Proving CPU Executions in Small Space

Vineet Nair[1] Justin Thaler[2] Michael Zhu[3]

Abstract

zkVMs are SNARKs for verifying CPU execution. They allow an untrusted prover to show that it correctly ran a specified program on a witness, where the program is given as bytecode conforming to an instruction set architecture like RISC-V. Existing zkVMs still struggle with high prover resource costs, notably large runtime and memory usage. We show how to implement Jolt—an advanced, sum-check-based zkVM—with a significantly reduced memory footprint, *without* relying on SNARK recursion, and with only modest runtime overhead (potentially well below a factor of two). We discuss benefits of this approach compared to prevailing recursive techniques.

1 Introduction

Succinct Non-Interactive Arguments of Knowledge (SNARKs) enable an untrusted prover to convince a verifier that it possesses knowledge of a witness w satisfying a claimed property. Naively, one can always send w to the verifier, but a SNARK allows the prover to generate a short proof π that the verifier can check much more quickly than it could directly verify w.

A *zkVM* is a SNARK designed to prove that a certain CPU execution was carried out correctly on some input. The code to be executed is specified in bytecode (machine code), corresponding to a particular VM or instruction set architecture (ISA). Because the proof attests to the correctness of the underlying CPU execution, developers can write high-level programs that compile down to this ISA and still benefit from a succinct proof of correctness.

Although easy for developers to use, modern zkVMs impose large computational burdens on the prover, often requiring hundreds of thousands or even millions of times more work than directly running the program. Memory usage is also significant: proving just one million CPU cycles (which is trivial to run natively, taking less than a millisecond on a modern laptop) requires gigabytes of RAM, and proving a billion cycles requires terabytes. More generally, prover space scales linearly with the number of CPU cycles T being proven.

Background on designing SNARKs and zkVMs. At a high level, most SNARKs are designed by first designing a protocol known as a *polynomial IOP* (PIOP) and then designing a cryptographic protocol called a *polynomial commitment scheme* (PCS). Combining the two yields a succinct interactive argument, which can then be rendered non-interactive via the Fiat-Shamir transformation.

In zkVMs, the protocol typically begins with the prover committing to the VM's *execution trace*: this is effectively a list of everything that happened at each cycle of the VM's execution. The prover then applies a PIOP to prove that the committed execution trace is valid, meaning that it actually corresponds to the VM's execution on a witness.

1.1 The Prevailing (Recursive) Approach to Controlling Prover Memory

A widely used strategy to manage the zkVM provers' memory consumption is to *shard* the execution trace into smaller pieces (e.g., blocks of 1 million cycles) and prove each shard separately. These shard proofs are then combined using SNARK recursion or a "folding" procedure into a single, short proof. This approach allows the prover to store and prove only one or two shards at a time. In zkVM contexts, this approach is often called "continuations" or "recursive composition."

[1]Arithmic
[2]a16z crypto research and Georgetown University
[3]a16z crypto research

This technique does indeed reduce memory usage in each shard proof, but it also introduces various downsides. Below, we give a brief overview of the main challenges.

1.1.1 Verifier Implementation Complexity

Because each shard's proof depends on a commitment to the VM state at that shard's start, shards cannot be proved in complete isolation. The prover must ensure that the register and memory state at the end of shard i matches the state at the beginning of shard $i+1$. Existing work provides ways to handle this matching efficiently [AS24, ST25], but it remains an additional layer of complexity.

Even more fundamentally, using recursion means that the proof statement is no longer just "I ran this program for T steps correctly." Rather, it becomes "I hold multiple proofs (one per shard) that each shard was executed correctly from some committed state to the next." This makes it more difficult for external parties to see that the *intended* statement is being proven. It increases the surface area for bugs, complicates formal verification efforts, and makes the entire system harder to audit.

1.1.2 Security Considerations

SNARKs without recursion enjoy firmer theoretical underpinnings in certain idealized security models. For instance, unconditional security in the random oracle holds for some non-recursive SNARKs, but not for their recursive counterparts.[4,5] Furthermore, recursion typically requires taking a SNARK verifier, turning it into a circuit or constraint system, and then applying another SNARK to that circuit. These circuits are non-trivial to implement, particularly if one wants to keep their size small to mitigate performance bottlenecks. These circuits today are still often specified in extremely low-level domain-specific languages, a process that is error-prone, time-intensive, and difficult to verify for correctness.

Additionally, practical forms of recursion often rely on *algebraic* hash functions have seen less cryptanalysis than standard hash functions like SHA2 or SHA3.[6] Although ongoing efforts continue to strengthen confidence in these constructions, having the option to forgo recursion, and thus avoid the reliance on algebraic hashing altogether, remains appealing from both a security and implementation standpoint.

1.1.3 Performance Overheads

A third limitation stems from the cost of recursion itself. With brute-force recursion, each shard's proof is *verified* inside a larger SNARK, requiring the system to encode and prove a SNARK verifier circuit. Storing this verifier circuit already imposes nontrivial overhead on the prover in both time and memory.

There are approaches (especially using elliptic-curve-based commitments) that exploit homomorphisms to reduce some of this overhead. However, similar techniques for *hash-based* SNARKs remain costly in practice [BMNW24, KNS24, Sze24].

Another source of prover overhead in recursive approaches today is the issue raised in Section 1.1.1 of ensuring that the state of the VM's memory at the end of shard i is equal to its state at the start of shard $i+1$. Prominent projects currently pay large prover overheads to address this issue, with some using Merkle-hashing (causing the first read to each memory page in a given shard to cost over one thousand VM cycles[7]) and others resorting to collision-resistant multi-set hash functions whose evaluations involve elliptic curve group operations, many times more expensive than lightweight fingerprinting techniques that have long been used in non-recursive contexts [Gro24, RR25]. Although highly efficient methods are known for addressing this issue [AS24, BC24, ST25], these methods have not yet seen wide deployment.

[4] However, see Section 1.3.4 for additional discussion of idealized models and the (non-)implications of security therein.

[5] There have been some efforts to avoid heuristic security in folding-based recursion [LS24, EGS+24].

[6] Note this is not always the case. For example, if working over fields of characteristic two, very fast SNARKs are known for standard hash functions [DP23]. As another example, in folding schemes that use homomorphic commitments [KST22, KS24, BC23], hashing done in circuitry is a low-order cost, so using non-algebraic hash functions does not significantly alter prover time or space.

[7] See https://dev.risczero.com/api/zkvm/optimization#paging.

In summary, due to performance overheads arising from the use of SNARK recursion, it is possible that a non-recursive approach to controlling prover space ultimately leads to a faster or smaller space prover than recursive approaches. Our goal is to take a significant step toward making that possibility a reality (although realizing a high-performance implementation of our proving algorithms will still be a major engineering challenge).

1.1.4 Large Shards Can Lead to Faster Proving

A final issue, especially pertinent to this work, is that some SNARKs benefit heavily from the ability to consider large shard sizes, notably larger than what is typically used in zkVMs today (roughly 2^{20} CPU cycles). In particular, very recent work of Setty and Thaler introduces two highly efficient memory-checking arguments called *Twist* and *Shout* [ST25]. Twist applies to read/write memories and Shout to read-only memories.

In the context of continuations, let T denote the number of CPU cycles in a shard and K denote the size of the VM's memory. The Twist and Shout provers (in the context of continuations) incur a number of field multiplications linear in the memory size K. To ensure that this does not become a bottleneck, the shard size should be comparable to (or larger than) the size of the CPU's memory.

Thus, even if one is willing to use SNARK recursion, any zkVM that aims to support CPUs with, say, gigabytes of memory would benefit substantially from larger shard sizes—for example, on the order of 2^{30} rather than 2^{20} cycles. Existing zkVMs cannot handle this without requiring multiple terabytes of prover space.[8]

Moreover, certain commitment schemes that pair naturally with Twist and Shout have undesirable prover time bottlenecks when applied to small polynomials. For instance, when combining Twist and Shout with the Dory polynomial commitment scheme [Lee21], the prover performs a multi-pairing of size $O(\sqrt{T})$ for each shard of size T, in addition to more standard operations (e.g., MSMs of size about T). Because pairing evaluations are concretely expensive, the $O(\sqrt{T})$-sized multi-pairing can be a bottleneck when T is under about 2^{24}, but is a low-order prover cost when T is large (see [ST25, Sections 2.9.1 and 9.2.3] for details). This means Dory will be slower than alternative commitment schemes if applied to small shards, but just as fast as other schemes if shards are large.

Consequently, Twist and Shout are likely to perform best when shard sizes can be significantly larger than those typically used today. Hence, even if recursion *is* used, an approach that also controls space directly can improve prover time by allowing larger shards without ballooning memory usage.

A related observation is that, even if recursion and small shards are used (e.g., shards of size T equal to roughly one million cycles or less, as is popular today), prover memory will be minimized if each shard is proven using our techniques (i..e, with space sublinear rather than linear in T). Hence, our results will be useful in any context where prover space costs are paramount.

Summarizing all of the above, recursion-based sharding may not be the best approach to controlling prover memory usage. Instead, one can attempt to bound the prover's space costs *directly*. This unlocks simpler verification procedures, stronger security guarantees, and the ability to choose larger shard sizes, skip sharding entirely, or minimize prover space even if small shards are used.

1.2 Non-Recursive Small-Space Techniques

Controlling prover memory in sum-check-based SNARKs without resorting to recursion is not new. Early implementations of the sum-check protocol already demonstrated how the prover can operate with small space by streaming the witness, often incurring only logarithmic overhead in runtime [CTY11, CMT12, BTVW14]. As discussed above, this technique can also be used in conjunction with recursion to allow for larger shards.

[8] Ligetron [WHV24] is an exception, but it has a linear-time verifier, and is not based on the sum-check protocol (we believe that sum-check-based proof systems lead to the fastest possible provers).

Despite its promise, non-recursive sum-check-based techniques for reducing space have been underutilized in practice. We attribute this partly to the fact that sum-check-based SNARKs themselves have only relatively recently grown in popularity, as showcased by newer zkVMs such as Jolt [AST24], a zkVM that achieves state-of-the-art prover runtime and is built around the sum-check protocol.

A second reason may be that most memory-checking arguments require the prover to sort memory operations, and this sorting procedure is incompatible with small-space, non-recursive proving. However, memory-checking procedures that *avoid sorting* have long been known, and indeed Jolt uses such procedures.

A third potential reason is a belief that controlling prover space while avoiding recursion forces the prover to suffer a logarithmic-factor slowdown in runtime, as small-space implementations of the sum-check prover take time $O(T \log T)$ instead of $O(T)$ [CTY11, CMT12, BTVW14].

In this paper, we show that a large concrete slowdown is not necessary, i.e., that Jolt can be adapted to run in small space without incurring a crippling time penalty–indeed, our estimates suggest a modest slowdown of well under a factor of 2, rather than the naive $\log T$ factor that prior theoretical bounds imply (where in the zkVM context, T denotes the number of cycles in the VM execution that is being proved).

Note that we do not actually avoid an $O(T \log T)$ prover runtime (at least, not if we want prover space to be $T^{1-\Omega(1)}$). Hence, there is both a positive framing and a negative framing of our results. The positive framing is that we show the fastest known methods of implementing zkVM provers *already* are naturally small-space, and hence the prover can be made small-space "almost for free". The negative framing is that zkVM proving is currently so slow (i.e., requiring just $O(T)$ field operations to prove T cycles, but with an enormous constant hidden by the Big-Oh notation) that the prover time looks concretely more like $O(T \log T)$ rather than $O(T)$. In other words, one can make zkVM provers small-space with minimal time hit only because the prover is so slow to begin with.

Both framings have merit. In Section 7.1, we estimate that the Jolt prover (with Twist and Shout, and combined with an appropriate elliptic-curve commitment scheme), when applied to RISC-V executions with a memory size of $K = 2^{25}$, incurs roughly $900T$ field multiplications (900 field multiplications per VM cycle proved) when using space $O(T)$. Meanwhile, we estimate that a small-space implementation of the prover incurs about $900T + 12T \log T$ field multiplications. The key point is that $12T \log T$ is much less than $900T$ for realistic values of T. In other words, quasilinear time with a small leading constant is faster than linear time proving with a large leading constant. See Section 1.3.3 for additional discussion.

Two versions of Jolt: Spice+Lasso vs. Twist+Shout. Currently, Jolt handles read/write memory by using a protocol called Spice [SAGL18] and manages read-only memory (such as instructions) via Lasso [STW24]. A new version, under active development and nearing completion, will replace Spice with Twist and Lasso with Shout [ST25]. Twist and Shout are memory-checking protocols that promise even faster proving than Spice and Lasso (as well as shorter proofs). In both versions of Jolt, the entire proof system is built on the sum-check protocol.

We describe how both versions of Jolt—(1) Spice+Lasso and (2) Twist+Shout—can be made small-space without recursion. Interestingly, the Twist+Shout version is simpler to adapt and comes with lower overheads, thanks to its more streamlined memory-checking logic. The reasons we discuss both versions in this work are: (1) the Spice+Lasso version is already implemented, while the Twist+Shout version is still in progress; (2) the Twist+Shout version performs best when combined with either an elliptic curve commitment scheme (e.g., Dory or HyperKZG), a lattice-based commitment with similar properties to curve-based ones [NS25], or a hashing-based commitment schemes over binary fields (e.g., Binius or Blaze). The Lasso+Spice version of Jolt may still prove useful for projects that, for whatever reason, seek to work with other commitment schemes that operate over other fields than the above; (3) Exciting progress in lattice-based folding schemes [NS25] involve polynomial commitments with expensive evaluation proofs. This may force the use of relatively small shard sizes within the folding scheme. Spice and Lasso may yet prove superior to Twist and Shout in this small-shard scenario.

The "right" goal for space usage: logarithmic space is much lower than needed. Many works seek SNARK provers that, when proving correct execution of a computer program running in space K and time T, use space $O(K + \log T)$. In fact, we achieve this time bound in this paper (for some choices of the polynomial commitment scheme).

However, we argue that far weaker space bounds are sufficient in practice. This is for two reasons. First, the values of K and T that arise in prominent SNARK applications (e.g., zkEVMs, which seek to prove statements of the form "I correctly processed this Ethereum block") often have $K \geq 2^{25}$ (corresponding to dozens of MBs of space used by the program execution) and $T \leq 2^{35}$.[9] This means that $O(K + \log T)$ and $O(K + T^{1/2})$ are the same in practice up to constant factors: both are dominated by the $O(K)$ term. Second, depending on the hidden constant, $O(K + T^{1/2})$ space (or even higher) can already improve dramatically upon the concrete space usage of existing recursive approaches to controlling the zkVM prover space. Indeed, today's zkVMs require about 10 GB or more of prover space (at least if one is not willing to pay a substantial hit to prover time). Yet storing even $T^{1/2}$ field elements for $T = 2^{40}$ translates to just a few dozen MBs of prover space, orders of magnitude less than the roughly 10+ GBs used by recursion-based zkVMs today.

