a polynomial evaluation argument.

Note that multiple invocations of the sum-check protocol and polynomial evaluation arguments, arising from running Shout for each of $\widetilde{z}_A, \widetilde{z}_B, \widetilde{z_C}$, can be batched together using standard techniques (see Sections 4.2.1 and 3.1.1 for details).

9.2.2 The special case of arithmetic circuit satisfiability

Arithmetic circuit satisfiability trivially reduces to the following special case of Plonkish:

$$f(x) = (1 - \widetilde{q}_m(x)) \cdot (\widetilde{z}_A(x) + \widetilde{z}_B(x)) + \widetilde{q}_m(x) \cdot \widetilde{z}_A(x) \cdot \widetilde{z}_B(x) \tag{83}$$

Here, $q_m(x) = 1$ if gate x is a multiplication gate, and equals 0 otherwise. For simplicity and ease of comparing to prior works, we focus on this special case when assessing the costs of SpeedySpartan

9.2.3 SpeedySpartan: Analysis of costs

Polynomial evaluation proofs and the choice of commitment scheme. SpeedySpartan's speedups relative to prior work are largest when polynomial evaluation proofs are not a major contributor to prover time. There are several reasons this could be the case. One is that SpeedySpartan can be instantiated with a commitment scheme that comes with very fast evaluation proofs. Another, discussed in Section 3.1.1, is that in applications where many different instances of (one or more) constraint systems arise, folding techniques can ensure only a single evaluation proof needs to be produced across all instances.

We now discuss attractive choices of polynomial commitment schemes that lead to fast SpeedySpartan prover even in settings that are not amenable to folding techniques.

Commitment schemes with fast evaluation proof computation. For fast evaluation proof computation, the most extreme choice is Hyrax [WTS+18]. In Hyrax, the prover simply performs a linear number of field multiplications, which are required anyway to compute the requested evaluation. Moreover, if the committed polynomial is sparse (as is the case in SpeedySpartan), the Hyrax prover's field work grows linearly with the sparsity of the polynomial (i.e., the number of non-zeros). More precisely, if M out of N entries of the committed vector are non-zero, the Hyrax prover's work during evaluation proof computation is $N + O(\sqrt{M})$ field multiplications.

This is a perfect fit to ensure a very fast SpeedySpartan prover. On top of this, Hyrax is transparent and has sublinear (square-root) size commitment key. The downside of Hyrax is that its commitment size and evaluation proof size are fairly large: they are each $\sqrt{N}$ field and group elements where N is the size of the committed polynomial. In summary, Hyrax is an attractive choice for SpeedySpartan when prover time is a top priority and sublinear-but-large verifier costs are acceptable.

Another commitment scheme with fast evaluation proof computation is Dory [Lee21]. Indeed, Dory can be seen as a refinement of both Hyrax and Bulletproofs/IPA [BBB+18, BCC+16], and inherits the fast evaluation proofs from Hyrax.[36] Dory evaluation proofs entail the same field work as in Hyrax evaluation proof computation, plus some extra cryptographic work that is asymptotically sublinear. This cryptographic work

[35] More precisely, if $z = (w, x)$ for a public vector x, then the prover commits to $\widetilde{w}$, and $\widetilde{z}(r)$ can be evaluated efficiently with one evaluation query to $\widetilde{w}$. See [Set20, STW23] for details.

[36] Essentially, the Dory committer uses pairings to *commit to a Hyrax commitment*, thereby obviating the $\sqrt{n}$ commitment size of Hyrax. For evaluation proofs, the Dory prover uses a protocol reminiscent of Bulletproofs/IPA to succinctly prove that it (a) knows a Hyrax commitment that opens the Dory commitment and (b) knows a valid Hyrax evaluation proof. It thereby obviates the $\sqrt{n}$ evaluation proof size of Hyrax.

consists of roughly $16\sqrt{N}$ scalar multiplications in $\mathbb{G}_1$ or $\mathbb{G}_2$ of a pairing-friendly group, and 6 multi-pairings of size $\sqrt{N}/2^i$ for each $i = 1, \ldots, \log N$.

Because pairings are concretely expensive (one pairing is equivalent in cost to about 4000 group operations), these evaluation proofs can be the concrete prover bottleneck (i.e., more expensive than committing or sum-check-proving in SpeedySpartan) unless N is not too small. This effect is somewhat amplified in SpeedySpartan, become some polynomials arising in the protocol (which are committed in pre-processing) are of slightly superlinear size $m \cdot n^{1/d}$. These polynomials have only m non-zero coefficients, so they do not require superlinear time to commit to even in pre-processing, but this does mean that the $O(\sqrt{N})$ cryptographic costs arising in Dory evaluation proof computation apply with $N = m \cdot n^{1/d}$. So while $O(\sqrt{N}) = O(\sqrt{m} \cdot n^{1/2d})$ is still sub-linear in m and n (and hence evaluation proof computation is asymptotically a low-order prover cost), it can be a concrete SpeedySpartan prover bottleneck if the circuit is small *and* the parameter d is small. If this is the case, per Section 3.1.1 one can always further lower the cost of Dory evaluation proof computation by a factor of $\log T$ with a constant-factor increase in verifier costs, by increasing the commitment size from one target group element to $\log T$ group elements.

Calculations with our analytic cost model (Section 3.1.2) show that Dory evaluation proofs represent a small fraction of total prover time (at most 30%) so long as m and n are at least 2^{26} and $d \geq 3$.

Challenges with HyperKZG or Bulletproofs. HyperKZG [ZSC24] and Bulletproofs/IPA [BCC+16, BBB+18] have expensive evaluation proof computation, as evaluation proofs require the prover to commit to a linear number of random field elements. As discussed above, this issue is amplified in the context of SpeedySpartan because some committed polynomials have size $m \cdot n^{1/d}$ (though they have sparsity only m). This means the commitment key size when using HyperKZG or Bulletproofs is slightly superlinear $m \cdot n^{1/d}$, and the evaluation proofs require committing to $O(m \cdot n^{1/d})$ random field elements. On top of that, unlike Hyrax and Dory, the field work required to compute evaluation proofs grows with the size $m \cdot n^{1/d}$ of the committed polynomial and not only its sparsity m. For these reasons, HyperKZG and Bulletproofs appear to be poor fits for SpeedySpartan.

Detailed costs for SpeedySpartan. As a running example throughout this section, we consider the case of $d = 2$ and Hyrax as the choice of polynomial commitment scheme. This leads to a very fast SpeedySpartan prover and achieves non-trivial verifier costs of $O(m^{1/2}n^{1/4})$. In this running example, we set $m = n = 2^{24}$ for concreteness. We also assume that the entries of the witness vector z are all "small" (see Section 2.1) and hence fast to commit to. This is indeed the case in many applications (where the entries of z often represent bit or byte decompositions, or 32-bit values).

Size of proving key. When Hyrax (or Dory) is used in SpeedySpartan, the (transparently generated) commitment key consists of $\sqrt{m \cdot n^{1/d}}$ group elements. For example, if $m = n = 2^{24}$ and $d = 2$, this is only 2^{18} group elements, translating to MBs of space.

Pre-processing. Pre-processing in SpeedySpartan requires committing to $3dm$ unit vectors each of length $n^{1/d}$, plus 5 vectors of length m. With Hyrax, committing to the unit vectors requires dm group operations, while committing to the other 5 vectors (assuming they have small entries per Section 2.1) requires roughly $5m$ group operations.

Online phase prover costs. In SpeedySpartan, during the online phase, the prover only commits to the witness vector z and nothing else (more precisely, the prover commits to the MLE of its purported witness vector w of length $n - \ell$). Thus, if the witness vector contains "small" field elements, then the prover only commits to small field elements.

Then the SpeedySpartan prover applies the sum-check protocol several times. Once is the Spartan sum-check, and the other is Shout's read-checking sum-check, applied to twice in parallel (once for each matrix A and B). The Spartan sum-check can be implemented with $9m$ field multiplications. This is the cost of the standard linear-time sum-check proving algorithm (Section 3.3) combined with optimizations of Dao and Thaler [DT24] and Gruen [Gru24] (see Section 3.6.1). As a modest additional optimization, we also exploit

	Plonk (small proof)	Plonk (fast prover)	BabySpartan	SpeedySpartan ($d = 2$)	SpeedySpartan ($d = 3$)
Committed field elements (excluding pre-processing)	$8m$ random $3m$ small	$6m$ random $3m$ small	$3m + n$ small	n small	n small
Field multiplications	$54m \log m$	$54m \log m$	$33m + 24n$	$19m + 8n$	$29m + 8n$
Total field multiplications (approximate)	$54m \log m + 546m$	$54m \log m + 414m$	$51m + 30n$	$19m + 14n$	$29m + 14n$
Total field multiplications (for $m = n = 2^{24}$)	$1842 \cdot 2^{20}$	$1710 \cdot 2^{24}$	$81 \cdot 2^{24}$	$33 \cdot 2^{24}$	$43 \cdot 2^{24}$

Figure 11: Prover costs for SpeedySpartan compared with two versions of Plonk [GWC19] and BabySpartan when applied to arithmetic circuits of fan-in two (n is the total witness size and m is the number of constraints (i.e., $n - m$ is the number of auxiliary, unconstrained inputs). For simplicity we assume $m = n$ when stating Plonk's costs. When translating committed field elements to field multiplications, we treat each small committed field element as costing one group operation, each random committed field element as costing 11 group operations, and each group operation as costing 6 field operations. See Section 3.1.2 for justification of these translations. The qualitative comparison between protocols is insensitive to these precise choices.

here that $q_m(x) \in \{0, 1\}$ for all $x \in \{0, 1\}^{\log m}$. This ensures that binding the array storing q_m evaluations produces only $2^{O(i)}$ distinct values in round i, which ensures that the prover only incurs $2^{O(i)}$ multiplications to bind the array in that round.