Although we achieve $O(K + \log T)$ prover space, allowing for higher space bounds can lead to a much simpler and faster prover implementation. Let us give an illustrative example. Consider Jolt with Twist and Shout, and Dory [Lee21] as the polynomial commitment scheme. Because the Dory commitment key consists of random group elements (which, in practice, are generated by evaluating a cryptographic PRG at different inputs), the commitment key can be generated "on the fly". But the Dory prover implementation is substantially simpler and *much* faster if the key is explicitly stored by the prover (see Section 1.3.1 for details).

When Dory is applied within Jolt using Twist and Shout, the commitment key consists of $2\sqrt{KT}$ group elements. (By using a more complicated variation of Twist, the commitment key can be reduced in size to $2\sqrt{K^{1/d} \cdot T}$ for any integer $d > 1$, but the Twist prover is fastest when $d = 1$).

Consider $K = 2^{25}$ and $T = 2^{35}$ as a representative instance. Storing $2\sqrt{KT}$ field elements costs about 100 GBs of space, which (while certainly much more space-intensive than desirable) is already acceptable in some settings. However, the size of the Dory commitment key can be lowered by a factor of 10 or more with a modest increase in verifier costs (see [ST25, Section 3.1.1] for details), and this would bring the commitment key size down to 10 GBs or less, which is roughly the amount of space used by recursion-based zkVMs today. Thus, we see that even $O(\sqrt{KT})$ space can be acceptable for the prover in practice. That is, while $O(T)$ space is too much, even modest improvements often suffice.

Another reason that a space bound of roughly $O(\sqrt{T})$ space is an attractive target is that certain meaningful optimizations to sum-check proving run in this space bound [DT24] (these techniques can more generally achieve a space bound of $T^{1/c}$ for any integer $c \geq 1$, but values of c larger than 2 lead to slower provers.)

Why the Jolt prover slowdown is smaller than a $\log T$ factor. Although prior theoretical analyses of streaming sum-check [CMT12, CTY11, BTVW14, BHR+20] suggest an $\mathcal{O}(T \log T)$ time bound for space-bounded provers and an $O(T)$ bound for a memory-intensive ones, several factors substantially reduce the prover time blowup in Jolt, concretely to a factor of well below 2 (per our analytic estimates in Section 7.1).

1. **Sparse sums for most of the protocol.** In Jolt (and especially in its Twist+Shout variant), the sum-check protocol is applied several times to sums over a large number of terms—yet most of these terms are zero. The prover employs "sparse techniques" dating back to [CTY11, CMT12] to handle these zero-terms efficiently, and these same sparse-sum optimizations *also* allow for small-space operation without extra overhead during the early rounds of sum-check. Only in the final rounds of the protocol do the relevant sums become "dense", and only then does the prover pay a time overhead for storing

[9] Today, a crude estimate for how many RISC-V cycles are required to prove a "heavy" Ethereum block is roughly 2^{30}. However, there are proposals to scale the amount of processing performed within each block by an order of magnitude or more [But25].

fewer items in memory. Since those "dense" rounds comprise only a fraction of the prover total work, the resulting overall slowdown for the prover is considerably less than a full factor of $\log T$.

2. **Some sum-checks are directly compatible with a streaming prover.** For some of the sum-check invocations in Jolt, something even stronger holds: these sum-check instances are amenable to the so-called *sparse-dense sum-check* proving algorithm from [STW24, Appendix G]. This protocol is inherently streaming, meaning the prover naturally runs in small space and very few passes over the values being summed, with essentially no slowdown needed to achieve this.

   In this work, we go a step further and describe a generalization of the sparse-dense sum-check protocol, which we call the *prefix-suffix inner product* protocol. This prover algorithm is a significant departure from that of [STW24, Appendix G], hewing much more closely to the standard linear-time sum-check prover algorithm [CTY11]. In a sense, the prefix-suffix inner product protocol identifies conditions under which the standard linear-time algorithm can actually be run in sublinear space.

   Even for sum-check invocations where the sparse-dense sum-check protocol does apply, the new prefix-suffix inner product protocol is simpler and concretely faster. More importantly for us, the generality of this new protocol let us render the prover in several of Jolt's sum-check invocations small space with no slowdown.

3. **"Small-value" sum-check is also directly compatible with a streaming prover.** We explain in Section 3.1.3 that certain techniques to speed up the Spartan prover as applied in Jolt (which leverage that all values arising in this Spartan instance are "small") [BDT24] are *also* compatible with a streaming prover, with little to no time overhead.

   To summarize the above three items, the fastest known ways of implementing the Jolt prover are already naturally space-controlled. For several sum-check invocations in Jolt, little to no overhead arises in making the prover small space. And for all of the others, time overheads arise only in late rounds of the protocol, and these rounds are only a small fraction of overall prover cost.

4. **Parallelism in repeated witness generation.** The small-space Jolt prover does have to generate the entire execution trace (the VM's state at each step) many times. In Jolt, this so-called *witness generation* procedure currently accounts for under 5% of the total prover time when done once, so doing it even several dozen times is not necessarily prohibitive. Moreover, we can exploit multithreading to dramatically lower the cost of repeated witness generation. Concretely, the first time the witness is generated (which is often done serially today, since arbitrary computer programs may not be parallelizable), the prover can store a small number of *checkpoints*—snapshots of the VM state at fixed intervals. This ensures that the prover can regenerate each chunk of the trace in parallel when needed. With M threads, this can yield up to a factor-M speedup compared to naively regenerating the witness from scratch in serial. This ensures repeated witness generation is only a modest additional cost for the prover.

5. **Stopping space-control early**. The space required by the standard linear-time sum-check proving algorithm [CTY11] halves in each round. Hence, after enough rounds have passed, the prover "switch" from a small-space prover implementation to the standard linear-time prover implementation, without increasing its space requirements. This saves a meaningful constant factor in prover time, while keeping the prover's space bounded by an small polynomial in the number T of VM cycles proven, e.g., $T^{1/2}$.

6. **No slowdown in committing or producing evaluation proofs.** With commitment schemes like Hyrax [WTS+18] and Dory [Lee21], if the prover is allowed space square root in the size of the committed polynomial, no slowdown whatsoever occurs compared to allowing a linear-space prover (see Section 6 for details).

Overview of the techniques. As discussed above, it's well-known that sum-check proving can be done in small space if the prover is capable of streaming the terms being summed (i.e., of iteratively generating the terms one after the next). So the bulk of our effort lies simply showing in that the Jolt prover is indeed able to stream the terms being summed in all relevant sum-checks.

For Twist and Shout, this is immediate from inspection of the efficient proving algorithms given in the paper [ST25]. For Spartan's application in Jolt, the main observation is simple: the primary sum-check invocation in Spartan requires the prover to stream the three vectors Az, Bz, and Cz, where here z is a solution to the R1CS instance. In Jolt, z is essentially an execution trace of the VM, and A, B, and C are essentially block-diagonal matrices (with constant-sized blocks, capturing under 100 constraints per CPU cycle). z can be streamed while executing the VM, and hence Az, Bz, and Cz can be streamed as well, due to the block-diagonal structure of A, B, and C. Intuitively, this simply means that every constraint in Jolt for cycle j of the VM's execution involves only those variables of z captures only what happens at cycle j, and there are only constantly many such variables. For the grand products in Lasso and Spice, our main observation is that, while streaming over inputs to the grand product (which are simply random fingerprints of values and counts or timestamps arising in reads or writes to memory performed during the VM's execution), the prover can also stream "partial products" of those inputs, via a depth-first traversal of the binary tree of multiplication gates computing those partial products.

For polynomial commitment schemes, we observe that for many polynomial commitment schemes the prover is directly implementable in space $O(\sqrt{T})$ where T is the size of the polynomial being committed. Hence, no time overhead whatsoever is required to implement the prover in this space bound. If space as low as $O(\log T)$ is desired, techniques from prior work [BHR+20] are easily adapted to a variety of polynomial commitments to achieve this while maintaining $O(T \log T)$ prover time.

Our new prefix-suffix inner product protocol, like the sparse-dense sum-check protocol that it generalizes [STW24], is used to compute $\sum_x u_x \cdot a_x$ where u_x is an arbitrary vector in $\mathbb{F}^n$ and a_x is a "structured" vector in $\mathbb{F}^n$. The primary insight in our new protocol is a cleaner formulation of what structure in a_x is exploitable to ensure a small-space prover that runs in linear time.

Key technical takeaways. Summarizing the above discussion, our work draws three key conclusions. First, each component of the Jolt prover (Spartan, Twist, and Shout) is "almost" streaming already. At worst, the only modification needed to render the prover small-space arises in the final $\log T$ rounds of the relevant sum-check instances, and simply invokes a small-space prover algorithm dating back to the first paper on linear-time and quasi-linear-time sum-check proving [CTY11]. Second, square-root rather than logarithmic prover space is often sufficient in practice, and is especially simple to achieve. In particular, many polynomial commitment schemes are directly implementable in space square root in the size of the committed polynomial. Third, asymptotic bounds on prover time do not necessarily match concrete bounds in practice. This is the case in our setting, where the $O(T \log T)$ term in our prover's runtime has a much smaller constant than the $O(T)$ term (by nearly two orders of magnitude).

1.3 Additional discussion

1.3.1 The high cost of repeatedly generating commitment keys

When considering transparent elliptic curve commitment schemes, if we want the prover to use space less than $O(T^{1/2})$, the prover does not have the space to store the entire commitment key. Hence, it has to generate the key "online" one or more times. What is the time cost of this procedure?

In practice, generating a random group element typically involves evaluating a cryptographic PRG and then applying a hash-to-curve procedure [Ham13, BCPR16, WJSS23]. A PRG evaluation takes only several CPU cycles per output byte produced. However, hash-to-curve procedures often require thousands of CPU cycles per curve element produced, which is the same order of magnitude as the cost of several dozen field operations over a 256-bit field $\mathbb{F}$. In fact, the bottleneck in state-of-the-art hash-to-curve procedures [Ham13] is computing a square root within $\mathbb{F}$, which asymptotically costs $O(\log |\mathbb{F}|) = O(\lambda)$ field operations.

This means that, with such commitment schemes, lowering the prover space to less than $O(\sqrt{T})$ in Jolt (e.g., achieving logarithmic space) requires more than $O(T \log T)$ field operations' worth of prover time. For example, when using (the simplest variation of) Hyrax [WTS+18] the prover's time becomes $O(T\lambda + T \log T) = O(T\lambda)$. When using commitment schemes like Dory [Lee21] that have logarithmic-round evaluation protocols, the prover has to generate elements of the commitment key $O(T \log T)$ times (see Section 6 for further details), and this is equivalent to $O(T \cdot \lambda \cdot \log T)$ field operations in total. Hence, repeatedly generating the commitment key to produce polynomial evaluation proofs is a significant prover time bottleneck. In practice, we do not believe this slowdown is worth the space savings. Instead, as we advocate for above, the prover should simply store the commitment key.

1.3.2 Committing to random values in Lasso and Spice

The grand product argument we use within Spice and Lasso is the one from Spartan/Quarks [Set20, SL20], which has the prover commit to T random values for grand products of size T. Using an elliptic curve commitment scheme, this requires $O(T\lambda/\log(T))$ field operations where λ is the security parameter. For values of λ arising in practice (e.g., $\lambda = 128$), this is comparable to $O(T \log T)$ time (though in practice committing to all these random values is a concrete prover bottleneck in Jolt when the Spartan/Quarks grand product is used rather than alternatives such as Thaler's grand product argument [Tha13] that are faster for the prover but have larger proofs). For simplicity, we state an asymptotic $O(T \log T)$ runtime bound for the Lasso and Spice provers.

1.3.3 Is Jolt with Twist and Shout linear time?

With a commitment scheme such as Dory, the Jolt prover always devotes $O(T)$ field operations to committing to data and producing evaluation proofs. The remaining components of Jolt are the various sum-check protocol instances in Spartan, Twist, and Shout. The linear-space prover implementations for Spartan and Shout run in $O(T)$ time. The RISC-V VM has 32 registers, at most two of which are read and one written cycle. Since the number of registers is a constant, When Twist is applied to process these register reads and writes, the prover runs in $O(T)$ time. The only component of linear-space Jolt proving that potentially requires superlinear time is Twist applied to the VM's reads and writes to RAM. In the worst case this takes time $O(T \log K)$. Unless the memory size K is modelled as a constant independent of T, $O(T \log K)$ is superlinear in T.[10]

However, there are two reasons why the Jolt prover's runtime can in fact be $O(T)$ for "typical" program executions even when using Twist. First, the Twist prover only "pays" field operations for cycles that actually perform a read or write to RAM. Second, for memory accesses that are "i-local" (accessing memory cells that were previously accessed at most 2^i cycles prior), the Twist prover only pays $O(i)$ field operations. Accordingly, with Twist and Shout, the linear-space Jolt prover performs $\Theta(T \log K)$ field operations only when a constant fraction of cycles perform reads or writes to RAM *and* those memory operations are not sufficiently local.

In summary, the prover for Jolt with Twist and Shout can be considered linear-time for many program executions. And since the runtime of our $O(\sqrt{KT})$-space proving algorithm is concretely within a small constant factor of the linear-space prover's time, the small-space prover's runtime concretely behaves in a similar fashion, even though it is asymptotically $O(T \log T)$.

1.3.4 Attacks on the Fiat-Shamir transformation

Recently, attacks have been demonstrated [KRS25] on proof systems that combine the GKR protocol [GKR15, CMT12, Tha13] (plus any multilinear polynomial commitment scheme) with the Fiat-Shamir transformation [FS87]. Let us refer to the attackable protocol as FS-GKR. Currently, these attacks are only known to work

[10] If using Spice with GKR-based grand products [Tha13] in place of Twist, the Jolt prover's runtime *is* $O(T)$ field operations in the worst case, regardless of the memory size K. But Spice is concretely slower for the prover than Twist [ST25], and moreover this variant of Spice also leads to bigger proofs (of size $O(\log^2 T)$) than with Twist.

when FS-GKR is applied to a specific class of circuits, key components of which must be deterministic (i.e., they must take no "untrusted advice"). This causes the circuits to be large and deep. From a performance standpoint, this is precisely the class of circuits for which the GKR protocol is *un*attractive, as it translates to large proofs and a slow prover.

Despite these caveats, the attacks raise concerns about the viability of the Fiat-Shamir paradigm in general (i.e., of designing SNARKs by first designing a public-coin interactive protocol, and then removing interaction via the Fiat-Shamir transformation). The aforementioned attacks on FS-GKR apply even though the protocol is *unconditionally secure in the random oracle model*, due to the difference between the random oracle idealization and actual hash functions. Understanding and mitigating such attacks (possibly by altering the FS-transformation itself [AY25]) remains an extremely important and active area of research.

Arguably, the circuits for which FS-GKR has been shown to be attackable are reminiscent of those used in recursive proving. By this, we mean that the circuit, to "make itself attackable" when FS-GKR is applied to it, performs the same computation that FS-GKR verifier does.[11] This offers further motivation for our work, as we focus on controlling prover memory usage *without* resorting to SNARK recursion.

Given the above attacks, it is worth clarifying the role of the GKR protocol within Jolt. The GKR protocol is used in Jolt-with-Spice-and-Lasso, but it is only applied to "grand product circuits" that are vastly simpler than those known to be attackable. Jolt-with-Twist-and-Shout does not invoke the GKR protocol at all.

1.4 Related Work

Cormode, Thaler, and Yi (CTY) [CTY11] gave two different algorithms for implementing the sum-check prover when the sum-check protocol is applied to a product of $\log(T)$-variate multilinear polynomials. One performs $O(T)$ field multiplications but requires space $O(T)$. We call this the *standard linear-time sum-check proving algorithm*. The other proving algorithm in CTY performs $O(m \log T)$, where m is the number of terms of the sum that are non-zero–we refer to m as the *sparsity* of the sum, and refer to this proving algorithm as the *sparse* proving algorithm. The $O(m \log T)$ time bound arises because the prover spends time $O(m)$ in *each* of the $O(\log T)$ rounds (whereas in the standard linear-time proving algorithm, the prover's time in round i is $O(T/2^i)$).

The sparse proving algorithm requires just $O(\log T)$ space if the prover has streaming access to the terms of the sum (i.e., if the prover can generate one term after the next in small space). We distill the sparse proving algorithm for the sum-check protocol in Section 3.1.

Somewhat confusingly, the work of CTY was framed around the notion of "streaming interactive proofs", where streaming here refers to the *verifier* processing the input in a streaming fashion. An unrelated (but historically more important) contribution of the same work was to describe the sparse sum-check prover algorithm (which is small-space for the prover if the prover is capable of generating the terms being summed in a streaming fashion). Thus, CTY achieved a "streaming verifier" *and* (in a different sense) also implicitly achieved a "streaming prover" for the sum-check protocol.

The sparse sum-check prover algorithm was subsequently applied [CMT12] to achieve a quasilinear-time prover for circuit evaluation (later improved to linear time via a series of works [Tha13, WJB+17, XZZ+19, Set20]), and in [BTVW14] to explicitly achieve a small-space, quasilinear time proving algorithm for circuit satisfiability. [BTVW14] was presented as a two-prover interactive proof, but from a modern perspective it implicitly gives a polynomial IOP (PIOP), as the second prover in the two-prover proof system plays the same role as a polynomial commitment scheme.

[BHR+20] construct a SNARK for virtual machines (VMs) with provers operating in quasilinear time and small space (linear in the space usage of the VM itself, plus a term that is logarithmic in the VM's runtime). They work in the random oracle model, assuming the hardness of the discrete logarithm problem in prime-order groups. A key technical contribution is to identify a "streaming-prover" implementation of the Bulletproofs/IPA polynomial commitment scheme [BCC+16, BBB+18] to combine with the small-space PIOP of Blumberg et al. [BTVW14].

[11] Though unlike actual recursion circuits, the circuit for which FS-GKR is attackable also has to deterministically compute a cryptographic commitment to the witness that it is fed.

A downside of the Bulletproofs PCS is that it requires (super-)linear time verification. [BHR+21] improve the verifier in [BHR+20] by leveraging the PCS from [BFS20], which features a sublinear verifier algorithm for the evaluation protocol. They further devise a small-space commitment algorithm and a small-space evaluation prover algorithm for the PCS in [BFS20]. This allows them to retain the PIOP from [BHR+20, BTVW14] while replacing its PCS with the one from [BFS20], achieving sublinear verifier complexity.

Gemini [BCHO22] reestablishes the result of Block et al. [BHR+20] regarding small-space, quasilinear-time proving for VM execution, though with a multilinear analog of the KZG PCS [KZG10] in place of the Bulletproofs/IPA PCS. As with KZG commitments themselves, the structured reference string for this PCS is of linear size, so the prover is only small space if this SRS can be "streamed" by the prover. Gemini more generally gives SNARKs for rank-one constraint systems where the prover may make many streaming passes over the witness and associated constraint evaluations.

Ligetron [WHV24] is a SNARK (*not* based on the sum-check protocol) for WASM execution, where the prover runs in quasilinear time and space square root of the number of gates in a circuit required to implement the WASM execution. The proof size is also square root in the number of gates. Ligetron involves a linear-time verifier.

Epistle [ZCL+24] introduces SNARKs with a small-space prover tailored for Plonkish constraint systems. As part of this work, they design SNARK for grand products, which verifies that the product of all elements in a committed vector equals 1. Their prover operates within a small-space regime, but their approach differs from ours as it does not rely on the sum-check protocol. Their technique constructs two vectors containing the partial products of the witness vector, where one vector is a cyclic shift of the other, and the final component of one vector encodes the product value. They then employ protocols to verify the cyclic relationship between these vectors, along with polynomial identities that enforce the correctness of the partial product computations. This method introduces higher overheads than our approach in Section D.

Sparrow [PP24] presents a SNARK tailored to data-parallel circuits. In FFT-friendly fields, their approach reduces the prover's space complexity to approximately the square root of the circuit's size, while the prover's runtime is superlinear by only a doubly-logarithmic factor. The main technique in [PP24] is as follows. Consider the sum-check protocol applied to a product of two multilinear polynomials over $\log T$ variables. By increasing the degree of both polynomials in their first two variables from one to, respecitvely, $\log T$ and $\sqrt{T}/\log T$, the sparse sum-check prover winds up spending slightly superlinear time in the first two rounds but very little time in subsequent rounds. Moreover, the prover's space stays bounded by roughly $O(\sqrt{T})$.

In Scribe, [BMM+24] consider SNARKs for circuit satisfiability. The allow the prover to use linear space in the size T of the circuit, but limit the prover's access to this large memory to read/write streaming passes. This means that the prover can repeatedly read and overwrite the entire memory from start to finish. They adapt the sum-check-based Hyperplonk PIOP [CBBZ23] to this setting, as well as several polynomial commitment schemes. Our work focuses on zkVMs rather than circuit-satisfiability (effectively, we focus on a specific "circuit" that captures execution of the RISC-V VM). In this context, we avoid linear space for the prover.

Throughout Scribe [BMM+24], the authors describe various "barriers to read-only streaming". A brief description of these issues and how they relate to our work is warranted. First, several of the barriers amount to characterizing $O(T \log T)$-time prover algorithms as too slow. But, as discussed in Section 1.2 above and Section 7.1 below, for zkVMs (a setting not considered in Scribe), $O(T \log T)$-time prover algorithms can be within a small constant factor (potentially well below $2\times$) slower than state-of-the-art linear-space proving algorithms. Scribe also describes various issues arising with a depth-first traversal of a binary tree of multiplication gates in grand product arguments (see [BMM+24, Remark 2.7] and [BMM+24, Section 2.8.1]). Our result about making grand products streaming for the prover (Appendix D) uses exactly this approach, showing that there is no fundamental issue with a depth-first traversal order, at least not if $O(T \log T)$ prover runtime is acceptable.[12]

[12]Scribe also describes concerns related to achieving a fast streaming prover for polynomial commitment schemes exploiting vector-

1.5 Paper Roadmap

We organize the remainder of this paper as follows (throughout, we use the term "streaming prover" as shorthand for a prover that runs in space sublinear in T).

- Section 3 contains relevant background on the sum-check protocol and its efficient implementation, especially the "small-value sum-check" prover implementation of [BDT24], the design of the Jolt zkVM, and the Twist and Shout memory-checking arguments.
- Section 4 presents the streaming implementation of the small-value sum-check protocol prover [BDT24] in the context of Jolt's use of Spartan applied to the R1CS instance arising in Jolt.
- Section 5 explains the streaming implementations of the Twist and Shout provers.
- Section 6 covers a handful of polynomial commitment schemes and how to render their prover streaming.
- Section 7 shows how to integrate these results into Jolt's three main components (Shout, Twist, and Spartan) to build a memory-efficient Jolt prover.
- Appendix A describes our new *prefix-suffix inner product* sum-check prover algorithm.
- Appendices B-E cover Spice and Lasso (the memory-checking arguments used in the Jolt implementation today) and explain how to render the prover in this version of Jolt streaming.

2 Preliminaries

We use $\mathbb{F}_p$ to denote a finite field of size p. In our runtime accounting, we regard an operation in $\mathbb{F}_p$ as one time step, and a field element as requiring one unit of memory. Note that for Jolt to achieve λ bits of security, it must work over a field $\mathbb{F}_p$ of size at least 2^λ. When the field size p is clear from the context, we omit it and simply write $\mathbb{F}$. Unless stated otherwise, $\mathbb{F}$ is a large prime-order field, though we also sometimes consider the binary field $\mathrm{GF}[2^{128}]$.

We use tobits to denote the function that takes a value in $\{0, 1, \ldots, 2^n - 1\}$ and returns the n-bit binary string corresponding to it. Similarly, we use val to denote the function that takes an n-bit binary string and returns the integer value corresponding to it. That is, $\mathrm{val}(b_1, \ldots, b_n) = \sum_{i=1}^{n} 2^{i-1} \cdot b_i$, and $\mathrm{tobits}(\mathrm{val}(b_1, \ldots, b_n)) = (b_1, \ldots, b_n)$. Note that we display the low-order bit b_1 on the left and the high-order bit b_n on the right.

Given a function $f : \{0,1\}^n \to \mathbb{F}$, its *multilinear extension* (MLE) is the polynomial $\widetilde{f}(X_1, X_2, \ldots, X_n)$ defined as follows:

$$\widetilde{f}(X_1, X_2, \ldots, X_n) = \sum_{x \in \{0,1\}^n} f(x) \prod_{i=1}^{n} ((1 - X_i)(1 - x_i) + X_i \cdot x_i),$$

where x_i denotes the i-th bit of x. It is easy to check that $\widetilde{f}$ is the unique multilinear polynomial such that $\widetilde{f}(y) = f(y)$ for all $y \in \{0,1\}^n$.

Given a vector $\mathbf{w} \in \mathbb{F}^{2^n}$, the MLE $\widetilde{\mathbf{w}}$ corresponding to $\mathbf{w}$ is the multilinear polynomial that evaluates to $\mathbf{w}$ over the boolean hypercube. In other words, $\widetilde{\mathbf{w}}(\mathrm{tobits}(i)) = w_i$, for $i \in \{0, 1, \ldots, 2^n - 1\}$, where w_i is the i-th component of $\mathbf{w}$.

The following basic fact about multilinear polynomials will be useful to us:

matrix-vector-product structure in polynomial evaluation queries. The concern relates to the prover needing to iterate over (a matrix representation of) the polynomial's evaluations in both row-major and column-major order. However, the precise nature of the barrier is unclear, and in Section 6 of our work, we do give a small-space prover implementation for these commitment schemes.

Fact 2.1. Let $\widetilde{u}\colon \mathbb{F}^n \to \mathbb{F}$ be any multilinear polynomial. Then for any $c \in \mathbb{F}$ and $x \in \mathbb{F}^{n-1}$,

$$\widetilde{u}(c, x) = (1 - c) \cdot \widetilde{u}(0, x) + c \cdot \widetilde{u}(1, x). \tag{1}$$

Proof. The left hand and right hand sides of Equation (1) are both multilinear polynomials in (c, x), and clearly are equal when $(c, x) \in \{0,1\} \times \{0,1\}^{n-1}$. The claim then follows from the fact that $\{0,1\}^n$ is an interpolating set for n-variate multilinear polynomials over $\mathbb{F}$. □

We define the multilinear extension of the *equality function* over two boolean strings $x, y \in \{0,1\}^n$ as follows:

$$\widetilde{\mathrm{eq}}(X, Y) = \prod_{i=1}^{n} \left((1 - X_i)(1 - Y_i) + X_i Y_i\right).$$

It is easy to verify that for any pair of strings $x, y \in \{0,1\}^n$, $\widetilde{\mathrm{eq}}(x, y)$ equals 1 if $x = y$ and equals 0 otherwise.

2.1 Succinct Non-Interactive Arguments of Knowledge

A Succinct Non-interactive ARgument of Knowledge for a relation $\mathcal{R}$ enables any prover to convince a verifier that it possesses a (private) witness w corresponding to publicly known x such that $(x, w) \in \mathcal{R}$. In this section, we formally define an Argument of Knowledge, and then state the required succinctness property. Given a pair of probabilistic interactive algorithms P, V, we denote $\langle P, V \rangle (x; w)$ as the output of the interaction of P and V, where x is publicly known, and w is privately known to P. Let $\mathcal{R} = \{(x, w)\}$ be a relation. Next, we formally define Arguments of Knowledge (AoK).

Definition 2.2 (Succinct Argument of Knowledge)**.** An Argument of Knowledge (AoK) for a relation $\mathcal{R}$ constitutes of a probabilistic algorithm $\mathsf{G}(1^\lambda)$ that takes as input the security parameter λ and outputs public parameters pp, together with a pair of probabilistic algorithms $\langle P, V \rangle$ satisfying:

1. Completeness: For all $\lambda \in \mathbb{N}$, $\mathsf{pp} \leftarrow \mathsf{G}(1^\lambda)$, and for every $(x, w) \in \mathcal{R}$ the following holds:

$$\Pr\left\{\langle P, V \rangle (\mathsf{pp}, x; w) = 1\right\} = 1$$

2. Knowledge-Soundness: For any pair PPT algorithms P_1, P_2 there exists an PPT algorithm Ext such that the following holds:

$$\Pr\left\{\begin{array}{c} (x, w) \notin \mathcal{R} \\ \langle P_2, V \rangle (\mathsf{pp}, x; st) = 1 \end{array} \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{G}(1^\lambda),\ (x, \mathsf{st}) \leftarrow P_1(\mathsf{pp}, 1^\lambda) \\ w \leftarrow \mathrm{Ext}^{P_2}(\mathsf{pp},) \end{array}\right\} \quad = \quad \mathsf{negl}(\lambda)$$

Remark 2.3. The AoK is succinct if the communication complexity between the prover and the verifier, and the verifier run-time is sub-linear in the size of the witness w.