Per Theorem 6 with $T = m$, $m = n$ and $d = 3$, both invocations of the Shout read-checking sum-check costs $(d^2 + 2)m + 4n + o(m)$ field operations. Of these, m field multiplications are devoted to the prover evaluating $\widetilde{z}_A(r)$ and $\widetilde{z}_B(r)$, which is already done over the course of the Spartan sum-check and accounted for in the costs reported there. So the actual cost of each of these three invocations is $(d^2 + 1)m + 4n + o(m)$. For $d = 2$ and $m = n$, this simplifies to about $9m$ field multiplications per invocation of Shout, and hence $18m$ across both invocations.

In total, the SpeedySpartan prover performs $25m$ field multiplications across all invocations of the sum-check protocol.

The SpeedySpartan prover must also provide an evaluation for each committed polynomial. Per the discussion in Section 9.2.3, we focus on settings where this is not a significant contributor to prover time.

Verifier costs. The proof size is $O(d \log(n + m))$ field elements, plus the polynomial evaluation proof provided for each committed polynomial. If desired, standard techniques can batch these evaluation proofs at minimal cost to the prover, ensuring that only a single evaluation proof is provided and verified (see for example [GLH+24, Section 5]). The verifier's runtime is $O(d \log(n + m))$ field operations plus the cost of checking the polynomial evaluation proof(s).

9.2.4 Comparison of SpeedySpartan with Plonk and BabySpartan

Figure 11 reports quantitative prover costs for Plonk, BabySpartan, and SpeedySpartan. The reported costs apply when these protocols are combined with any elliptic-curve commitment scheme like Hyrax and Dory. The reported costs also apply to HyperKZG and Bulletproofs/IPA, but these commitment schemes are poor fits for SpeedySpartan due to high evaluation proof computation time and commitment key size (§9.2.3).

We use the reported costs from Plonk [GWC19]. We omit other natural baselines such as HyperPlonk [CBBZ23] as BabySpartan is strictly faster than these baselines. For all schemes, we ignore the cost of evaluation proof computation. This is justified for SpeedySpartan by the discussion above (i.e., we are focused either on choices of commitment schemes for which this is not a significant prover cost, or settings where this cost is insignificant for other reasons).

The comparison can be summarized as follows. With $d = 2$, SpeedySpartan improves over the field multiplications performed by the Plonk prover by roughly $50\times$, and improves the commitment costs by about $75\times$. For commitment costs, the large improvement comes both because SpeedySpartan commits to about $9\times$

fewer field elements than Plonk, and because all of the values committed by the SpeedySpartan prover are small, while 2/3 of the values committed by the Plonk prover are random (recall from Sections 2.1 and 3.1.2 that random values take an order of magnitude more time to commit to than small ones). SpeedySpartan with $d = 2$ improves over the commitment cost of BabySpartan by about $4\times$, and over the field work of BabySpartan by about $2\times$.

With $d = 3$, the field work of SpeedySpartan increases about 30% relative to $d = 2$, and the online commitment costs remain the same. The reason to consider higher values of d is that commitment key size, evaluation proof size, evaluation proof computation time (in the case of Dory), and pre-processing time all decrease as d grows.

9.2.5 Obtaining a folding scheme from SpeedySpartan

Like how an "early stopping" (Super)Spartan leads to HyperNova [KS24a], SpeedySpartan leads to a folding scheme for Plonkish with attractive characteristics: the folding scheme can fold multiple Plonkish instances defined with respect to Plonkish constraint systems that do *not* necessarily share the same circuit structure. Compared to a naive solution that achieves this property, a SpeedySpartan-based solution has far better costs. In fact, the costs are similar to HyperNova: the prover simply commits to its witness and incurs some field operations in the sum-check protocol. The verifier circuit size is logarithmic in the circuit sizes, which is similar to HyperNova's. We leave it to the future work to improve the size of the verifier circuit with ideas from NeutronNova to use an early-stopping sum-check protocol [KS24b].

9.3 Details of Spartan++

9.3.1 Spark++: A faster commitment scheme for sparse polynomials

Commitment phase. Let p be a T-sparse multilinear polynomial to be committed. This means p is a polynomial in ℓ variables, so p has size 2^ℓ,but there are only $T = o(2^\ell)$ values of $x \in \{0,1\}^\ell$ such that $p(x) \neq 0$. Let $S = \{x \in \{0,1\}^\ell : p(x) \neq 0\}$. For simplicity, let us assume that $p(x) \in \{0,1\}$ for all $x \in \{0,1\}^\ell$; this is sufficient to give a SNARK for arithmetic circuits, and the below protocol easily extends (with some increased costs) to eliminate this assumption.

Consider a memory of size $K = 2^\ell$. Associate each index of memory with an input $x \in \{0,1\}^\ell$. For parameter $d \geq 1$, the commitment to p is simple the d-dimensional one-hot encoding of each $x \in S$. That is, the commitment to p consists of d committed polynomials $\widetilde{\mathsf{ra}}_1, \ldots, \widetilde{\mathsf{ra}}_d$, where for each $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{ra}}_1(\cdot, j), \ldots, \widetilde{\mathsf{ra}}_d(\cdot, j)$ provides the d-dimensional one-hot encoding of the j'th element of S.

If the commitment were computed by an untrusted party, any verifier, before running the evaluation phase, would need to run the one-hot-encoding-checking PIOP (Figure 6) to ensure that the commitment to p is a valid list of d-dimensional one-hot encodings. However, in the context of Spartan++, the commitments to sparse polynomials p are all computed by an honest party and hence these checks can be omitted.

Evaluation phase. Suppose the verifier requests $p(r)$. By multilinear Lagrange interpolation and the assumption that $p(x) \in \{0,1\}$ for all $x \in \{0,1\}^\ell$, we can write:

$$p(r) = \sum_{k_1,\ldots,k_d \in \{0,1\}^{\ell/d}, j \in \{0,1\}^{\log T}} \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j) \right) \cdot \widetilde{\mathsf{eq}}(k, r).$$

In other words, if we consider the lookup table of size $K = 2^\ell$ whose k'th entry (for $k \in \{0,1\}^\ell$) is $\widetilde{\mathsf{eq}}(k, r)$, and we consider S to be a list of T addresses for lookups into this table, then $p(r)$ is simply

$$\sum_{j \in \{0,1\}^{\log T}} \widetilde{\mathsf{rv}}(j), \tag{84}$$

where $\widetilde{\mathsf{rv}}$ is the MLE of the values returned by the lookups.

Based on Equation (84), to prove the correct value of $p(r)$, the prover has two options. One is to apply the sum-check protocol to compute Expression (84), at the end of which the verifier has to evaluate $\widetilde{\mathsf{rv}}(r')$ for a random $r' \in \mathbb{F}^{\log T}$. The verifier can then accomplish this using Shout's read-checking sum-check applied to the table above. The other option is to observe that Expression (84) equals $T \cdot \widetilde{\mathsf{rv}}(2^{-1}, \ldots, 2^{-1})$ (this observation also arose surrounding Equation (26) in Section 4.1.2), and apply Shout's read-checking sum-check to compute this particular evaluation of $\widetilde{\mathsf{rv}}$.

Note that the table to which Shout is applied above is MLE-structured. Indeed, for this table $\widetilde{\mathsf{Val}}(k) = \widetilde{\mathsf{eq}}(k, r)$, which can clearly be evaluated at any point $k \in \mathbb{F}^\ell$ in $O(\log K) = O(\ell)$ time, via Equation (15).

Costs. Committing to the sparse polynomial p entails committing to d vectors, each with T 1s and $K^{1/d}$ 0s. The evaluation phase involves applying Shout to T lookups into a table of size $K = 2^\ell$. This is a canonical table to which the sparse-dense sum-check protocol applies (Section 7.1), enabling the Shout prover to run in time $O(CT)$ where C is such that $K \leq T^C$.

9.3.2 Details of Spartan++

Recall that Spartan++ is essentially just SuperSpartan, but with the Spark sparse polynomial commitment scheme replaced with Spark++. Here, we spell out the details. For ease of exposition, we focus on R1CS (a special case of CCS) and Spartan (a special case of SuperSpartan).

A special case of R1CS capturing arithmetic circuits. In Spartan++ the prover only commits to the values (i.e., outputs) w of multiplication gates. If there are M multiplication gates, then $w \in \mathbb{F}^M$, and the prover begins the protocol by committing to the multilinear extension $\widetilde{w}$.

Let $z = (w, x) \in \mathbb{F}^n$ where $x \in \mathbb{F}^{n-M}$ is public input. Ensuring that w indeed contains the correct value of every multiplication gate entails proving that z satisfies M rank-one constraints. Specifically, if a_i and b_i are the left and right inputs to multiplication gate i, then the i'th constraint is simply

$$a_i \cdot b_i = w_i.$$

Let us express this constraint system as $Az \circ Bz = w$ where A and B are appropriate matrices in $\mathbb{F}^{M \times n}$. Note that, unlike in SpeedySpartan, any particular row of A and B may have many non-zero entries (addition gates in the circuit lead to non-zero entries in A and B, but addition gates do not increase the number of rows). Further, notice that all of the non-zero entries of these matrices are equal to 1.

The Spartan++ protocol. In pre-processing, an honest party commits to $\widetilde{A}$ and $\widetilde{B}$ using Spark++.

In the online phase, as in Spartan, the prover first applies the standard zero-check PIOP (Section 3.6) to confirm that all constraints are satisfied. That is, letting $a = Az$ and $b = Bz$, the verifier picks a random $r \in \mathbb{F}^{\log M}$ and checks that

$$0 = \sum_{x \in \{0,1\}^{\log M}} \widetilde{\mathsf{eq}}(r, x) \cdot \left(\widetilde{a}(x) \cdot \widetilde{b}(x) - \widetilde{w}(x) \right). \tag{85}$$

At the end of the sum-check protocol, the verifier has to evaluate $\widetilde{a}(r')$, $\widetilde{b}(r')$, $\widetilde{w}(r')$ and $\widetilde{\mathsf{eq}}(r, r')$ where $r' \in \mathbb{F}^{\log M}$ is the randomness chosen round-over-round during the sum-check protocol. The evaluation $\widetilde{w}(r')$ can be obtained from the commitment to $\widetilde{w}(r')$, and $\widetilde{\mathsf{eq}}(r, r')$ can be computed in $O(\log M)$ time. But $\widetilde{a}$ and $\widetilde{b}$ are not committed—they are effectively virtual polynomials. Hence, $\widetilde{a}(r')$ and $\widetilde{b}(r')$ are computed with an additional application of the sum-check protocol, applied to compute the right hand sides of the following two expressions for $\widetilde{a}(r')$ and $\widetilde{b}(r')$:

$$\widetilde{a}(r') = \sum_{j \in \{0,1\}^{\log n}} \widetilde{A}(r', j) \cdot \widetilde{z}(j),$$

and

$$\widetilde{b}(r') = \sum_{j \in \{0,1\}^{\log n}} \widetilde{B}(r', j) \cdot \widetilde{z}(j).$$

At the end of these two instances of sum-check (which can be run in parallel and batched per Section 4.2.1), the verifier needs to evaluate $\widetilde{A}(r', r'')$, $\widetilde{B}(r', r'')$ and $\widetilde{z}(r'')$ where r'' denotes the randomness chosen over the course of the two parallel sum-check instances. The evaluation $\widetilde{z}(r'')$ can be obtained from the commitment to $\widetilde{z}$. The evaluations of $\widetilde{A}$ and $\widetilde{B}$ are obtained via Spark++.

Prover Costs. Assume for simplicity that A and B have the same number of non-zero entries and that this number is T (T is proportional to total number of gates in the circuit, counting both addition and multiplication gates).

Given the vectors $a = Az$ and $b = Bz$, the first sum-check (to compute Expression (85)) costs $5M$ field operations (see [DT24] for full details), while the other two sum-check invocations (to compute $\widetilde{a}(r')$ and $\widetilde{b}(r')$) together cost $M + 5n$ multiplications. In total this is $6M + 5n$ multiplications for the prover.

It remains to account for the prover cost of applying Shout. Shout is applied to perform $2T$ lookups into a table of size $M \cdot n$, and this table is amenable to the sparse-dense sum-check protocol (Section 7.1). Hence, the cost is at most $2(d^2 + 4)T$ field multiplications for the prover.

The prover must also provide one evaluation of each committed polynomial $\widetilde{\mathsf{ra}}_i$. If Hyrax is the commitment scheme used, this costs T field multiplications for the prover (and no cryptographic operations), with the field operations needed simply to compute the requested evaluations. If Dory is the commitment scheme used, then additionally several multi-pairings and scalar multiplications all of size at most $\sqrt{(Mn)^{1/d} \cdot T}$ are also required (see Section 9.2.3 for details of what Dory evaluation proofs entail). Asymptotically this is a low-order cost relative to the linear field work required in Shout, though concretely it may be a prover bottleneck unless d is fairly large, say $d \geq 4$.

Most applications have a small number of public inputs, and hence $n \approx m$. Then the above costs can be summarized as follows. Ignoring the (asymptotically low-order) costs of computing evaluation proofs, the Spartan++ prover commits only to the values of all M multiplication gates in the circuit, and performs the following number of field multiplications:

$$6M + 5n + 2(d^2 + 5)T \approx 11M + (2d^2 + 10)T.$$

For $d = 4$ this translates to about 42 field multiplications per addition gate and 53 per multiplication gate. This is concretely slower than SpeedySpartan (see Figure 11) though it is interesting that the Spartan++ prover's commitment costs grow only with M, the number of multiplication gates.

Despite being slower than SpeedySpartan, Spartan++ is about 6× faster than Spartan itself. The cost analysis for Spartan underlying this comparison is as follows. Recall that Spartan applies Spark to $\widetilde{A}$ and $\widetilde{B}$, and that Spark applies Lasso internally. In this application of Lasso, the prover commits to 2 random field elements per non-zero entry of A and B. This translates to about 22 group operations per non-zero, which is roughly equivalent to $22 \cdot 6 = 132$ field operations. On top of the commitment costs, the prover also performs 24 field operations within Lasso. So this yields 156 field operations per non-zero matrix entry. Since there are $2T$ non-zeros between the two matrices, this is about $312T$ field operations in total to implement Lasso-within-Spark-within-Spartan. This is about 6× the prover cost of Spartan++ with $d = 4$.

Disclosures. Justin Thaler is a Research Partner at a16z crypto and is an investor in various blockchain-based platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see `https://www.a16z.com/disclosures/`.)

References

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

[And17] Andrew Waterman1, Krste Asanovic. The RISC-V instruction set manual. `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`, 2017.

[AS24] Arasu Arun and Srinath Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. Cryptology ePrint Archive, 2024.

[AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2024.

[AW09] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM Transactions on Computation Theory (TOCT)*, 1(1):1–54, 2009.

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

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2018.

[BC24] Benedikt Bünz and Jessica Chen. Proofs for deep thought: Accumulation for large memories and deterministic computations. In *International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt)*, pages 269–301. Springer, 2024.

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

[BCF+24] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D Rothblum, and Hadas Zeilberger. Blaze: Fast SNARKs from interleaved RAA codes. Cryptology ePrint Archive, 2024.

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

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: Extended abstract. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS)*, 2013.

[BEG+91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991. Journal version in Algorithmica, 199.

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

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, August 2013.

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

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

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

[Dor24] Daniel Dore. TaSSLE: Lasso for the commitment-phobic. Cryptology ePrint Archive, 2024.

[DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023. `https://eprint.iacr.org/2023/1784`.

[DP24] Benjamin E Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, 2024.

[DT24] Quang Dao and Justin Thaler. More optimizations to sum-check proving. Cryptology ePrint Archive, Paper 2024/1210, 2024.

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

[EHB22] Youssef El Housni and Gautam Botrel. EdMSM: multi-scalar-multiplication for SNARKs and faster montgomery multiplication. *Cryptology ePrint Archive*, 2022.

[FS86a] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[FS86b] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 186–194, 1986.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)*, 62(4):1–64, 2015.

[GLH+24] Yanpei Guo, Xuanming Liu, Kexi Huang, Wenjie Qu, Tianyang Tao, and Jiaheng Zhang. Deepfold: Efficient multilinear polynomial commitment from reed-solomon code and its application to zero-knowledge proofs. Cryptology ePrint Archive, 2024.

[GM24] Albert Garreta and Ignacio Manzur. FLI: Folding lookup instances. Cryptology ePrint Archive, Paper 2024/1531, 2024.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, 2021.

[Gru24] Angus Gruen. Some improvements for the piop for zerocheck. Cryptology ePrint Archive, 2024.

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

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

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

[Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.

[KS24a] Abhiram Kothapalli and Srinath Setty. HyperNova: Recursive arguments for customizable constraint systems. In *Annual International Cryptology Conference*, pages 345–379, 2024.

[KS24b] Abhiram Kothapalli and Srinath Setty. NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive, 2024.