Remark 2.4. An AoK is public coin if the random bits sampled by the verifier is public. A Succinct Non-interactive Argument of Knowledge (SNARK) is a one-message public-coin AoK, that is, the prover sends the message to the verifier and consequently verifier outputs either 1 or 0. An interactive public-coin argument can be rendered non-interactive using the Fiat-Shamir transformation. If the interactive argument satisfies an enhanced soundness property called round-by-round soundness [CCH+19], the SNARK obtained via the Fiat-Shamir transformation is knowledge sound in the Random Oracle Model (ROM) (augmented with any additional cryptographic assumptions upon which the security of the interactive argument is based).

2.2 Polynomial Commitment Scheme

A polynomial commitment scheme (PCS) enables a prover to succinctly demonstrate the correctness of polynomial evaluations ([KZG10]). A PCS over $\mathbb{F}$ consists of the tuple $\mathsf{PC} = (\mathsf{setup}, \mathsf{commit}, \mathsf{open}, \mathsf{eval})$, where:

- $\mathsf{pp} \leftarrow \mathsf{setup}(1^\lambda, n, N)$. Given a security parameter λ, the number of variables n, and an upper bound $N \in \mathbb{N}$ on the number of distinct monomials in n variables, setup generates the public parameters pp.
- $(C, \tilde{\mathbf{c}}) \leftarrow \mathsf{commit}(\mathsf{pp}, f, D)$. Given the public parameters pp, an n-variate polynomial $f(X) \in \mathbb{F}[X]$ with at most $D \leq N$ monomials, commit outputs a commitment C to the polynomial along with an opening hint $\tilde{\mathbf{c}}$.
- $b \leftarrow \mathsf{open}(\mathsf{pp}, f(X), D, C, \tilde{\mathbf{c}})$. Given the public parameters pp, commitment C, opening hint $\tilde{\mathbf{c}}$, and a polynomial $f(X)$ with at most $D \leq N$ monomials, open outputs a binary decision indicating whether the proof is valid.
- $b \leftarrow \mathsf{eval}(\mathsf{pp}, C, D, \mathbf{x}, y; f(X))$. This is a public-coin interactive protocol $\langle P_{\mathsf{eval}}, V_{\mathsf{eval}} \rangle (\mathsf{pp}, C, D, \mathbf{x}, y; f(X))$ between a PPT prover and a PPT verifier. The common input consists of the public parameters pp, the commitment C, the monomial bound D, an evaluation point $\mathbf{x} \in \mathbb{F}^n$, and the claimed evaluation y. The prover additionally has access to the opening f of C with at most D monomials. The verifier outputs 1 if the proof is accepted (i.e., confirming $f(\mathbf{x}) = y$), or outputs 0 otherwise.

A polynomial commitment scheme must satisfy the properties of completeness, binding, and knowledge soundness.

Definition 2.5 (Completeness)**.** For any polynomial $f(X) \in \mathbb{F}[X]$ with at most $D \leq N$ monomials, and for any $\mathbf{x} \in \mathbb{F}^n$,

$$\Pr\left[b = 1 \ : \ \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{setup}(1^\lambda, N) \\ (C, \tilde{\mathbf{c}}) \leftarrow \mathsf{commit}(\mathsf{pp}, f(\mathbf{X}), D) \\ y \leftarrow f(\mathbf{x}) \\ b \leftarrow \mathsf{eval}(\mathsf{pp}, C, D, \mathbf{x}, y; f(X)) \end{array} \right] = 1.$$

Definition 2.6 (Binding)**.** A polynomial commitment scheme PC satisfies binding if for every PPT adversary $\mathcal{A}$, the probability of the following event occurring is negligible in λ:

$$\Pr\left[\begin{array}{c} \mathsf{open}(\mathsf{pp}, f_0, D, C, \tilde{\mathbf{c}}_0) = 1 \wedge \\ \mathsf{open}(\mathsf{pp}, f_1, D, C, \tilde{\mathbf{c}}_1) = 1 \wedge \\ f_0 \neq f_1 \end{array} \ : \ \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{setup}(1^\lambda, N) \\ (C, f_0, f_1, \tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1, D) \leftarrow \mathcal{A}(\mathsf{pp}) \end{array} \right].$$

Definition 2.7 (Knowledge Soundness)**.** The protocol eval serves as an argument of knowledge (AoK) for the relation $\mathcal{R}_{\mathsf{eval}}$, defined as:

$$\begin{aligned} \mathcal{R}_{\mathsf{eval}} = \{ ((\mathsf{pp}, C, \mathbf{x} \leftarrow_R \mathbb{F}^n, \ y \in \mathbb{F}); \ (f(X), \tilde{\mathbf{c}})) : \\ (\mathsf{open}(\mathsf{pp}, f, D, C, \tilde{\mathbf{c}}) = 1) \wedge y = f(\mathbf{x}) \} \end{aligned}$$

3 Background

3.1 The Sum-Check Protocol

The sum-check protocol is designed to verify the sum of evaluations of a given n variate polynomial g over the Boolean hypercube $\{0,1\}^n$. In Jolt, the sum-check protocol is applied specifically to cases where g is

expressed as a product of ℓ multilinear polynomials.[13] Specifically, there exists ℓ multilinear polynomials $g_1, g_2, \ldots, g_\ell$ in n variables such that

$$g(X_1, \ldots, X_n) = \prod_{k \in \{1,\ldots \ell\}} g_k(X_1, \ldots, X_n) \tag{2}$$

In this scenario, the sum-check protocol is employed to prove the following relation:

$$\{(v, \{C_{g_k}\}_{k\in\{1,\ldots,\ell\}}) \mid C_{g_k} \text{ is an oracle to an } n\text{-variate multilinear polynomial } g_k, \text{ for } k \in \{1, \ldots, \ell\}, \text{ and } v = \sum_{x\in\{0,1\}^n} g(x), \text{where } g \text{ is as in Equation (2) } \}. \tag{3}$$

The sum-check protocol operates over n rounds, and in each round the prover sends a univariate polynomial $f_i(X_i)$ to the verifier, who then responds with a random value r_i. The polynomial $f_i(X_i)$ is defined as follows:

$$f_1(X_1) = \sum_{x\in\{0,1\}^{n-1}} g(X_1, x), \text{ and } f_i(X_i) = \sum_{x\in\{0,1\}^{n-i}} g(r_1, \ldots, r_{i-1}, X_i, x) \ \forall i \in \{2, \ldots, n\}.$$

In the first round, the verifier checks $v = f_1(0) + f_1(1)$, and for every round $i \in \{2, \ldots, n-1\}$ after that the verifier checks $f_i(r_i) = f_{i-1}(1) + f_{i-1}(0)$. At the end of the protocol, the verifier additionally checks $g(r_1, \ldots, r_n) = f_n(r_n)$. To accomplish this, the verifier evaluates $g_1(r_1, \ldots, r_n), \ldots, g_\ell(r_1, \ldots, r_n)$ by making a single query to each of their respective oracles and then uses these results to compute

$$g(r_1, \ldots, r_n) = \prod_{k=1}^{\ell} g_k(r_1, \ldots, r_n).$$

It is well-known that the sum-check protocol has soundness error at most $\frac{\ell \cdot n}{|\mathbb{F}|}$ (see Section 4.1 from [Tha22] for a formal statement and a proof of this). The sum-check protocol allows the verifier to reduce the task of checking the relation in Equation (2) to verifying the evaluationg g at a random point $(r_1, \ldots, r_n)$.

There is a well-known linear-time honest prover algorithm for the sum-check protocol from [CTY11]. We note the complexity of this honest prover algorithm in Theorem 3.1.

Theorem 3.1 ([CTY11])**.** *There is an honest prover algorithm for the sum-check protocol that takes as input the evaluations of $g_1, \ldots, g_\ell$ over $\{0,1\}^n$ and in each $i \in \{1, \ldots, n\}$ computes $f_i(X_i)$ in time and space $O(2^{n-i})$.*

Thus, for an honest prover, the sum-check protocol incurs at most a constant-factor overhead beyond computing the witness polynomials $g_1, \ldots, g_\ell$. However, the linear-time honest prover algorithm also requires linear space, which becomes a bottleneck when the witness polynomials themselves are large, as is the case in Jolt. In Section 3.1.2, we distill results from [CTY11, CMT12, BTVW14] and present an honest prover algorithm for the sum-check protocol that trades off runtime to operate in logarithmic space. Specifically, Algorithm 1 in Section 3.1.2 provides an honest prover algorithm for the sum-check protocol that operates in $O(n)$ space. However, it requires the terms being summed to be recomputed iteratively in each round of the sum-check protocol. This results in linear prover time per round, and hence a logarithmic factor overhead compared to the linear-time honest prover algorithm in Theorem 3.1.

Finally, in Section 3.1.3, we demonstrate how recent honest prover algorithms for the sum-check protocol, which have been optimized for the case where all terms being summed are "small", can be adapted (with no time overhead) to operate in small space as well.

[13] The one exception is a sum-check used in Lasso to coalesce the results of several lookups into smaller tables into the final result of the original lookup. Even there, the relevant polynomial can be decomposed into a constant number of sums of products of multilinear polynomials. Hence, describing g as a product of multilinear polynomials is general enough to capture all applications of the sum-check protocol in Jolt.

3.1.1 Overview of Linear-Time Sum-Check Prover Algorithm

Our starting point is the linear time honest prover algorithm from [CTY11, Tha13]. The algorithm accepts as input the evaluations of $g_1, g_2, \ldots, g_\ell$ over $\{0,1\}^n$. For $k \in \{1, \ldots, \ell\}$ let A_k represent the evaluations of g_k, where the j-th entry of A_k is the evaluation of g_k at tobits(j), for $j \in \{0, 1, \ldots, 2^n - 1\}$. In round i of the sum-check protocol the prover sends the univariate polynomial $f_i(X_i)$ to the verifier, who responds with a random point $r_i \in \mathbb{F}$. The polynomial f_i is constructed by evaluating it at $\ell + 1$ points.

(In fact, standard optimizations reduce the number of evaluation points to ℓ, and in many cases just $\ell - 1$. The reduction from $\ell + 1$ evaluations to ℓ is possible because for the honest prover, $f_i(0) + f_i(1) = f_{i-1}(r_{i-1})$, and so $f_i(1)$ can be directly inferred from $f_{i-1}(r_{i-1})$ and $f_i(0)$. The reduction from ℓ to $\ell - 1$ evaluations is due to Gruen [Gru24, Section 3] and applies when $g_1(x) = \widetilde{\text{eq}}(r, x)$ for some $r \in \mathbb{F}^n$. For simplicity we ignore these optimizations in this work.

The evaluations of f_i at the relevant points is computed by using the evaluations of $g_k(r_1, \ldots, r_{i-1}, X_i, \ldots, X_n)$ over $\{0,1\}^n$ for $k \in \{1, \ldots, \ell\}$. The prover updates the arrays $A_1, \ldots, A_\ell$ such that at the beginning of round i, A_k contains the evaluations of $g_k(r_1, \ldots, r_{i-1}, X_i, \ldots, X_n)$ over $\{0,1\}^{n-i}$. This update step works as follows. At the end of round $i-1$, the array A_k is updated and its size is reduced by a factor of two as follows:

$$A_k[m] = (1 - r_{i-1}) \cdot A_k[2m] + r_{i-1} \cdot A_k[2m+1], \quad \text{for } m \in \{0, 1, \ldots, 2^{n-(i-1)}\}. \tag{4}$$

Thus, at the beginning of round i the size of A_k is $2^{n-(i-1)}$, for $k \in \{1, \ldots, \ell\}$ and $i \in \{1, \ldots, n\}$. It is straightforward to verify that this update step ensures A_k contains the evaluations of $g_k(r_1, \ldots, r_{i-1}, X_i, \ldots, X_n)$ at the start of round i. This follows from the fact that for any n-variate multilinear polynomial g_k and any $x \in \mathbb{F}^{n-1}$ and $r \in \mathbb{F}$, it holds that $g_k(r, x) = (1 - r) \cdot g_k(0, x) + r \cdot g_k(1, x)$. Clearly, since the arrays A_k at the beginning store the evaluations of g_k over $\{0,1\}^n$, the algorithm requires $O(k \cdot 2^n)$ space. Next, we describe the logarithmic-space algorithm.

3.1.2 Sum-Check Prover in Logarithmic Space

In this section, we present Algorithm 1, which provides an honest prover algorithm for the sum-check protocol that operates in logarithmic space. We assume that the prover has oracle access to $g_1, \ldots, g_\ell$ and iteratively queries their evaluations over $\{0,1\}^n$. Specifically, the prover queries the evaluations of each g_k in the sequence: tobits(0), tobits(1), $\ldots$, tobits$(2^n - 1)$. For all invocations of the sum-check protocol in Jolt, the evaluations of $g_1, \ldots, g_\ell$ can be sequentially enumerated in this manner by simply executing the computer program whose execution is being proven. That is, there is a witness generation algorithm that computes each evaluation of g_k in $O(1)$ time and logarithmic space (on top of the space K needed simply to run the VM), in the following order: $g_k(\text{tobits}(0))$, $g_k(\text{tobits}(1))$, $\ldots$, $g_k(\text{tobits}(2^n - 1))$.

Let $S = \{\alpha_0, \ldots, \alpha_\ell\}$ be a set of $\ell + 1$ distinct points. At round i of the sum-check protocol, the prover needs to evaluate f_i at all points in S. To this end, the prover computes each summand of the following expression

$$f_i(\alpha_s) = \sum_{m=0}^{2^{n-i}-1} \prod_{k=1}^{\ell} g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m)).$$

Algorithm 1 computes the summand in the expression corresponding to each $m \in \{0, \ldots, 2^{n-i} - 1\}$, which is given by

$$\prod_{k=1}^{\ell} g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m)).$$

Assume for now at Steps 18-20, $\prod_{k \in \{1,\ldots,\ell\}} g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ is computed correctly; we will justify this in the next paragraph. In Step 19, this term is added to $\mathit{accumulator}[s]$ for $s \in \{0, \ldots, \ell\}$. Here $\mathit{accumulator}[s]$ serves as a variable to accumulate the terms so that at Step 23 we have, $f_i(\alpha_s) = \mathit{accumulator}[s]$. Assuming $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ is correctly computed correctly at Step 16 the correctness of the algorithm follows. Next we show how $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ is computed in $O(1)$ space in Steps 4-16.