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

[KT23] Tohru Kohrita and Patrick Towa. Zeromorph: Zero-knowledge multilinear-evaluation proofs from homomorphic univariate commitments. Cryptology ePrint Archive, 2023.

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

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

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

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

[Lip90] Richard J Lipton. Efficient checking of computations. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 207–215. Springer, 1990.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[NT25] Vineet Nair and Justin Thaler. Proving CPU executions in small space. 2025.

[OLB24] Alex Ozdemir, Evan Laufer, and Dan Boneh. Volatile and persistent memory for zkSNARKs via algebraic interactive proofs. Cryptology ePrint Archive, Paper 2024/979, 2024.

[PH23] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, 2023.

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

[Rob91] John Michael Robson. An o (t log t) reduction from ram computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[Rot24] Ron D. Rothblum. A note on efficient computation of the multilinear extension. Cryptology ePrint Archive, Paper 2024/1103, 2024.

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

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

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

[SSS22] Robin Salen, Vijaykumar Singh, and Vladimir Soukharev. Security analysis of elliptic curves over sextic extension of small prime fields. Cryptology ePrint Archive, 2022.

[ST23] Srinath Setty and Justin Thaler. BabySpartan: Lasso-based SNARK for non-uniform computation. Cryptology ePrint Archive, 2023.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, 2023.

[STW24] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with Lasso. Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), 2024.

[Tea22] Polygon Zero Team. Plonky2: Fast recursive arguments with PLONK and FRI, 2022.

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

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

[VSBW13] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[WSR+15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

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

[YH23] Yibin Yang and David Heath. Two shuffles make a RAM: Improved constant overhead zero knowledge RAM. Cryptology ePrint Archive, Paper 2023/1115, 2023.

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

[ZGK+18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

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

[ZSC24] Jiaxing Zhao, Srinath Setty, and Weidong Cui. MicroNova: Folding-based arguments with efficient (on-chain) verification. Cryptology ePrint Archive, Paper 2024/2099, 2024.

A Overview of offline memory-checking protocols

Read/write memories. For read/write memories, there are two high-level approaches. In the first approach, which we refer to as *reordering-based approach*. the prover commits to a *re-ordering* of the memory operations (both reads and writes) such that they are grouped by which memory cell is accessed, and ordered by timestep (i.e., cycle) within each group. Confirming that this "second" committed vector of memory operations is indeed a reordering of the first amounts to a permutation check. It must be proved that within each group the memory operations are indeed ordered by timestep. Given this ordering, it is then relatively straightforward to prove that each write to that cell returns the value most recently written to that cell. This approach was first implemented in [BSCG+13, WSR+15], but dates back to the much older work [Rob91, BCGT13]. For proving a permutation check, these old works employ a permutation network (a technique originally motivated by non-interactivity requirements of the PCP model). The approach was refined in vRAM [ZGK+18] with the use of interaction to use fingerprinting techniques [Lip89, Lip90] (see

Section 2.3) rather than a permutation network. See also a recent work by Ozdemir et al. [OLB24] that uses the reordering-based approach.

The second approach, which we refer to as "reordering-free approach", was first implemented and made practical by Spice [SAGL18], building on the older work of Blum et al. [BEG+91]. Spartan [Set20] refined this approach to use fingerprinting-based multiset equality check, and employed it for proving sparse polynomial evaluations. This approach is further refined in Jolt [AST24], to use lookup arguments for establishing timestamp checks in the memory checking algorithm. Recently, Yang and Heath [YH23] describe an approach that is essentially the same as Spice's for proving memory operations.

In this approach, the prover does not reorder the memory operations, but commits to some extra data for each read and write to memory. Specifically, each cell is augmented to store not only a value but a timestamp, which is returned with each read and updated with each write. Also, each read to memory is followed by a "dummy write" operation that writes the returned value back to the same cell with the timestamp updated. Similarly, each write operation is preceded by a "dummy read" operation to that cell. The point of having the prover commit to this extra data (timestamps, dummy reads, and dummy writes) is that it ensures the set of "read tuples" (i.e., address, value, timestamp triples) returned by all read operations is a permutation of the set of all "write tuples" if and only if the value returned by every read operation is indeed the value most recently written to that cell.

At least as implemented in Jolt, the way that timestamps are computed on write operations in Spice necessitates range-checking two values. Specifically, Spice maintains a *global timestamp* gt. At the end of cycle i, gt is set to i. Here, we use the term cycle to refer to one read operation followed by one write operation, where one of the two operations is a "dummy" operation as described above.

In Spice, when a read operation returns a value along with timestamp t, the associated write operation (if the prover is honest) uses timestamp $t' = \max\{t+1, \mathsf{gt}+1\}$. Intuitively, the point of using this timestamp update procedure is to prevent "time travel attacks", where a cheating prover returns on an early read a value that will only be written in the future to that cell, and returns on a later read a value that was written in the distant past to that cell. The timestamp update procedure in Spice forces the timestamps to increase monotonically as the cycles proceed, preventing this attack.

If the prover is honest, the timestamp t returned by a read operation at cycle i is always less than or equal to $\mathsf{gt} = i - 1$, and under this guarantee, the prescribed write timestamp t' simply equals $\mathsf{gt} + 1$. To ensure that t is indeed less than or equal to gt, Jolt confirms that both t itself and $\mathsf{gt} - t$ are in the range $\{0, \ldots, T-1\}$, where T is (an upper bound on) the total number of cycles that the zkVM is run for.

These "read-timestamp range checks" are implemented in Jolt by applying the Lasso lookup argument [STW24] to the table $\{0, \ldots, T-1\}$. In this application of Lasso, the prover commits to two small values per lookup, and hence four small values total. These four values also contribute one factor each (so four factors in total) to a grand product that must be proved within the invocation of Lasso.

Read-only memories and lookup arguments. Lookup arguments refer to protocols that allow a prover to prove that every value in one vector v is present in some position in another vector called a lookup table t. The verifier in this protocol holds commitments to both v and t (or, in many cases, t is sufficiently structured that no commitment to it is needed).

While lookup arguments as typically formulated do not care about the position of the entries in the lookup table, Lasso [STW24] introduced a generalization of lookup arguments, called *indexed lookups*, where there is an additional committed vector a, and the prover must prove that $\forall i \in [|v|]$, $v[i] = t[a[i]]$. For ease of reference, we refer to traditional lookup arguments as unindexed lookup arguments. Note that indexed lookup arguments trivially provide unindexed lookup arguments by having the prover provide a commitment to the vector a as its first message. There are also transformations from unindexed lookup arguments to indexed lookup arguments (see [STW24] for details). Indexed lookup arguments are equivalent to memory-checking arguments for read-only memory.

There are many lookup arguments in the literature [BCG+18, GW20b, ZBK+22, PK22, ZGK+22, EFG22, Hab22, STW24, PH23], and they all rely on fingerprinting techniques described in Section 2.3 to prove that

the lookup relation holds. We now provide details on some of these works.

Among these, Lasso [STW24] directly builds on the reordering-free read-write memory approach, specifically it generalizes the approach in Spartan [Set20], which itself was specialized from the read/write memories in Spice [SAGL18]. Since the values are only read and never written, some simplifications are possible compared to the read/write setting solved in Spice: there is no need to perform range checks on timestamps, they can be simply incremented by 1.

The first explicit (unindexed) lookup argument was given in Arya [BCG+18]. In Arya, the prover commits to the binary representation of address *multiplicities*, i.e., the number of times each address is read. The prover then proves that looked-up values are a permutation of the vector in which each table element is repeated a number of times equal to its (committed) multiplicity. The approach in Plookup [GW20b] can be viewed as similar to the reordering-based read-write memory in vRAM, with some simplifications arising from the unindexed lookup argument setting. In particular, in Plookup, the prover commits to a sorting of the table entries and looked-up values. The approach in LogUp [Hab22] is similar to plookup [GW20b] except that it changes the fingerprinting strategy: while plookup uses a grand product, LogUp uses a grand sum of inverses.

A key insight of Lasso [STW24] is that the use of sum-check-based layered grand product arguments [Tha13] avoids the need to commit to random field elements, resulting in large prover speedups. Inspired by this, LogUpGKR [PH23] employs a sum-check-based layered grand sum argument to improve the prover efficiency of LogUp. LogUpGKR can also be viewed as a refinement of the original lookup argument, Arya, in that the LogUpGKR prover commits only to the multiplicities of each address (unlike Arya, in LogUpGKR, these multiplicities are each specified via a single field element rather than via its binary representation).

Separately, a line of work starting with Caulk and culminating in cq [ZBK+22, PK22, ZGK+22, EFG22], seeks lookup arguments in which an expensive pre-processing phase is performed (requiring, e.g., a number of scalar multiplications that is quasilinear in table size) after which m lookups can be answered in time that depends only on m and not on table size. However, these solutions are concretely expensive (e.g., cq requires the prover to commit to seven random field elements per lookup).