For a clearer presentation, we denote $A_{k,i}$ as the state of the array A_k at the beginning of round i in the linear time honest prover algorithm from Section 3.1.1. We make the following observation regarding the entries of $A_{k,i}$ for $k \in \{1,\ldots,\ell\}$ and $i \in \{1,\ldots,n-1\}$ (proof in Appendix G).

Claim 3.2. For $i \in \{1,\ldots,n-1\}$ and $m \in \left\{0,\ldots,2^{n-(i-1)}-1\right\}$, the m-th entry of $A_{k,i}$ is equal to

$$\sum_{j \in \{0,\ldots,2^{i-1}-1\}} \widetilde{\mathrm{eq}}(r_1,\ldots,r_{i-1},\mathrm{tobits}(j)) \cdot A_k[2^{i-1}\cdot m + j]\,. \tag{5}$$

From Claim 3.2 we have $g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m))$ is equal to

$$\begin{aligned} g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m)) =& (1-\alpha_s)\cdot\Big(\sum_{j\in[0,2^{i-1}-1]} \widetilde{\mathrm{eq}}(r_1,\ldots,r_{i-1},\mathrm{tobits}(j))\cdot A_k[2^{i-1}\cdot(2m)+j]\Big) \\ &+ \alpha_s\cdot\Big(\sum_{j\in[0,2^{i-1}-1]} \widetilde{\mathrm{eq}}(r_1,\ldots,r_{i-1},\mathrm{tobits}(j))\cdot A_k[2^{i-1}\cdot(2m+1)+j]\Big) \end{aligned} \tag{6}$$

Here $m \in \{0,\ldots,2^{n-i}-1\}$, $k \in \{1,\ldots,\ell\}$, and $s \in \{0,\ldots,\ell\}$. In Algorithm 1, the primary strategy is to avoid storing the intermediate arrays $A_{k,i}$ entirely. Instead, the algorithm computes $g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m))$ in each round by using Equation (6), directly querying the oracles $\mathcal{A}_1,\ldots,\mathcal{A}_\ell$ at points $u \in \{0,\ldots,2^n-1\}$.

At the start of each round in Algorithm 1, an array *accumulator* is initialized, where each entry *accumulator*$[s]$ iteratively computes the value of $f_i(\alpha_s)$ for $s \in \{0,1,\ldots,\ell\}$ through Steps 4-21. At Step 5 *witness_eval* is a two-dimensional array; *witness_eval*$[k][s]$ at Step 13 eventually stores the values of $g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m))$ for $k \in \{1,\ldots,\ell\}$, and $s \in \{0,\ldots,\ell\}$. Steps 7-16 iteratively compute $g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m))$ using Equation (6) for a fixed $m \in \{0,\ldots,2^{n-i}-1\}$. Using Equation (6), it becomes clear that the value $g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m))$ is computed correctly at the end of Step 16. This allows *accumulator*$[s]$ to accumulate the terms in the definition of f_i accurately at Step 19. Thus, at the end of Step 21, *accumulator*$[s]$ holds the correct value for $f_i(\alpha_s)$, ensuring the correctness of the algorithm in each round. Since Algorithm 1 does not store the arrays $A_1,\ldots,A_\ell$, it operates in logarithmic space (i.e., space $O(n)$).[14] However, this comes at the cost of making $O(n\cdot 2^n)$ oracle queries. In practice, this means that the evaluations of $g_1,\ldots,g_\ell$ over $\{0,1\}^n$ must be recomputed iteratively across n rounds. To optimize the prover's space usage in the sum-check protocol, a hybrid approach could be employed. Algorithm 1 would be used in the initial rounds to leverage logarithmic space, and then switch to a linear-time honest prover algorithm once the arrays $A_{k,i}$, for $k \in \{1,\ldots,\ell\}$, are reduced to a manageable size. Theorem 3.3 formally notes that Algorithm 1 is a quasi-linear time, logarithmic space, honest prover algorithm for the sum-check protocol. The proof of Theorem 3.3 follows from the above discussion.

Theorem 3.3. *Algorithm 1 implements the honest prover for sum-check protocol and has time complexity $O(\ell^2 n\cdot 2^n)$ and space complexity $O(n+\ell^2)$.*

Algorithm 1 Small Space Sum-Check Prover Algorithm

$\mathcal{P}$'s input: Oracles $\mathcal{A}_1,\ldots,\mathcal{A}_\ell$ that take as input $m \in \{0,\ldots,2^n-1\}$ and for $k \in \{1,\ldots,\ell\}$, $\mathcal{A}_k$ returns $g_k(\mathrm{tobits}(m))$.

1. /* We use A_k to denote the array that contains the evaluations of g_k over the boolean hypercube. Specifically, the j-th entry of A_k is equal to $g_k(\mathrm{tobits}(j))$.*/
2. **for** $i \in \{1,\ldots,n\}$ **do**
3. Initialize an array *accumulator* of size $\ell+1$ and set all the entries to zero.
4. **for** $m \in \{0,\ldots,2^{n-i}-1\}$ **do**
5. Initialize a two dimensional array *witness_eval* of size $\ell\times(\ell+1)$ and set all the entries to zero. /* *witness_eval* is used below to compute $g_k(r_1,\ldots,r_{i-1},\alpha_s,\mathrm{tobits}(m))$ */

[14] $O(n)$ space is required simply to store the n verifier challenges.

for $j \in \{0, \ldots, 2^{i-1} - 1\}$ **do**
Set $u_{\text{even}} = 2^i \cdot 2m + j$.
/* u_{even} has binary representation $(j, 0, \text{tobits}(m))$.*/
Query $\mathcal{A}_1, \ldots, \mathcal{A}_\ell$ at point u_{even} and determine $A_1[u_{\text{even}}], \ldots, A_\ell[u_{\text{even}}]$ respectively.
Set $u_{\text{odd}} = 2^i \cdot (2m + 1) + j$.
/* u_{odd} has binary representation $(j, 1, \text{tobits}(m))$.*/
Query $\mathcal{A}_1, \ldots, \mathcal{A}_\ell$ at point u_{odd} and determine $A_1[u_{\text{odd}}], \ldots, A_\ell[u_{\text{odd}}]$ respectively.
for all $k \in \{1, \ldots, \ell\}$ and $s \in \{0, \ldots, \ell\}$ **do**
$\mathit{witness\_eval}[k][s]$ += $\widetilde{\text{eq}}((r_1, \ldots, r_{i-1}), \text{tobits}(j)) \cdot ((1 - \alpha_s) \cdot A_k[u_{\text{even}}] + \alpha_s \cdot A_k[u_{\text{odd}}])$.
end for
end for
/* At this point, $\mathit{witness\_eval}[k][s]$ equals $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$.*/
for $k \in \{1, \ldots, \ell\}$ and $s \in \{0, \ldots, \ell\}$ **do**
$\mathit{accumulator}[s]$ += $\prod_{k=1}^{\ell} \mathit{witness\_eval}[k][s]$ for all $s \in \{0, \ldots, \ell\}$.
end for
end for
/* At this point, $\mathit{accumulator}[s]$ equals $\sum_{m=0}^{2^{n-i}-1} \prod_{k=1}^{\ell} g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$.*/
Set $f_i(\alpha_s) = \mathit{accumulator}[s]$ for $s \in \{0, 1, \ldots, \ell\}$ and send this quantity to $\mathcal{V}$.
end for

3.1.3 Sum-Check for Small Values

Recent work [BDT24] proposes enhancements to the linear-time honest prover algorithm for the sum-check protocol in cases where the summed values are small. By small here, we mean one of two things:

1. The protocol is applied over a large prime-order field (say, a 128-bit or 256-bit field $\mathbb{F}$) but for all $x \in \{0,1\}^n$, $g_1(x), \ldots, g_\ell(x)$ all reside in a small subset B of the field like $B = \{0, 1, \ldots, 2^{32} - 1\}$. In particular, it must be possible to multiply any two values in B using a single machine multiplication.

2. The protocol is applied over a field of small characteristic such as $\mathbb{F} = \text{GF}(2^{128})$, and all values in B reside in a small subfield (which we call the *base field*) such as $\text{GF}(2)$ or $\text{GF}(2^{32})$.[15]

In these settings, the standard linear-time proving algorithm (Section 3.1.1) performs $O(2^n)$ field operations, but about half of them are over the entire field $\mathbb{F}$. [BDT24] improve on this by reducing the number of necessary multiplications over the large field, concretely lowering the prover's runtime. We provide an overview of Algorithm 3 from [BDT24] and then show that the logarithmic-space algorithm (Algorithm 1) can be adjusted to incorporate this method.

Algorithm 3 of [BDT24] is also applicable in the setting where a single factor $g_1(x)$ does *not* output small values, e.g., $g_1(x) = \widetilde{\text{eq}}(r, x)$ for a random $r \in \mathbb{F}^n$. The speedups it offers in this context compared to the standard linear-time algorithm are smaller but still significant. In our context, what is most important is that Algorithm 3 from [BDT24] is no slower than the standard linear-time proving algorithm when the values being summed are small, while also being *directly streaming* (i.e., the prover can be implemented in small space with no slowdown).

To be more precise, Algorithm 3 from [BDT24] ensures that essentially all multiplications performed by the prover in the first several rounds involve only small values, and hence they can be performed with just a single machine multiplication (in Case (1) above), or with a single base-field multiplication. Such operations can be at least an order of magnitude faster than a general 128-bit or 256-bit field multiplication.

[15] Gruen [Gru24] provides an alternative proving algorithm offering speedups in Case (2) above, but not Case (1). Case (1) is essential for applicability to Jolt when using an elliptic curve commitment scheme.

The number of machine- or base-field- multiplications that must be done by the prover in each round of Algorithm 3 of [BDT24] increases by a constant factor each round (in contrast, for the standard linear-time algorithm of Section 3.1.1, the number of field multiplications *falls* by a constant factor round-over-round). Hence, after several rounds have passed, it is faster for the Algorithm 3 prover to "switch over" to an alternative proving algorithm.

Remark 3.4. Bagad et al. [BDT24] also provide a more efficient algorithm they call Algorithm 4. Algorithm 4 is also directly compatible with a streaming prover but it is substantially more complicated than [BDT24, Algorithm 3]. We focus on Algorithm 3 in this manuscript only for simplicity of exposition.

Overview of Algorithm 3 from [BDT24]. We recall that in each round, the linear-time honest prover algorithm for the sum-check protocol updates arrays $A_1, \ldots, A_\ell$ such that at the beginning of round i, A_k contains the evaluations of $g_k(r_1, \ldots, r_{i-1}, x)$ as x ranges over $\{0,1\}^{n-i+1}$. In the small-field setting, however, updating these arrays requires most operations to be performed over the large extension field, which is computationally intensive. The main insight in [BDT24] is that, instead of updating the arrays during the initial rounds, the prover can calculate a few additional terms at each round. Most of these calculations can be performed over the small field, enabling the prover to utilize the arrays $A_1, \ldots, A_\ell$, along with these additional terms, to compute the coefficients of f_i more efficiently. While this approach increases the number of required multiplications overall, most are over the small field, significantly reducing the prover's runtime. We illustrate this algorithm for $\ell = 2$, setting $\alpha_0 = 0$, $\alpha_1 = 1$, and $\alpha_2 = 2$. In this configuration, at round i of the sum-check protocol the prover needs to compute $f_i(0), f_i(1), f_i(2)$ for $i \in \{1, \ldots, n\}$.

Round 1: At the start the prover maintains an array C of size 2^n. The j-th entry of C is $A_1[j] \cdot A_2[j] = g_1(\text{tobits}(j)) \cdot g_2(\text{tobits}(j))$. In round 1, computing $f_1(0)$, and $f_1(1)$ is done as follows:

$$f_1(0) = \sum_{i \in \{0,\ldots,2^{n-1}-1\}} C[2 \cdot i], \text{ and } f_1(1) = \sum_{i \in \{0,\ldots,2^{n-1}-1\}} C[2 \cdot i + 1]$$

To compute $f_1(2)$, observe that

$$\begin{aligned}
f_1(2) &= \sum_{x \in \{0,1\}^{n-1}} g_1(2, x) \cdot g_2(2, x) \\
&= \sum_{x \in \{0,1\}^{n-1}} (2 \cdot g_1(1, x) - g_1(0, x)) \cdot (2 \cdot g_2(1, x) - g_2(0, x)) \\
&= \sum_{x \in \{0,1\}^{n-1}} 4 \cdot g_2(1, x) \cdot g_2(1, x) - 2(g_1(1, x) \cdot g_2(0, x) + g_1(0, x) \cdot g_2(1, x)) + g_1(0, x) \cdot g_2(0, x) \\
&= \sum_{i \in \{0,\ldots,2^{n-1}-1\}} 4 \cdot C[2 \cdot i + 1] \; - \sum_{i \in \{0,\ldots,2^{n-1}-1\}} 2 \left(A_1(2 \cdot i) \cdot A_2(2 \cdot i + 1) + A_1(2 \cdot i + 1) \cdot A_2(2 \cdot i)\right) \\
&\quad + \sum_{i \in \{0,\ldots,2^{n-1}-1\}} C[2 \cdot i]
\end{aligned}$$

Hence, to compute $f_1(2)$, the prover must compute two additional terms for all $i \in \{0, \ldots, 2^{n-1} - 1\}$:

- $A_1(2 \cdot i) \cdot A_2(2 \cdot i + 1)$, and
- $A_1(2 \cdot i + 1) \cdot A_2(2 \cdot i)$.

For any $i \in \{0, \ldots, 2^{n-1} - 1\}$, observe that these products are equivalent to $g_1(x) \cdot g_2(x')$, where x, x' satisfy:

- the most significant $n - 1$ bits of x and x' are equal to $\text{tobits}(i)$,
- x and x' differ in their least significant bit.

Thus, the prover computes $g_1(x) \cdot g_2(x')$, for all pairs $x, x' \in \{0,1\}^n$, where x and x' only differ in their least significant bit. These terms are added to the set C, as they will be required in subsequent rounds of the protocol.