B Details of the $\widetilde{\mathsf{Val}}$-evaluation sum-check prover

B.1 Computing the Less-Than evaluation table

In the $\widetilde{\mathsf{Val}}$-evaluation sum-check in Twist (Figure 9), the prover needs to compute $\widetilde{\mathsf{LT}}(j', r)$ for all $j' \in \{0,1\}^{\log T}$ for $r = r_{\text{cycle}} \in \mathbb{F}^{\log T}$. Here, $\widetilde{\mathsf{LT}}$ is the multilinear extension of the $\mathsf{LT}(x, y)$ function defined in Section 3.2, which indicates whether or not x is the binary representation of an integer that is strictly less than (the integer with binary representation given by) y.

We explain here how the prover can do this with only T field multiplications. We use simple, explicit expressions for $\widetilde{\mathsf{LT}}$ given in [STW24].

Let $x = (x_1, \ldots, x_{\log T})$ and $y = (y_1, \ldots, y_{\log T})$. If x and y are binary representations of integers between 0 and $T-1$, we think of $x_{\log T}$ and $y_{\log T}$ as the "low-order bit" of the two binary representations, and x_1 and y_1 as the high order bit.

For $i = 2, \ldots, \log T$, let $x_{>i} = (x_{i+1}, \ldots, x_{\log T})$ and $x_{\geq i} = (x_i, \ldots, x_{\log T})$. Define

$$\widetilde{\mathsf{LT}}_i(x, y) = (1 - x_i) \cdot y_i \cdot \widetilde{\mathsf{eq}}(x_{>i}, y_{>i}). \tag{86}$$

It is easy to see that

$$\widetilde{\mathsf{LT}}(x, y) = \sum_{i=1}^{\log T} \widetilde{\mathsf{LT}}_i(x, y). \tag{87}$$

Note that LT_i does not depend on the first (i.e., low-order) $i-1$ bits of x. Accordingly, while Equation (86) expresses LT_i as a function of all $\log T$ bits of x and y, we can equivalently think of LT_i as a function only of the last $\log(T) - (i-1)$ bits of x and y. We will follow this convention henceforth.

Suppose the prover has already computed a table E_i defined as follows:

$$E_i \text{ stores all } T/2^i \text{ evaluations of } \widetilde{\mathsf{eq}}(j'_{>i}, r_{>i}) \text{ as } j'_{>i} \text{ ranges over } \{0,1\}^{\log(T)-i}. \tag{88}$$

Suppose further that the prover has also already computed a table D_i of size $T/2^{i-1}$ such that for all $x \in \{0,1\}^{\log(T)-(i-1)}$,

$$D_i[x] = \sum_{\ell=i}^{\log T} \widetilde{\mathsf{LT}}_\ell(x, r_{\geq \ell}). \tag{89}$$

Via standard observations [VSBW13] (see Lemma 1), the tables $E_{\log(T)-1}, \ldots, E_1$ can all be computed with $T/2$ field multiplications in total. Furthermore, given table E_i, the table D_i (which recall has size $2 \cdot T/2^i$) can be computed with just $T/2^i$ field multiplications. To see this, observe that the following holds.

Let $x = (x_i, \ldots, x_{\log T}) \in \{0,1\}^{\log(T)-(i-1)}$ and let $x_{>i} = (x_{i+1}, \ldots, x_{\log T})$. Then:

- If $x_i = 1$, $D_i[x] = D_{i+1}[x_{>i}]$. This is because

$$\widetilde{\mathsf{LT}}_i(x, r_{\geq i}) = (1 - x_i) \cdot r_i \cdot \widetilde{\mathsf{eq}}(x_{>i}, r_{>i}) = 0.$$

- If $x_i = 0$ then $D_i[x] = D_{i+1}[x_{>i}] + r_i \cdot E_i[x_{>i}]$.

The table D_1 is exactly the table we want, containing all T evaluations of $\widetilde{\mathsf{LT}}(j', r)$ as j' ranges over $\{0,1\}^{\log T}$. As explained above, D_1 can be computed with $3T/2$ multiplications in total. $T/2$ of these are devoted to computing all of the E_i tables, and an additional T multiplications are devoted to computing all of the D_i tables.

B.2 Optimizing the $\widetilde{\mathsf{Val}}$-evaluation sum-check prover further

Consider using the sum-check protocol to compute

$$\sum_{x \in \{0,1\}^m} \widetilde{\mathsf{LT}}(r', x) \cdot g(x).$$

In Twist we are interested in the case where g is multilinear, i.e., where g has degree $d = 1$ in each variable, so let us restrict our discussion to that case. We assume that an array storing all evaluations of g over $\{0,1\}^m$ is already computed prior to the start of the prover algorithm (i.e., our accounting in this section does not charge the prover to compute such an array. In Twist, it will cost $2K$ field operations per Section 8.1).

Let $(r_1, \ldots, r_{i-1})$ denotes the randomness chosen by the sum-check verifier in rounds $1, \ldots, i-1$. Then

$$
\begin{aligned}
s_i(c) &= \sum_{x' \in \{0,1\}^{m-i}} \widetilde{\mathsf{LT}}(r', r_1, \ldots, r_{i-1}, c, x') \cdot g(r_1, \ldots, r_{i-1}, c, x') \\
&= \sum_{x' \in \{0,1\}^{m-i}} \left(\sum_{j=1}^{m} (1 - r'_j) \cdot r_j \cdot \widetilde{\mathsf{eq}}(r'_{>j}, (r_1, \ldots, r_{i-1}, c, x')_{>j}) \right) \cdot g(r_1, \ldots, r_{i-1}, c, x') \\
&= \left(\sum_{x' \in \{0,1\}^{m-i}} \left(\sum_{j=1}^{i-1} (1 - r'_j) \cdot r_j \cdot \widetilde{\mathsf{eq}}(r'_{>j}, r_{j+1}, \ldots, r_{i-1}, c, x') \right) \cdot g(r_1, \ldots, r_{i-1}, c, x') \right) \\
&+ \left(\sum_{x' \in \{0,1\}^{m-i}} (1 - r_i) \cdot c \cdot \widetilde{\mathsf{eq}}(r'_{>i}, x') \cdot g(r_1, \ldots, r_{i-1}, c, x') \right) \\
&+ \left(\sum_{x' \in \{0,1\}^{m-i}} \left(\sum_{j=i+1}^{m} (1 - r'_j) \cdot x'_{j-i} \cdot \widetilde{\mathsf{eq}}(r'_{>j}, r_{j+1}, \ldots, r_{i-1}, c, x') \right) \cdot g(r_1, \ldots, r_{i-1}, c, x') \right) \\
&= \left(\sum_{x' \in \{0,1\}^{m-i}} \left(\sum_{j=1}^{i-1} (1 - r'_j) \cdot r_j \cdot \widetilde{\mathsf{eq}}(r'_{>j}, r_{j+1}, \ldots, r_{i-1}, c, x') \right) \cdot g(r_1, \ldots, r_{i-1}, c, x') \right) && (90) \\
&+ \left(\sum_{x' \in \{0,1\}^{m-i}} (1 - r_i) \cdot c \cdot \widetilde{\mathsf{eq}}(r'_{>i}, x') \cdot g(r_1, \ldots, r_{i-1}, c, x') \right) && (91) \\
&+ \left(\sum_{x' \in \{0,1\}^{m-i}} D_{i+1}[x'] \cdot g(r_1, \ldots, r_{i-1}, c, x') \right). && (92)
\end{aligned}
$$

Here, the first equality invokes Expressions (86) and (87) for $\widetilde{\mathsf{LT}}$ and $\widetilde{\mathsf{LT}}_i$. The final equality invokes the definition of D_{i+1} (Equation (89)), using the shorthand

$$D_{i+1}[c, x'] = (1 - c) \cdot D_{i+1}[0, x'] + c \cdot D_{i+1}[1, x'].$$

Call Expression (90) (i.e., the first sum the final expression) $A(c)$, the second sum (i.e., Expression (91)) $B(c)$ and and the third sum (i.e., Expression (92)) $C(c)$.

Turning our attention to $A(c) + B(c)$, for each $1 \leq j \leq i$, let

$$Q_{j,i}(c) = (1 - r'_j) \cdot r_j \cdot \prod_{z=j+1}^{i-1} \widetilde{\mathsf{eq}}(r'_{j+1}, \ldots, r'_{i-1}, r_{j+1}, \ldots, r_{i-1}) \cdot \widetilde{\mathsf{eq}}(r'_i, c).$$

The prover can, with a constant number of field multiplications per round, maintain the quantity $Q_{i,c} \coloneqq \left(\sum_{j=1}^{i} Q_{j,i}(c) \right)$.