Round 2: In round 2, the prover has to compute $f_2(s)$, for $s \in \{0,1,2\}$. $f_2(s)$ can be expressed as follows:

$$\begin{aligned} f_2(s) &= \sum_{x \in \{0,1\}^{n-2}} g_1(r_1, s, x) \cdot g_2(r_1, s, x) \\ &= \sum_{x \in \{0,1\}^{n-2}} ((1-r_1) \cdot g_1(0, s, x) + r_1 \cdot g_1(1, s, x)) \cdot ((1-r_1) \cdot g_2(0, s, x) + r_1 \cdot g_2(1, s, x)) \\ &= \sum_{x \in \{0,1\}^{n-2}} \sum_{y_1, y_2 \in \{0,1\}} \widetilde{\mathrm{eq}}(r_1, y_1) \cdot \widetilde{\mathrm{eq}}(r_1, y_2) \cdot g_1(y_1, s, x) \cdot g_2(y_2, s, x) \end{aligned}$$

Analysing the above expression, it is clear that to compute $f_2(s)$, for $s \in \{0,1,2\}$ the prover must compute

- $\widetilde{\mathrm{eq}}(r_1, 0) \cdot \widetilde{\mathrm{eq}}(r_1, 0) = (1-r_1)^2$,
- $\widetilde{\mathrm{eq}}(r_1, 0) \cdot \widetilde{\mathrm{eq}}(r_1, 1) = (1-r_1) \cdot r_1$,
- $\widetilde{\mathrm{eq}}(r_1, 1) \cdot \widetilde{\mathrm{eq}}(r_1, 1) = r_1^2$

The prover computes them and stores it in the array E. Now $f_2(0)$, and $f_2(1)$ are computed using the appropriate entries of C, and E. Recall, that at the end of round 1, the prover had to compute $g_1(y_1, s, x) \cdot g_2(y_2, s, x)$ for $y_1 \neq y_2$ and these products were stored in C. Computation of $f_2(2)$ is done using the following expression

$$\begin{aligned} f_2(2) &= \sum_{x \in \{0,1\}^{n-2}} \sum_{y_1, y_2 \in \{0,1\}} \widetilde{\mathrm{eq}}(r_1, y_1) \cdot \widetilde{\mathrm{eq}}(r_1, y_2) \cdot (2 \cdot g_1(y_1, 1, x) - g_1(y_1, 0, x)) \cdot (2 \cdot g_1(y_2, 1, x) - g_2(y_2, 0, x)) \\ &= \sum_{x \in \{0,1\}^{n-2}} \sum_{y_1, y_2 \in \{0,1\}^{n-2}} \widetilde{\mathrm{eq}}(r_1, y_1) \cdot \widetilde{\mathrm{eq}}(r_1, y_2) \cdot (4 \cdot g_1(y_1, 1, x) \cdot g_2(y_2, 1, x) \; - \\ &\qquad 2 \cdot g_1(y_1, 1, x) \cdot g_2(y_2, 0, x) - 2 \cdot g_1(y_1, 0, x) \cdot g_2(y_2, 1, x) + g_1(y_1, 0, x) \cdot g_2(y_2, 0, x)) \end{aligned}$$

Similar to round 1, computing $f_2(2)$ requires the prover to additionally compute $g_1(x) \cdot g_2(x')$, for all pairs $x, x' \in \{0,1\}^n$, where the least two significant bits of x and x' are not equal. These terms are computed by the prover and added to C, as they are required in the later rounds.

Round i, $i > 2$: Before the start of this round, C stores $g_1(x) \cdot g_2(x')$, for all pairs $x, x' \in \{0,1\}^n$, such that the last i bits of x and x' are not equal. Additionally, E stores the sequence $\{\widetilde{\mathrm{eq}}(\mathbf{r}_{i-1}, \mathbf{y}_1) \cdot \widetilde{\mathrm{eq}}(\mathbf{r}_{i-1}, \mathbf{y}_2)\}_{\mathbf{y}_1, \mathbf{y}_2 \in \{0,1\}^i}$ where $\mathbf{r}_{i-1}$ denotes $(r_1, \ldots, r_{i-1})$ In round i, for $s \in \{0,1,2\}$, the prover has to compute $f_i(s)$. We work out the expression for $f_i(s)$ next.

$$\begin{aligned} f_i(s) &= \sum_{x \in \{0,1\}^{n-i}} g_1(\mathbf{r}_{i-1}, s, x) \cdot g_2(\mathbf{r}_{i-1}, s, x) \\ &= \sum_{x \in \{0,1\}^{n-i}} \sum_{\mathbf{y}_1, \mathbf{y}_2 \in \{0,1\}^i} \widetilde{\mathrm{eq}}(\mathbf{r}_{i-1}, \mathbf{y}_1) \cdot \widetilde{\mathrm{eq}}(\mathbf{r}_{i-1}, \mathbf{y}_2) \cdot g_1(\mathbf{y}_1, s, x) \cdot g_2(\mathbf{y}_2, s, x) \end{aligned}$$

Similar to round 2, the values $f_i(0)$ and $f_i(1)$ can be computed directly using the values stored in C and E. Computation of $f_i(2)$ is done using the following expression

$$\begin{aligned} f_i(2) &= \sum_{x \in \{0,1\}^{n-i}} \sum_{\mathbf{y}_1, \mathbf{y}_2 \in \{0,1\}^{n-i+1}} \widetilde{\mathrm{eq}}(\mathbf{r}_1, \mathbf{y}_1) \cdot \widetilde{\mathrm{eq}}(\mathbf{r}_1, \mathbf{y}_2) \cdot (2 \cdot g_1(\mathbf{y}_1, 1, x) - g_1(\mathbf{y}_1, 0, x)) \cdot (2 \cdot g_1(\mathbf{y}_2, 1, x) - g_2(\mathbf{y}_2, 0, x)) \\ &= \sum_{x \in \{0,1\}^{n-i}} \sum_{\mathbf{y}_1, \mathbf{y}_2 \in \{0,1\}^{n-i+1}} \widetilde{\mathrm{eq}}(\mathbf{r}_1, \mathbf{y}_1) \cdot \widetilde{\mathrm{eq}}(\mathbf{r}_1, \mathbf{y}_2) \cdot (4 \cdot g_1(\mathbf{y}_1, 1, x) \cdot g_2(\mathbf{y}_2, 1, x) - \end{aligned}$$

$$2 \cdot g_1(\mathbf{y}_1, 1, x) \cdot g_2(\mathbf{y}_2, 0, x) - 2 \cdot g_1(\mathbf{y}_1, 0, x) \cdot g_2(\mathbf{y}_2, 1, x) + g_1(\mathbf{y}_1, 0, x) \cdot g_2(\mathbf{y}_2, 0, x))$$

The computation of the above expression requires the prover to additionally compute $g_1(x) \cdot g_2(x')$, for pairs $x, x' \in \{0,1\}^n$, where the least i significant bits of x and x' are not equal. Similar to the previous rounds, these terms are computed by the prover and added to C.

Switching to the linear-time honest prover algorithm. Using this approach for the first few rounds minimizes the number of large field operations, making it computationally efficient. However, after the initial rounds (approximately around $\frac{n}{2}$rounds) the overhead of maintaining the array E becomes higher than if the linear-time honest prover algorithm were used directly. Therefore, to optimize performance, it is preferable to switch to the linear-time algorithm after roughly half the rounds. This transition leverages the benefits of reduced large field operations initially, followed by the faster overall update mechanism of the linear-time approach in later rounds.

3.1.4 Small-space Sum-check Proving for Small Values

In the small-space adaptation of Algorithm 3 from [BDT24], the prover computes the array C, on-the-fly rather than pre-computing it, by using oracles to g_1 and g_2 to compute C at run-time. This change significantly reduces the space requirements, since the array C now only requires $O(2^i)$ space at round i, rather than the full $O(2^n)$ space required for precomputed storage. Further, the prover maintains the array E as is in the original algorithm, and this requires $O(2^i)$ space. Although this runtime computation increases the number of small-value or base-field operations, it is computationally inexpensive and only adds overhead during the initial rounds. Thus, this small-space modification preserves efficiency while reducing storage needs. As the rounds progress, it becomes advantageous for the prover's runtime to switch to the linear-time prover algorithm. This will typically happen after roughly 4 rounds have passed [BDT24]. A more complicated variation of [BDT24, Algorithm 3], called Algorithm 4 in [BDT24], has better costs and can extend the optimal "crossover round" to round 8 or later. This would lead to at least a $2^8 = 256$-fold reduction in prover space compared to running the standard linear-time proving algorithm beginning in Round 1. If further reduction of prover space is desired, rather than switching directly to the linear-space algorithm, the prover can instead switch to the standard small-space proving algorithm of Section 3.1.2, until enough rounds have passed that the prover can finally switch over to the linear-time prover algorithm without increasing its overall space usage.

3.2 Jolt Overview

Jolt is a state-of-the-art zkVM designed to prove the correct execution of programs that compile to the RISC-V instruction set architecture. At its core, Jolt leverages sum-check-based memory checking arguments to streamline proof generation, minimizing both the number of committed values and the size of each committed value, as well as the number of field operations needed to prove that the committed values are well-formed.

The proof system is built on three tightly integrated components: a memory-checking argument for read-only memory (Lasso [STW24] today, and Shout [ST25] in the near future), a memory-checking argument for read/write memory (Spice [SAGL18] today and Twist [ST25] in the near future), and a SNARK for a simple constraint system such as R1CS (namely Spartan [Set20]). Each component, as well as the way they are applied and combined, contributes significantly to Jolt's efficiency and simplicity compared to prior zkVMs.

Lasso and Shout are indexed lookup arguments (also known as SNARKs for reads into read-only memory), and Jolt uses them to handle proving correct execution of primitive RISC-V instructions, avoiding the need for complex, instruction-specific circuits. This design significantly reduces the overhead (in both prover time and implementation complexity) associated with proving correct instruction execution. Jolt

also uses lookup arguments to ensure that reads into a computer program's bytecode are processed correctly.

Spice and Twist are arguments for read/write memory, which Jolt uses to ensure that the prover correctly processes all reads and writes to registers and RAM.

Finally, Jolt employs Spartan to prove satisfaction of a rank-one constraint system (R1CS). The constraint system essentially ensures that all primitive instructions are executed with the correct source and destination registers (and that the correct primitive instruction is executd each cycle), and that all reads and writes to memory access the correct memory cells.

This constraint system arising in Jolt is uniform across all RISC-V cycles, meaning it is defined for a single cycle and then replicated for every executed cycle. Jolt's implementation adapts a variant of Spartan to prove such a uniform R1CS especially quickly.

The Jolt prover executes the RISC-V program to generate and commit the necessary witnesses for its three constituent proof systems: the read-only memory-checker, read/write memory checker, and R1CS satisfaction. The witness generation algorithm operates efficiently by computing witnesses as a set of vectors, with each vector of length equal to the number T of executed CPU cycles. The witness generation algorithm computes the vector entries for each cycle sequentially, requiring only $O(1)$ time per cycle (and $O(1)$ words of space, on top of the K words of space needed to run the RISC-V program).

Observation 3.5. The witness generation procedure in Jolt computes the witnesses for the three constituent proof systems as a set of vectors, with each vector's size bounded by $O(T)$, where T is the number of CPU cycles executed. The procedure iteratively generates each vector's entries while executing the program cycle-by-cycle, with additive constant overhead in time and words of space required.

3.3 Twist and Shout Overview

Shout is a memory-checking argument for read-only memory, while Twist is for read-write memory.

Overview of Shout. In Shout, there is a predetermined read-only memory M of size K. The prover commits to (the multilinear extension $\widetilde{\mathsf{ra}}$ of) a vector ra with length $T \cdot K$, where T is the number of reads into memory. The vector ra is interpreted as specifying T *addresses*, one per read operation. Each address is specified via *one-hot encoding*. This mans that if the k'th memory cell (for $k \in \{0, \ldots, K-1\}$), is being read, then this is specified by the unit vector $e_k \in \{0,1\}^K$ whose k'th entry equals 1. Since the prover is not trusted, the verifier must *check* that each address is indeed a valid one-hot encoding (i.e., a unit vector in $\{0,1\}^K$).

Let $\widetilde{\mathsf{rv}}$ denote the multilinear extension of the vector $\mathsf{rv} \in \mathbb{F}^T$ whose j'th entry is *the value returned by the j'th read operation*. That is, if the j'th address in $\widetilde{\mathsf{ra}}$ is e_k (i.e., the j'th read is to memory cell k, then $\widetilde{\mathsf{rv}}(j) = M(k)$ (i.e., the value stored in memory cell k).

The goal of the Shout protocol is to not only confirm that all committed addresses are indeed unit vectors, but also to allow the verifier to evaluate $\widetilde{\mathsf{rv}}(r)$ for a point $r \in \mathbb{F}^{\log T}$ of the verifier's choosing. Note that in Shout, $\widetilde{\mathsf{rv}}$ itself need not be committed by the prover.

The Shout protocol works by observing that

$$\widetilde{\mathsf{rv}}(r) = \sum_{(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}} \widetilde{\mathrm{eq}}(r,j) \cdot \widetilde{\mathsf{ra}}(k,j) \cdot \widetilde{M}(k).$$

Hence, the verifier forces the prover to tell it $\widetilde{\mathsf{rv}}(r)$ by applying the sum-check protocol to the $(\log(K) + \log(T))$-variate polynomial

$$g(k,j) = \widetilde{\mathrm{eq}}(r,j) \cdot \widetilde{\mathsf{ra}}(k,j) \cdot \widetilde{M}(k).$$

This is referred to as the *read-checking sum-check* in Shout.

At the end of this invocation of the sum-check protocol, the verifier has to evaluate $\widetilde{\mathrm{eq}}(r,r'')$, $\widetilde{\mathsf{ra}}(r',r'')$, and $\widetilde{M}(r')$ for a random point $(r',r'') \in \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$. The verifier can directly evaluate $\widetilde{\mathrm{eq}}(r,r'')$ in $O(\log T)$ time, and can obtain the evaluation of $\widetilde{\mathsf{ra}}$ from the commitment to $\widetilde{\mathsf{ra}}$.

In addition, Shout invokes the sum-check protocol twice to confirm that all committed addresses are indeed unit vectors. The first invocation of the sum-check protocol confirms that all entries of all committed addresses are in $\{0,1\}$—this is called the *Booleanity-checking sum-check*. The second sum-check invocation confirms that the Hamming weight of each address is exactly 1—this is called the *Hamming-weight-one-checking sum-check*.

In Jolt, Shout is applied primarily to lookup tables M for which $\widetilde{M}(r')$ can be evaluated by the verifier for any point r' with only logarithmically many field operations. The one exception is for lookups into program bytecode. In this case, $\widetilde{M}$ is committed in pre-processing by an honest party, and using this commitment scheme, the prover provides the necessary evaluation of $\widetilde{M}$.

Overview of Twist. Twist is most cleanly formulated under the assumption that T read operations and T write operations are performed in an interleaved manner, with the first read followed by the first write, which is followed by the second read and then the second write and so forth. Let us further assume in this formulation that all memory cells are intialized to store value 0.