Observe that

$$A(c) + B(c) = \left(\sum_{j=1}^{i} Q_{j,i}(c) \right) \sum_{x' \in \{0,1\}^{m-i}} \widetilde{\mathsf{eq}}(r'_{>i}, x') \cdot g(r_1, \ldots, r_{i-1}, c, x').$$

Define

$$A'(c) = \sum_{x' \in \{0,1\}^{m-i}} \widetilde{\mathsf{eq}}(r'_{>i}, x') \cdot g(r_1, \ldots, r_{i-1}, c, x') = \sum_{x' \in \{0,1\}^{m-i}} E_i[x'] \cdot g(r_1, \ldots, r_{i-1}, c, x'),$$

where recall that E_i is defined in Equation (88). Then for any point c,

$$s_i(c) = \left(\sum_{j=1}^{i-1} Q_{j,i}(c)\right) A'(c) + C(c). \tag{93}$$

The sum-check prover needs to evaluate s_i at $c \in \{0, 2\}$. The prover will store all the intermediate arrays $D_m, D_{m-1}, \ldots$ and $E_m, E_{m-1}, \ldots, E_1$ en route to building D_1 per Section B.1. (As we will see momentarily, ultimately the prover does not even need the array D_1). Then per the standard linear-time sum-check proving algorithm (Section 3.3), the prover can compute

$$C(c) = \sum_{x' \in \{0,1\}^{m-i}} D_{i+1}[x'] \cdot g(r_1, \ldots, r_{i-1}, c, x'),$$

and

$$A'(c) = Q_{i,c} \cdot \sum_{x' \in \{0,1\}^{m-i}} E_i[x'] \cdot g(r_1, \ldots, r_{i-1}, c, x'),$$

with 2^{m-i} field multiplications each.

In total, across the entire protocol, the prover spends $T/2$ multiplications to build the arrays $E_m, \ldots, E_1$, $T/2$ additional multiplications to build the arrays $D_m, \ldots, D_2$, T multiplications to bind the array storing evaluations of g, T multiplications to compute $C(0)$ and $C(2)$ across all rounds i given these arrays, and T to compute $A'(0)$ and $A'(2)$ across all rounds. This is $4T$ multiplications in total.

C A Shout variation with a linear prover dependence on d

Recall from Equation (30) that Shout simply applies the sum-check protocol to confirm that:

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j) \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j)\right) \cdot \widetilde{\mathsf{Val}}(k).$$

For very large values of d, a nuisance is that the final $\log T$ rounds of this protocol cause the prover to incur roughly $d^2 T$ field multiplications. Here, we describe an alternative application of the sum-check protocol that avoids quadratic dependence on d (but has a worse leading constant, and hence is only preferable when d is very large).

Let $j_0, j_1, \ldots, j_d$ each consist of $\log T$ variables. Let $\widetilde{\mathsf{eq}}(j_0, j_1, \ldots, j_d)$ denote the multilinear extension of the function that evaluations to 1 if all $d+1$ vectors $j_0, \ldots, j_d$ are equal, and zero otherwise. Then

$$\widetilde{\mathsf{eq}}(j_0, j_1, \ldots, j_d) = \prod_{i=1}^{\log T} \left(\left(\prod_{k=0}^{d} (1 - j_{k,i})\right) + \left(\prod_{k=1}^{d} j_{k,i}\right)\right). \tag{94}$$

Now, in our modified version of Shout, replace Equation (30) with:

$$\widetilde{\mathsf{rv}}(r_{\text{cycle}}) = \sum_{k=(k_1,\ldots,k_d) \in \left(\{0,1\}^{\log(K)/d}\right)^d,\ j_1,\ldots,j_d \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j_1, j_2, \ldots, j_d) \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j_i)\right) \cdot \widetilde{\mathsf{Val}}(k). \tag{95}$$

Note that this invocation of the sum-check protocol has $\log(K) + d \log T$ rounds rather than $\log(K) + \log(T)$ rounds in the standard variant of Shout (i.e., applying sum-check to check Equation (30)). However, the total proof size remains $O(\log(K) + d \log(T))$ field elements, as each of the final $d \log T$ rounds in the new variant has the prover send a degree-2 polynomial (whereas in the standard variant the prover sent a degree-d polynomial in each of the final $\log T$ rounds).

Theorem 7. *Shout (Figure (7)) with Equation (30) replaced by Equation (95) is perfectly complete and has soundness error $O((\log(K) + d\log(T))/|\mathbb{F}|)$.*

Proof. The proof is identical to Theorem 3, except that we must establish that Equation (95) holds. To see this, observe that the right hand side of Equation (95) is a multilinear polynomial in r_{cycle}. So it suffices to establish that the right hand side and left hand side agree at all inputs $r_{\text{cycle}} \in \{0,1\}^{\log T}$. Observe that for any term of the sum on the right hand side, if $j_1, \ldots, j_d \in \{0,1\}^{\log T}$, then $\widetilde{\mathsf{eq}}(r_{\text{cycle}}, j_1, j_2, \ldots, j_d) = 0$ unless

$$j_1 = j_2 = \cdots = j_d = r_{\text{cycle}}, \tag{96}$$

and if this equality does hold then $\widetilde{\mathsf{eq}}(r_{\text{cycle}}, j_1, j_2, \ldots, j_d) = 1$.

If Equation (96) does hold, then $\left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j_i)\right) = 1$ if the register with binary representation $k = (k_1, \ldots, k_d)$ is read at cycle r_{cycle}, and $\left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(k_i, j_i)\right) = 0$ otherwise. Hence, the right hand side equals $\widetilde{\mathsf{Val}}(k)$ where k is the register read at cycle r_{cycle}. The result follows. □

C.1 Fast prover implementation

First $\log K$ rounds. The first $\log K$ rounds of this sum-check invocation proceed identically to the standard variant of Shout (in these rounds, the prover message is no different between the two variants). The reader may be initially surprised by this because Expression (95) has $K \cdot T^d$ terms while Expression (30) only has $K \cdot T$ terms. However, per the proof of Theorem 7, the factor $\widetilde{\mathsf{eq}}(r_{\text{cycle}}, j_1, j_2, \ldots, j_d)$ evaluates to 0 over $\left(\{0,1\}^{\log T}\right)^d$ unless $j_1 = j_2 = \cdots = j_d$. This means that in the first $\log K$ rounds (where only variables of k are being bound, not variables of $j_1, \ldots, j_d$), $j_1, \ldots, j_d$ effectively act as a single entity $j \in \{0,1\}^{\log T}$, and this leads to the same prover messages as in the original Shout protocol for these rounds.

Final $d \log T$ rounds. Let $r = (r_1, \ldots, r_d) \in \left(\mathbb{F}^{\log(K)/d}\right)^d$ denote the randomness chosen by the verifier across the first $\log K$ rounds of sum-check. Then the final $d \log T$ rounds compute the following sum:

$$\widetilde{\mathsf{Val}}(r) \cdot \sum_{j_1, \ldots, j_d \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r_{\text{cycle}}, j_1, j_2, \ldots, j_d) \left(\prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(r_i, j_i)\right). \tag{97}$$

We bind the first variable of $j_1, j_2, \ldots, j_d$ in sequence, followed by the second variable of each, and so on. Let us call the first d rounds "Stage 1" (i.e., when we bind the first variable of each of $j_1, j_2, \ldots, j_d$), the next d rounds Stage 2, and so forth.

The key insight for a fast prover. While there are T^d terms in Expression (97), round-over-round almost all of them are zero. Specifically, $\widetilde{\mathsf{eq}}(r_{\text{cycle}}, j_1, j_2, \ldots, j_d)$ is a product of $\log(T)$ terms, one per variable in each j_i–intuitively, it tests equality of $j_1, \ldots, j_d$ "bitwise". Since $\widetilde{\mathsf{eq}}$ checks equality of each bit independently, it turns out that in each round of the protocol, the prover need only consider terms that agree in all but their first bit. This keeps the total prover time in each round of Stage S to $T/2^S$. Since there are d rounds in each stage, this is $O(\sum_{S=1}^{\log T} dT/2^s) = O(dT)$ time in total.

C.1.1 Detailed prover algorithm description for $d = 2$

To simplify notation, we begin with a description of the prover algorithm in the case $d = 2$. Further, let $r' = r_{\text{cycle}}$.

Consider the first round within Stage i. This is round $t = 2(i-1) + 1$ within the final $d \log T$ rounds of the sum-check applied to compute Expression (95), and round $t' = \log(K) + t$ within the entire modified-Shout protocol.

Let $w_1 = (w_{1,1}, \ldots, w_{1,i-1}) \in \mathbb{F}^{i-1}$ denote the vector of random values chosen for the first $i-1$ coordinates of j_1 across the already-completed $i-1$ stages, and similarly let $w_2 = (w_{2,1}, \ldots, w_{2,i-1})$.