Before the Twist protocol begins, the prover has already committed to $\widetilde{\mathsf{ra}}$ (specifying all the read addresses), as well as $\widetilde{\mathsf{wa}}$ (specifying the write addresses) and $\widetilde{\mathsf{wv}}$, where for $j \in \{0,1\}^{\log T}$, $\widetilde{\mathsf{wv}}(j)$ specifies the value written by the j'th write operation. Let $\widetilde{\mathsf{rv}}$ denote the multilinear extension of the vector whose j'th entry equals the value returned by the j'th read operation.

The goal of Twist is to confirm that all read-address and write-address are indeed unit vectors, and to allow the verifier to compute $\widetilde{\mathsf{rv}}(r)$ for a point $r \in \mathbb{F}^{\log T}$ of the verifier's choosing.

In Twist, the prover begins by committing to (the multilinear extension $\widetilde{\mathsf{Inc}}$) of an extra vector $\mathsf{Inc} \in \mathbb{F}^{\log T}$. If the prover is honest, then $\mathsf{Inc}(j)$ is the *difference* between the value written by the j'th write operation, namely $\widetilde{\mathsf{wv}}(j)$, and the value stored at the relevant cell at that time. By "the value stored at the relevant cell at that time", we mean the following: let j' be the largest time step strictly less than j such that $\widetilde{\mathsf{wa}}(j') = \widetilde{\mathsf{wa}}(j)$. Then $\widetilde{\mathsf{wv}}(j')$ is what we mean by the value stored at the relevant cell at time j. If no such j' exists, then this relevant value is 0.

Twist consists of three invocations of the sum-check protocol. The first is called the *read-checking sum-check*, which intuitively allows the verifier to compute $\widetilde{\mathsf{rv}}(r)$ *assuming* $\widetilde{\mathsf{Inc}}$ *is (the multilinaer extension of) the correct vector of increments.* The second sum-check is called the *write-checking sum-check*, which intuitively checks that $\widetilde{\mathsf{Inc}}$ is indeed (the multilinear extension of) the correct vector of increments. The third sum-check is called the $\widetilde{M}$*-evaluation sum-check*. This sum-check intuitively arises because the read-checking sum-check actually requires the verifier to be able to evaluate a certain polynomial $\widetilde{M}$ at a point $(r', r'') \in \mathbb{F}^{\log K} \times \mathbb{F}^{\log T}$, where $\widetilde{M}$ is *not* $\widetilde{\mathsf{Inc}}$ but is *derived from* $\widetilde{\mathsf{Inc}}$. Specifically, $\widetilde{M}$ is the unique multilinear polynomial such that, for any memory address $k \in \{0,1\}^{\log K}$ and timestep $j \in \{0,1\}^{\log T}$, $\widetilde{M}(k,j)$ equals the value stored at address k at time j according to the write operations specified by $\widetilde{\mathsf{wa}}$ and $\widetilde{\mathsf{wv}}$.

From the verifier's perspective, the $\widetilde{M}$-evaluation sum-check reduces the task of evaluating $\widetilde{M}(r', r'')$ to the task of evaluating $\widetilde{\mathsf{Inc}}$ at a different point. Since $\widetilde{\mathsf{Inc}}$ was explicitly committed by the prover, the prover itself can provide this evaluation of $\widetilde{\mathsf{Inc}}$ (along with a proof that the provided evaluation is indeed consistent with the commitment).

In more detail, the three invocations of the sum-check protocol in Twist are applied to the following polynomials:

- The read-checking sum-check computes the sum

$$\sum_{(k,j) \in \{0,1\}^{\log K} \times \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r,j) \cdot \widetilde{\mathsf{ra}}(k,j) \cdot \widetilde{M}(k,j),$$

  where $r \in \mathbb{F}^{\log T}$ is the point at which the verifier wishes to evaluate $\widetilde{\mathsf{rv}}(r)$.

- The write-checking sum-check computes the sum

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

  Here, r and r' are chosen at random by the verifier before the sum-check protocol is invoked. If the prover is honest, this sum is 0 for any choice of r and r'.

- The $\widetilde{M}$-evaluation sum-check computes the sum

$$\widetilde{M}(r,r') = \sum_{j\in\{0,1\}^{\log T}} \widetilde{\mathsf{Inc}}(r,j)\cdot\widetilde{\mathrm{LT}}(r',j),$$

  where $\widetilde{\mathrm{LT}}$ is the multilinear extension of the so-called *less-than* function that on any input $(j,j')\in\{0,1\}^{\log T}\times\{0,1\}^{\log T}$, outputs 1 if $\mathrm{val}(j)<\mathrm{val}(j')$ and outputs 0 otherwise. Here, r and r' are points chosen by the verifier over the course of the read-checking sum-check.

The "dimension" parameter d. For both Twist and Shout, if the memory size K is too big, the prover is not able to commit to $\widetilde{\mathsf{ra}}$ (and $\widetilde{\mathsf{wa}}$ as well in the case of Twist) directly. If using an elliptic curve commitment scheme, the issue is that the commitment key will be too large to commit to (the multilinear extension of) a vector of size $K\cdot T$, no matter how sparse the vector is. If using a binary-field hashing-based commitment scheme such as Binius [DP23, DP24], the issue is not that the commitment key is too large (as the commitment key is just a specification of a hash function) but rather that committing to a vector of size $K\cdot T$ is simply too expensive, no matter how sparse the committed vector is. In these situations, $\widetilde{\mathsf{ra}}$ (and $\widetilde{\mathsf{wa}}$ in the case of Twist) can be replaced with a product of d smaller polynomials. That is, for $k=(k_1,\ldots,k_d)$ where each k_i consists of $log(K)/d$ variables, $\widetilde{\mathsf{ra}}(k,j)$ is replaced with

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

We call d the dimension parameter of Twist and Shout. The larger d is, the more field operations the prover must perform within the read-checking sum-check in Shout, and the read-checking and write-checking sum-checks in Twist. For elliptic curve commitment schemes, larger choices of d also lead to somewhat higher commitment costs (as the prover commits to d 1s per address, and each committed 1 costs the prover one group operation).

Hence, for curve-based commitments, one wants to to set d as small as possible subject to the constraint that the commitment key is not too large for the prover to store. For hashing-based commitments, one wants to set d as small as possible subject to the constraint that commitment time is not a bottleneck for the prover. In this work, we regard d as a constant. This ensures that the Twist prover runs in time $O(K+T\log K)$ and the Shout prover runs in time $O(T)$.

With curve-based commitment schemes it is typically optimal to set d between 1 and 4. With hashing-based commitment schemes over binary fields, optimal settings of d range from 1 (for very small memories such as the 32 registers in RISC-V) to 16 (for enormous, structured lookup tables such as those arising in Jolt's handling of primitive instruction execution).

4 Spartan-Proving in Small Space for Uniform Constraint Systems

Jolt employs Spartan to prove the satisfaction of its R1CS-based constraint system. In this section, we demonstrate how to implement Spartan's prover algorithm in small space by leveraging the uniform structure of Jolt's constraints and witness vectors.

4.1 R1CS Structure in Jolt

An R1CS constraint system comprises three matrices $A, B, C \in \mathbb{F}^{m \times n}$, where typically each row contains $O(1)$ non-zero entries. A witness $\mathbf{w} \in \mathbb{F}^{n-1}$ satisfies the system if $\mathbf{u} = (1, \mathbf{w})$ satisfies

$$(A \cdot \mathbf{u}) \circ (B \cdot \mathbf{u}) = C \cdot \mathbf{u},$$

where $\circ$ denotes component-wise product. Jolt's constraint system has a highly regular structure:

- Each CPU cycle's execution is captured by β constraints (assumed to be a power of two for simplicity).
- These constraints are repeated across all T execution cycles, giving $m = \beta \cdot T$ total constraints.
- The constraints verify index computation, lookup correctness, and program counter updates, among other things.

The witness vector $\mathbf{w}$ is constructed by interleaving k different vectors $\mathbf{w}_1, \ldots, \mathbf{w}_k \in \mathbb{F}^T$:

$$\left\{ \{w_{i,j}\}_{i \in \{0,\ldots,k-1\}} \right\}_{j \in \{0,\ldots,T-1\}}$$

We call the elements in positions $(j \cdot k, \ldots, (j+1) \cdot k - 1)$ of $\mathbf{w}$ the j-th slice of $\mathbf{w}$. From Observation 3.5, each slice is computable in $O(1)$ space and time after executing the corresponding CPU cycle.

In addition to enforcing constraints within individual cycles, the constraint system must also perform a single check across every pair of adjacent cycles, to ensure that the program counter (pc) used in the next cycle is correctly determined by the instruction being executed in the current cycle. This is simply an equality constraint: the value of the program counter at the end of cycle i should be equal to its value at the start of cycle $i+1$. However, for performance rasons, in Jolt these equality constraints are not enforced with an explicit constraint within the R1CS instance. Rather they are enforced as described next.

Jolt introduces two vectors: one for the program counter (pc) and another for the next program counter (pcnext), where pcnext is simply the pc values shifted by one position to the right. The R1CS itself can "pretend" that the prover has committed to both pc and pcnext, even though only pc is explicitly committed. In the terminology of Diamond and Posen [DP23], pcnext is a *virtual polynomial*: it's not explicitly committed, but the verifier is still able to obtain its evaluation at any point r, by invoking the sum-check protocol to reduce the task of evaluating pcnext at r to the task of evaluating next (which *is* committed by the prover) at a different point. This reduction is achieved by applying the sum-check protocol to the right hand side of the following equality:

$$\widetilde{\mathsf{pcnext}}(r) = \sum_{j \in \{0,1\}^{\log T}} \widetilde{\mathsf{shift}}(r, j) \cdot \widetilde{\mathsf{pc}}(j), \tag{7}$$

where the shift function is defined as:

$$\mathsf{shift}(i, j) = \begin{cases} 1, & \text{if } \mathrm{val}(i) = \mathrm{val}(j) + 1, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, when the verifier needs to compute $\widetilde{\mathsf{pcnext}}(r)$, it applies the sum-check protocol to evaluate the summation efficiently.[16] We call this the pcnext-*evaluation sum-check*. At the end of this sum-check protocol invocation, the verifier has to evaluate $\widetilde{\mathsf{shift}}$ at a randpom point, as well as pc. Evaluating $\widetilde{\mathsf{shift}}$ at a random

[16] There are alternative approaches that would also work but with higher costs and complications. One approach is to have the prover explicitly commit to pcnext and add equality constraints to the R1CS to force pcnext to indeed equal, entry-by-entry, pc shifted to the right by one entry. The other approach is to have a constraint for cycle $i+1$ directly read the program counter at the end of the previous cycle. Both approaches render the constraint system not quite block-diagonal, seemingly prevented the use of various concrete optimizations to the protocol [AT+24]. Hence, we prefer to keep the constraint system block-diagonal and incur an extra invocation of the sum-check protocol.

point can be done in logarithmic space and time (for details, see Theorem 2 in [STW23] or Equation (21) in Section 4.2 below).

In summary, the R1CS instance arising in Jolt can treat pcnext as a variable in the constraint system corresponding to the RISC-V cycle; this preserves the block-diagonal nature of the R1CS. However, at the end of Spartan applied to this R1CS, the verifier must evaluate not only $\widetilde{\mathsf{pc}}$ at a random point r, but also $\widetilde{\mathsf{pcnext}}(r)$. This latter evaluation is obtained via the sum-check invocation described above.

4.2 Small-Space Honest-Prover Algorithm for Spartan in Jolt

To prove witness satisfaction in Spartan, the prover must show that the following polynomial $q(\mathbf{S})$ is formally zero:

$$q(\mathbf{S}) = \sum_{y \in \{0,1\}^{\log m}} \widetilde{\mathrm{eq}}(\mathbf{S}, y) \cdot \Big(\Big(\sum_{x \in \{0,1\}^{\log n}} \widetilde{A}(y, x) \cdot \widetilde{\mathbf{u}}(x) \Big) \circ \Big(\sum_{x \in \{0,1\}^{\log n}} \widetilde{B}(y, x) \cdot \widetilde{\mathbf{u}}(x) \Big) - \sum_{x \in \{0,1\}^{\log n}} \widetilde{C}(y, x) \cdot \widetilde{\mathbf{u}}(x) \Big) \tag{8}$$

For clarity, we define:

$$\widetilde{h}_A(\mathbf{Y}) = \sum_{x \in \{0,1\}^{\log n}} \widetilde{A}(\mathbf{Y}, x) \cdot \widetilde{\mathbf{u}}(x)$$

$$\widetilde{h}_B(\mathbf{Y}) = \sum_{x \in \{0,1\}^{\log n}} \widetilde{B}(\mathbf{Y}, x) \cdot \widetilde{\mathbf{u}}(x)$$

$$\widetilde{h}_C(\mathbf{Y}) = \sum_{x \in \{0,1\}^{\log n}} \widetilde{C}(\mathbf{Y}, x) \cdot \widetilde{\mathbf{u}}(x)$$

To this end the prover and verifier engage in two sum-check protocols in sequence:

1. First Sum-Check Protocol:
   - Verifier samples random point $\mathbf{r}_s \in \mathbb{F}^{\log m}$.
   - The sum-check protocol is invoked to verify:

$$0 = \sum_{y \in \{0,1\}^{\log m}} \widetilde{\mathrm{eq}}(\mathbf{r}_s, y) \cdot \Big(\widetilde{h}_A(y) \cdot \widetilde{h}_B(y) - \widetilde{h}_C(y) \Big). \tag{9}$$

   - At the conclusion of the first sum-check protocol, the prover provides claimed evaluations of relevant multilinear polynomials at a point $\mathbf{r}_y$. Specifically, the prover needs to convince the verifier of the correctness of the following values: $\widetilde{\mathrm{eq}}(\mathbf{r}_s, \mathbf{r}_y)$, $\widetilde{h}_A(\mathbf{r}_y)$, $\widetilde{h}_B(\mathbf{r}_y)$, and $\widetilde{h}_C(\mathbf{r}_y)$, where $\mathbf{r}_y \in \mathbb{F}^{\log m}$ consists of the random field elements sampled by the verifier during the sum-check protocol. The verifier can compute $\widetilde{\mathrm{eq}}(\mathbf{r}_s, \mathbf{r}_y)$ directly, while establishing the correctness of the remaining evaluations requires an additional sum-check protocol, which we describe next.
2. Second Sum-Check Protocol:
   - To validate the claimed evaluations, the prover must establish:

$$\widetilde{h}_A(\mathbf{r}_y) = \sum_{x \in \{0,1\}^{\log n}} \widetilde{A}(\mathbf{r}_y, x) \cdot \widetilde{\mathbf{u}}(x)$$

$$\widetilde{h}_B(\mathbf{r}_y) = \sum_{x \in \{0,1\}^{\log n}} \widetilde{B}(\mathbf{r}_y, x) \cdot \widetilde{\mathbf{u}}(x)$$

$$\widetilde{h}_C(\mathbf{r}_y) = \sum_{x \in \{0,1\}^{\log n}} \widetilde{C}(\mathbf{r}_y, x) \cdot \widetilde{\mathbf{u}}(x).$$

- To minimize proof size and verifier time, these three equations are verified simultaneously using a single sum-check protocol over their random linear combination.
- At the end of this second sum-check instance, the verifier must evaluate $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$ at point $(\mathbf{r}_y, \mathbf{r}_x)$, along with $\widetilde{\mathbf{u}}$ at $\mathbf{r}_x$. Here, $\mathbf{r}_x \in \mathbb{F}^{\log n}$ comprises the random field elements chosen by the verifier during this protocol. Due to the blog-diagonal structure of A, B, and C, the evaluations of $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$ can be computed by the verifier in $O(\log T)$ time. The evaluation of $\widetilde{\mathbf{u}}$ is provided by the prover along with an evaluation proof from the PCS used to commit to $\widetilde{\mathbf{u}}$.