Simplifying the prover's message definition. The prover's prescribed message $s_{t'}$ in round t' of our modified version of Shout satisfies:

$$s_t(c) = \widetilde{\mathsf{Val}}(r) \cdot \sum_{j_1' \in \{0,1\}^{\log(T)-i}, j_2' \in \{0,1\}^{\log(T)-i+1}} \widetilde{\mathsf{eq}}\,(r', (w_1, c, j_1'), (w_2, j_2')) \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j_1') \cdot \widetilde{\mathsf{ra}}_2(r, w_2, j_2').$$

Clearly, $\widetilde{\mathsf{eq}}\,(r', (w_1, c, j_1'), (w_2, j_2')) = 0$ unless $j_{2,>1}' = j_1'$. So this sum simplifies to

$$\widetilde{\mathsf{Val}}(r) \cdot \sum_{j_1' \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}\,(r', (w_1, c, j_1'), (w_2, b, j_1')) \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j_1') \cdot \widetilde{\mathsf{ra}}_2(r, w_2, b, j_1'). \tag{98}$$

For any vector $x \in \mathbb{F}^{\log T}$, let $x_{>i}$ denote the last $\log(T) - i$ coordinates of x. By Equation (94),

$$\widetilde{\mathsf{eq}}\,(r', (w_1, c, j_1'), (w_2, b, j_1')) = \left(\prod_{\ell=1}^{i-1} (r_\ell' w_{1,\ell} w_{2,\ell} + (1 - r_\ell')(1 - w_{1,\ell})(1 - w_{2,\ell}))\right) \cdot \widetilde{\mathsf{eq}}(r_i', c, b) \cdot \widetilde{\mathsf{eq}}(r_{>i}', w_{1,>i}, w_{2,>i}).$$

Hence, letting

$$C = \widetilde{\mathsf{Val}}(r) \cdot \left(\prod_{\ell=1}^{i-1} (r_i' w_{1,\ell} w_{2,\ell} + (1 - r_\ell')(1 - w_{1,\ell})(1 - w_{2,\ell}))\right), \tag{99}$$

Expression (98) equals:

$$C \cdot \sum_{j_1' \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}(r_i', c, b) \cdot \widetilde{\mathsf{eq}}(r_{>i}', j_1', j_1') \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j_1') \cdot \widetilde{\mathsf{ra}}_2(r, w_2, b, j_1'). \tag{100}$$

Furthermore, note that for $j_1' \in \{0,1\}^{\log(T)-i}$,

$$\widetilde{\mathsf{eq}}(r_{>i}', j_1', j_1') = \widetilde{\mathsf{eq}}(r_{>i}', j_1'),$$

where the polynomial $\widetilde{\mathsf{eq}}$ on the right hand side refers to Equation (15). Hence, Expression (100) equals:

$$C \cdot \sum_{j_1' \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}(r_i', c, b) \cdot \widetilde{\mathsf{eq}}(r_{>i}', j_1') \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j_1') \cdot \widetilde{\mathsf{ra}}_2(r, w_2, b, j_1'). \tag{101}$$

Similarly, turning our attention to the prover's message in round $t+1$,

$$s_{t+1}(c) = C \cdot \widetilde{\mathsf{eq}}(r_i', w_{1,i}, c) \cdot \sum_{j_2' \in \{0,1\}^{\log(T)-i}} \widetilde{\mathsf{eq}}(r_{>i}', j_2') \cdot \widetilde{\mathsf{ra}}_1(r, w_1, r_{1,i}, j_2') \cdot \widetilde{\mathsf{ra}}_2(r, w_2, c, j_2'). \tag{102}$$

Prover algorithm and cost accounting for $d = 2$. Before the final $d \log T$ rounds of Shout's read-checking sum-check protocol begin, the prover the prover computes and stores $\log(T) - 1$ arrays $E_{\log(T)-1}, \ldots, E_1$, where E_i contains all evaluations $\widetilde{\mathsf{eq}}(r_{>i}, j_{>i})$ as $j_{>i}$ ranges over $\{0,1\}^{\log(T)-i}$. Also, at the start of these final $\log T$ rounds, the prover will have already computed two arrays, say A_1 and A_2, that store all evaluations of the form $\widetilde{\mathsf{ra}}_1(r', j)$ and $\widetilde{\mathsf{ra}}_2(r', j)$ as j ranges over $\{0,1\}^{\log T}$. As per the standard linear-time sum-check prover algorithm (Section 3.3), the prover will bind A_1 and A_2 (see Equation (20)) as appropriate round-over-round. Per Section 6.2.2, if $K^{1/d} = o(T)$, the cost of binding these arrays round-over-round is a low-order cost, so we ignore it in our accounting below.

The prover can maintain the value C from Equation (99) round-over-round with only a constant number of field multiplications per round.

During the first round in Stage i, with prover message given by Expression (101). Note that for $c = 0$, $\widetilde{\mathsf{eq}}(r'_i, c, b) = 0$ if $b = 1$, so for $c = 0$ $b = 1$ need not be considered at all by the prover. In addition, $\widetilde{\mathsf{eq}}(r'_{>i}, j'_1)$ can be obtained with a single lookup into E_i. The prover can compute Expression (101) for $c = 0$ with $2T/2^i + O(1)$ field multiplications. Expression (101) for $c = 2$ can be computed with $4T/2^i + O(1)$ multiplications (twice as many as for $c = 0$ because both $b = 0$ and $b = 1$ must be considered when $c = 2$).

The prover's message for second round of Stage i (Expression (102)) requires at most $4T/2^i$ field multiplications, with $2T/2^i$ required for $c = 0$ and the same number required for $c = 2$.

Hence, each stage $i = 1, \ldots, \log(T)$ requires $10T/2^i$ field multiplications for the prover. Summing across all i, this is at most $10T$ multiplications. Accounting also for the $T/2$ multiplications to compute the arrays $E_{\log(T)-1}, \ldots, E_1$, in total across all stages i this is $10.5 \cdot T$ field multiplications.

An optimization when $K^{1/d}$ is $o(\sqrt{T})$. When $K^{1/d} = o(\sqrt{T})$, a significant optimization is possible, similar to the one described in Section 6.2.2 that renders binding the arrays A_1 and A_2 a low-order cost. At the start of the final $d \log T$ rounds of sum-check, for each $\ell \in \{1, 2\}$, $\widetilde{\mathsf{ra}}_\ell(r, j)$ takes on only $K^{1/d}$ distinct values as j ranges over $\{0,1\}^{\log T}$. Hence, all possible products of the form $\widetilde{\mathsf{ra}}_1(r, j_1) \cdot \widetilde{\mathsf{ra}}_2(r, j_2)$ as j_1 and j_2 range over $\{0, 1, 2\} \times \{0, 1\}^{\log(T)-1}$ can be computed and stored in a table of size $O(K^{2/d})$, thereby saving the prover the need to incur T field multiplications to compute such quantities arising in Expression (100) for round $\log(T) + 1$. Similar optimizations apply for several rounds, with the size of the lookup table of "pre-computed products of $\widetilde{\mathsf{ra}}_1$ and $\widetilde{\mathsf{ra}}_2$ evaluations" growing (rapidly) round-over-round. After a few rounds, the lookup tables would have size larger than T and this optimization is no longer helpful.

Depending on how much smaller $K^{1/d}$ is than $\sqrt{T}$, this can save up to about half of the $10T$ field multiplications accounted for above. This yields a total number of multiplications of roughly $5.5 \cdot T$.

C.1.2 General d: Simplifying the prover's message definition

Consider the first round in stage i. This is round $t' = \log(K) + t$ where $t = d(i-1) + 1 = di - d + 1$. The prover's prescribed message $s_{t'}$ in round t' satisfies:

$$s_{t'}(c) = \widetilde{\mathsf{Val}}(r) \cdot \sum_{j'_1 \in \{0,1\}^{\log(T)-i}, j'_2, \ldots, j'_d \in \{0,1\}^{\log(T)-i+1}} \widetilde{\mathsf{eq}}\left(r', (w_1, c, j'_1), (w_2, j'_2), \ldots, (w_d, j'_d)\right) \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j'_1) \cdot \prod_{\ell=2}^{d} \widetilde{\mathsf{ra}}_\ell(r, w_\ell, j'_\ell).$$

Clearly, $\widetilde{\mathsf{eq}}\left(r', (w_1, c, j'_1), (w_2, j'_2), \ldots, (w_d, j'_d)\right) = 0$ unless $j'_1 = j'_{2,>1} = \cdots = j'_{d,>1}$. So this sum simplifies to

$$s_i(c) = \widetilde{\mathsf{Val}}(r) \cdot \sum_{j'_1 \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}\left(r', (w_1, c, j'_1), (w_2, b, j'_1), \ldots, (w_d, b, j'_1)\right) \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j'_1) \prod_{\ell=2}^{d} \widetilde{\mathsf{ra}}_\ell(r, w_\ell, b, j'_1). \tag{103}$$

Let

$$D = \left(\prod_{z=1}^{i-1} \left(r'_z \left(\prod_{\ell=1}^{d} w_{\ell,z} w_{2,z} \right) + (1 - r'_z) \left(\prod_{\ell=1}^{d} (1 - w_{\ell,z}) \right) \right) \right).$$

By Equation (94), for $b \in \{0, 1\}$,

$$\widetilde{\mathsf{eq}}\left(r', (w_1, c, j'_1), (w_2, b, j'_1), \ldots, (w_d, b, j'_1)\right) = D \cdot \widetilde{\mathsf{eq}}(r'_i, c, b) \cdot \widetilde{\mathsf{eq}}(r'_{>i}, j'_1, j'_1, \ldots, j'_1).$$

Hence, letting

$$C = \widetilde{\mathsf{Val}}(r) \cdot D, \tag{104}$$

Expression (103) equals:

$$C \cdot \sum_{j_1' \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}(r_i', c, b) \cdot \widetilde{\mathsf{eq}}(r_{>i}', j_1', j_1', \ldots, j_1') \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j_1') \cdot \prod_{\ell=2}^{d} \widetilde{\mathsf{ra}}_\ell(r, w_\ell, b, j_1'). \tag{105}$$

Furthermore, note that for $j_1' \in \{0,1\}^{\log(T)-i}$,

$$\widetilde{\mathsf{eq}}(r_{>i}', j_1', j_1', \ldots, j_1') = \widetilde{\mathsf{eq}}(r_{>i}', j_1'),$$

where the polynomial $\widetilde{\mathsf{eq}}$ on the right hand side refers to Equation (15). Hence, Expression (100) equals:

$$C \cdot \sum_{j_1' \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}(r_i', c, b) \cdot \widetilde{\mathsf{eq}}(r_{>i}', j_1') \cdot \widetilde{\mathsf{ra}}_1(r, w_1, c, j_1') \cdot \prod_{\ell=2}^{d} \widetilde{\mathsf{ra}}_\ell(r, w_\ell, b, j_1'). \tag{106}$$

Similarly, turning our attention to the prover's message in round M of Stage i, for $M = 2, \ldots, d-1$, let

$$C' = C \cdot \widetilde{\mathsf{eq}}(r_i', w_{1,i}, w_{2,i}, \ldots, w_{M-1,i}, c).$$

Then

$$s_{t'+M-1}(c) = C' \cdot \sum_{j_M' \in \{0,1\}^{\log(T)-i}, b \in \{0,1\}} \widetilde{\mathsf{eq}}(r_{>i}', j_M') \cdot \left(\prod_{N=1}^{M-1} \widetilde{\mathsf{ra}}_N(r, w_{N,i}, j_M') \right) \cdot \widetilde{\mathsf{ra}}_M(r, w_M, c, j_M') \cdot \left(\prod_{\ell=M+1}^{d} \widetilde{\mathsf{ra}}_\ell(r, w_\ell, b, j_M') \right). \tag{107}$$

Finally, for round $M = d$ of Stage i, the prover's message is:

$$s_{t'+M-1}(c) = C' \cdot \sum_{j_M' \in \{0,1\}^{\log(T)-i}} \widetilde{\mathsf{eq}}(r_{>i}', j_M') \cdot \left(\prod_{N=1}^{M-1} \widetilde{\mathsf{ra}}_N(r, w_{N,i}, j_M') \right) \cdot \widetilde{\mathsf{ra}}_M(r, w_M, c, j_M'). \tag{108}$$

Prover algorithm for general d. Before the final $d \log T$ rounds of Shout's read-checking sum-check protocol begin, the prover the prover computes and stores $\log(T) - 1$ arrays $E_{\log(T)-1}, \ldots, E_1$, where E_i contains all evaluations $\widetilde{\mathsf{eq}}(r_{>i}, j_{>i})$ as $j_{>i}$ ranges over $\{0,1\}^{\log(T)-i}$. Also, at the start of these final $\log T$ rounds, the prover will have already computed d arrays, say $A_1, A_2, \ldots, A_d$, such that A_ℓ stores all evaluations of the form $\widetilde{\mathsf{ra}}_\ell(r', j)$ as j ranges over $\{0,1\}^{\log T}$. As per the standard linear-time sum-check prover algorithm (Section 3.3), the prover will bind each A_ℓ (see Equation (20)) as appropriate round-over-round.

The prover can maintain the value C from Equation (104) round-over-round with only a constant number of field multiplications per round.

For each $j_M' \in \{0,1\}^{\log(T)-i}$, the prover can compute

$$\left(\prod_{\ell=M+1}^{d} \widetilde{\mathsf{ra}}_\ell(r, w_\ell, b, j_M') \right) \tag{109}$$

for *all* relevant values of M (i.e., for $M = d-1, d-2, \ldots, 1$) with $d-2$ multiplications. So that's $(d-2)T/2^i$ multiplications in total across all $j_M' \in \{0,1\}^{\log(T)-i}$.

Also, across all d rounds M of Stage i, the prover can also compute

$$\left(\prod_{N=1}^{M-1} \widetilde{\mathsf{ra}}_N(r, w_{N,i}, j_M') \right) \tag{110}$$

with $(d-2) \cdot T/2^i$ multiplications in total.

Given these values, for round $M = 1, \ldots, d-1$ in Stage i, the prover can compute its message (Expression (107)) at $c = 0$ with $2 \cdot T/2^i$ field multiplications and at $c = 2$ with $4 \cdot T/2^i$ field multiplications (up to additive $O(d)$ terms). For round d, due to the lack of a sum over $b \in \{0,1\}$, the evaluation of the prover's message at each of $c = 0$ and $c = 2$ takes $2T/2^i$ field multiplications.

Summarizing the costs:

- $T/2$ to compute the E_i arrays before the start of the final $d \log T$ rounds of the sum-check protocol.
- $2(d-2)T$ to compute the products in Expression 109 and (110).
- $6(d-1)T$ for the first $d-1$ rounds of all stages in total.
- $4T$ for the final round of all stages in total.

The total number is multiplications is therefore

$$(8d - 5.5)T.$$

Additional optimizations.

- In the final round of every stage, Gruen's optimization (Section 3.6.1) applies. Recall that Gruen's optimization considers applying sum-check to polynomials of the form $\widetilde{\mathsf{eq}}(r', x) \cdot g(x)$ and avoids $\widetilde{\mathsf{eq}}(r', x)$ from contributing to the degree of the univariate polynomial $s_t(c)$ the prover computes in each round t. This same technique applies to the factor

  $$\widetilde{\mathsf{eq}}(r'_i, w_{1,i}, w_{2,i}, \ldots, w_{M-1,i}, c) \tag{111}$$

  appearing above in the definition of C' and hence Equation (108) capturing the prover's message in the final round of stage i. The key property that ensures Gruen's optimization applies is that all values other than c appearing in Expression (111) are fixed. In other words, $r'_i, w_{1,i}, \ldots, w_{M-1,i}$ are *not* being summed over in the final round of stage i. Accordingly Expression (108) is a degree-1 polynomial in c alone. This is all Gruen's optimization needs in order to apply.

  This reduces the $4T$ multiplications required for the final round of all stages down to $2T$. Hence, it reduces total prover multiplications from $(8d - 5.5)T$ down to

  $$(8d - 7.5)T.$$

- The optimization that we described in the case of $d = 2$ that applies when $K^{1/d}$ is $o(\sqrt{T})$ also applies for general d. In the context of $d > 2$, this optimization can be used to save a significant fraction of the $2(d-2)T$ multiplications needed to compute the products in Expression 109 and (110). Details follow.

  This optimization is primarily applicable when combining Shout with a binary-field hashing-based commitment scheme like Binius [DP23, DP24] or Blaze [BCF+24], as this is the context in which d may get set quite large (and hence $K^{1/d}$ quite small) especially when considering gigantic, structured tables. For instance, if $K = 2^{64}$ then setting $d = 16$ is natural (see Section 2.8). In this case, $K^{1/d}$ is just $2^4 = 16$).

  Optimization details. For each fixed $i = 1, \ldots, d$, the evaluations $\widetilde{\mathsf{ra}}_i(r, j)$ take on only $K^{1/d}$ as j ranges over $\{0,1\}^{\log T}$. Hence, during round $\log(K) + 1$ of the modified-Shout protocol, for any integer $\ell > 1$, the prover can compute lookups tables of size $K^{2/d}, K^{3/d}, \ldots, K^{\ell/d}$, with the i'th table storing all *possible* products of the form

  $$\prod_{v=M-i}^{M} \mathsf{ra}_v(r, j), \tag{112}$$

  for $j \in \{0,1\}^{\log T}$, since there are only $K^{i/d}$ such values. After this table is computed, the prover can compute Expression (109) (i.e., the actual quantities of Expression (112) for all T values of $j \in \{0,1\}^{\log T}$)

with a single lookup into this table for each j. This saves $(\ell - 1) \cdot T$ multiplications. A similar technique applies for computing Expression (110) in round $\log(K) + 1$, but the i'th table has size $4K^{2id}$ in this case, and similarly for computing Expression (109) in round $\log(K) + 2$, and so on.

In the context of $K = 2^{64}$ and $d = 16$, using a handful of tables each of size at most 2^{20} (i.e., several dozen MBs of space), this can save $5T$ multiplications: $3T$ during the computation of Expression (110) in round $\log(K) + 1$ with ℓ set to 4, T during the computation of Expression (110) in round $\log(K) + 1$ with ℓ set to 2, and T during the computation of Expression (109) in round $\log(K) + 1$. This brings the total number of prover field multiplications in this application down to $8d - 12.5T$.