Remark 4.1. Due to the block-diagonal nature of the constraint system arising in Jolt, this second invocation of the sum-check protocol can in fact involve a sum over only $O(1)$ terms (namely, the number of committed variables per CPU cycle). For brevity, we omit a description of this optimization as it leads to only a constant-factor improvement in the costs of Spartan, albeit a substantial one (a factor of about 2 in the Spartan prover's runtime, as well as Spartan's round and communication complexity). Details can be found at [AT+24].

In Jolt, all values arising in the vectors h_A, h_B, h_C (whose multilinear extension polynomials are $\widetilde{h}_A$, $\widetilde{h}_B$, and $\widetilde{h}_C$) are in $\{0, 1, \ldots, 2^{64} - 1\}$ (in fact, all but one such value per cycle is in $\{0, 1, \ldots, 2^{32} - 1\}$). This means that the prover in the first sum-check instance within Spartan can use the "small-value sum-check" algorithm of Section 3.1.3 so long as the elements of the vectors $\{\mathrm{eq}(\mathbf{r}_s, \mathbf{y})\}_{\mathbf{y} \in \{0,1\}^{\log m}}$, h_A, h_B, and h_C are enumerable in lexicographic order in logarithmic space. The values in the set $\{\mathrm{eq}(\mathbf{r}_s, \mathbf{y})\}_{\mathbf{y} \in \{0,1\}^{\log m}}$ can indeed be enumerated in lexicographic order in logarithmic space and $O(m) = O(T)$ total time See [CFFZE24, Section 4.1] or [Rot24, Appendix A] for details).[17]

Since the matrices A, B, and C are block-diagonal and have only $O(1)$ non-zero entries per row, the computation of each element of h_A, h_B, and h_C for a specific $\mathbf{y}$ depends only on the variables materialized for the corresponding cycle and hence can be computed in $O(1)$ space and time. This completes the description of how to implement the prover in the first Spartan sum-check instance in small space.

Similarly, for the prover to execute the second sum-check instance in small space, each element of the witness vectors $\widetilde{A}(\mathbf{r}_y, x)$, $\widetilde{B}(\mathbf{r}_y, x)$, and $\widetilde{C}(\mathbf{r}_y, x)$ must also be computable in $O(1)$ space and time. This follows again from the block-diagonal structure of A, B, and C.

The protocol concludes by verifying evaluations at the point $\mathbf{r}_x$ using the PCS for $\widetilde{\mathbf{u}}$. For various polynomial commitment schemes, this procedure is discussed in Section 6.

The pcnext**-evaluation sum-check.** Recall from Equation (7) that the pcnext-evaluation sum-check applies the sum-check protocol to the polynomial

$$g(j) = \widetilde{\mathsf{shift}}(r, j) \cdot \widetilde{\mathsf{pc}}(j).$$

Tis invocation of the sum-check protocol is amenable to the new prefix-suffer inner product protocol we describe in Appendix A (the pcnext-evaluation sum-check does *not* appear to be directly amenable to the sparse-dense sum-check prove algorithm from [STW24, Appendix G]). For any integer $C \geq 1$, this allows the prover to execute the pcnext-evaluation sum-check with just C passes over the evaluations of $\widetilde{\mathsf{pc}}(j)$ as j ranges over $\{0,1\}^{\log T}$. The prover's runtime is $O(T)$ and the space bound is $O(C \cdot T^{1/C})$. See Appendix A for details.

5 Small-Space Proving for Twist and Shout

Shout proving. Setty and Thaler [ST25] give a variety of algorithms for implementing the read-checking sum-check prover within Shout. The right algorithm to use depends on whether $O(K + T)$ time is acceptable for the prover. If $K \ll T$ then this runtime is acceptable. If $K \gg T$ then it is not. In particular, Jolt

[17] If $O(\sqrt{m}) = O(T^{1/2})$ space is acceptable, then state-of-the-art techniques for sum-check proving actually allow the prover to avoid materializing all evaluations of $\widetilde{\mathrm{eq}}$ in this set [DT24]

applies Shout twice (see Section 3.2 for details): once to perform lookups into bytecode, and once to perform lookups into "the evaluation table of all primitive instructions". This table has size at least $K > 2^{64}$ (as primitive instructions in RISC-V operate on two 32-bit inputs), and it is unacceptable for the prover to run in time linear in 2^{64}.

When $O(K+T)$ runtime is acceptable, Setty and Thaler give a prover algorithm that works as follows:

- With a single pass over the read addresses and time $O(T)$ and space $O(K)$, the prover initializes a data structure of size $O(K)$, the contents of which are sufficient to get through the first $\log K$ rounds of the read-checking sum-check (out of $\log(K) + \log(T)$ total rounds) in time $O(K)$.
- The final $\log T$ rounds apply the sum-check protocol to compute

$$\sum_{j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, j) \cdot \widetilde{\mathsf{ra}}(r^*, j) \cdot \widetilde{M}(r^*),$$

  where r^* is the randomness chosen by the read-checking sum-check verifier over the first $\log K$ rounds. This invocation of the sum-check protocol is actually amenable to the sparse-dense sum-check protocol from [STW24, Appendix G] (this is more properly called the sparse-dense sum-check *algorithm*, as it does not alter the standard sum-check protocol, but rather offers a particularly efficient prover implementation when the sum-check protocol is applied to compute $\sum_{j \in \{0,1\}^{\log T}} p(j)q(j)$ if p satisfies certain structural properties). Here, $p(x) = \widetilde{\mathsf{eq}}(r, j)$ indeed satisfies the properties exploited by the sparse-dense sum-check prover algorithm).

  This protocol is capable of making, for any integer $C \geq 1$, only C passes over the input, using space $O(K + T^{1/C})$ and running in $O(CT)$ time.[18] If we are targeting $O(T^{1/2})$ space then we can set $C = 2$.

  Here, the $O(K)$ term in the prover's space bound ensures that the prover can compute a size-K table of all evaluations of the form $\widetilde{\mathsf{eq}}(r^*, k)$ as k ranges over $\{0,1\}^{\log K}$. This table can be computed in $O(K)$ time and space via standard techniques, and enables the prover to determine $\mathsf{ra}(r^*, j)$ for each $j \in \{0,1\}^{\log T}$ with a single lookup into the table for each j.

  An alternative (simpler but slower) way of implementing the final $\log T$ rounds of the sum-check protocol is to use the standard logarithmic-space sum-check proving algorithm (Theorem 3.3), switching over to the the standard linear-time proving algorithm (Section 3.1.1) once enough rounds have passed that the switch does not increase the prover's overall space usage.

When runtime $O(K+T)$ is unacceptable, Setty and Thaler observe that one can invoke the sparse-dense sum-check protocol from [STW24, Appendix G] to allow the prover to get through the first $\log K$ rounds of Shout's read-checking sum-check (so long as the look table M satisfies certain structural properties that are indeed satisfied by the primitive-instruction-evaluation-table used in Jolt). This protocol allows the prover to complete the first $\log K$ rounds of Shout's read-checking sum-check. For any $C > 1$, this algorithm runs in time $O(CK^{1/C} + CT)$, uses space $O(K^{1/C})$, and requires the prover to make C passes over the input. The final $\log T$ rounds proceed identically to the case where $O(K+T)$ runtime is acceptable.[19]

The situation for the Booleanity-checking and Hamming-weight-one-checking sum-checks is analogous. If $O(K^{1/d} + T)$ time is acceptable, the prover algorithms given in [ST25] require one pass over the read addresses to get through the first $\log(K)/d$ rounds of sum-check (out of $\log(K)/d + \log T$ in total) and the final $\log T$ rounds can be handled simiarly to the read-checking sum-check. If $O(K^{1/d} + T)$ time is not acceptable, [ST25] shows that a variation of the sparse-dense sum-check protocol applies, such that for any

[18] The algorithm called Blendy [CFFZE24] is a very special case of this sparse-dense sum-check algorithm. Blendy explicitly observes that in this special case the prover only requires space $K^{1/C}$ and C passes over the input. The same bounds apply in general to the sparse-dense sum-check algorithm. as well as to our new prefix-suffix inner product protocol in Appendix A.

[19] As our new prefix-suffix inner product protocol (Appendix A) is strictly simpler and concretely faster than the sparse-dense sum-check protocol, and so the Jolt implementation will use our new protocol rather than the sparse-dense sum-check protocol when Shout is fully integrated.

integer $C > 1$, the prover makes C passes over the input to get through the first $\log(K)/d$ rounds of these invocations of the sum-check protocol, runs in space $O(CK^{1/(Cd)}$ and time $O(CK^{1/(Cd)} + CT)$.

In summary, we conclude that Shout can be implemented, for any constant integer $C \geq 1$ by a prover running in space $O(K^{1/C} + T^{1/C})$, with at most a constant-factor increase in prover time.

Twist proving. Recall from Section 3.3 that Twist invokes the sum-check protocol three times, with the three invocations called the read-checking sum-check, the write-checking sum-check, and the $\widetilde{M}$-evaluation sum-check.

[ST25] gives two different algorithms for implement the read-checking and write-checking prover. Both have worst-case runtime $O(T \log K)$. One only applies when $d = 1$, but has the benefit that when the bulk of memory accesses are "2^i-local" (meaning, the read or write operation access a cell that was read or written at most 2^i steps prior), the prover's runtime is $O(T \cdot i)$ rather than $O(T \log K)$. Both algorithms have the following property: in each of the first $\log K$ rounds of the read-checking or write-checking sum-check, the prover makes a single pass over the read and write operations, runs in $O(K)$ space, and $O(T)$ total time.

The final $\log T$ rounds of these two instances of the sum-check protocol *cannot* be proven with the sparse-dense sum-check algorithm. Instead, we resort to the slower method of applying the standard logarithmic-space sum-check proving algorithm (Theorem 3.3), switching over to the the standard linear-time proving algorithm (Section 3.1.1) once enough rounds have passed that the switch does not increase the prover's overall space usage. This allows the prover to run in total time $O(T \log T)$ and space $O(K + \log T)$.

The $\widetilde{M}$-evaluation sum-check is amenable to the prefix-suffix inner-product prover algorithm that we describe in Appendix A. For any integer $C \geq 1$, this enables the prover to compute its messages in the $\widetilde{M}$-evaluation sum-check protocol with C passes over the $O(T)$ write operations to which Twist is applied, using $O(T)$ time and space $O(C \cdot T^{1/C})$. See Appendix A for details.

6 Polynomial Commitment Schemes in Small-Space

We categorize multilinear polynomial commitment schemes of interest into two groups: curve-based (HyperKZG, Hyrax, Dory, etc.) and hash-based (BaseFold, Ligero, Brakedown, Binius, FRI-Binius, Blaze, etc.).

6.1 Small-space implementation of hashing-based commitments

For some hash-based PCS implementations such as Ligero [AHIV17], Brakedown [GLS+23] and Binius [DP23, DP24], it is already known that the commitment and evaluation protocols require space complexity proportional to the square root of the size of the committed polynomial [BBHV22]. We provide details below for completeness.

When committing to a $(\log n)$-variate multilinear polynomial p, the Ligero, Brakedown and Binius polynomial commitment schemes arrange the n evaluations in the set $S = \{p(x) \colon x \in \{0,1\}^{\log n}\}$ into a $\sqrt{n} \times \sqrt{n}$ matrix M. They then apply the encoding function of an error-correcting code to each row of the matrix, and the committing is the Merkle hash of the encoded rows. To simplify our description, it is helpful to define the commitment to actually be the $\sqrt{n}$ hash values produced by Merkle-hashing each row independently. This increases the size of the commitment from a single hash value to $\sqrt{n}$ hash values, but since evaluation proofs are of size $O_\lambda(\sqrt{n})$ anyway, this leads to only a constant-factor increase in overall proof size (commitment size plus evaluation proof size).

When producing a proof for the claim that $p(r) = v$ for a point $r \in \mathbb{F}^{\log n}$ requested by the verifier, these schemes require the prover to do two things:

- Compute a linear combination of the rows of the (unencoded) matrix M, where the coefficients of the linear combination are determined by r.
- Open up $O(\lambda)$ randomly chosen columns of the committed (i.e., encoded) matrix.

If the prover is able to stream the evaluations of p (i.e., the entries of the unencoded matrix M) row-wise, then the commitment can be computed in $O(\sqrt{n})$ space with a single pass, as each row is encoded and Merkle-hashed independently of all the other rows.

For computing evaluation proofs, the necessary linear combination of the rows can be computed in $O(\sqrt{n})$ space with a single pass over the matrix. So can all $O(\lambda\sqrt{n})$ entries (and Merkle authentication paths) of the $O(\lambda)$ columns to be opened.

We mention that in typical implementations of these hashing-based commitment schemes, it is preferable to commit to the encoded matrix by hashing each column and then building a size-$\sqrt{n}$ Merkle tree over the resulting hash evaluations. This saves a logarithmic factor in proof size, as a separate Merkle authentication path does not need to be provided for each entry of an opened column. Unfortunately, this approach does not appear to be compatible with a small-space streaming prover: the values in the matrix have to be streamed in row-major order to allow for encoding in $O(\sqrt{n})$ space, and this precludes hashing entire columns of the encoded matrix in small space.

6.2 Small-space implementation of curve-based commitments

6.2.1 Hyrax

In this section we describe the Hyrax commitment scheme for multilinear polynomials [WTS+18], and how its prover can be implemented in a streaming manner with a constant number of passes over the committed polynomial. Throughout, G denotes an additive group with scalar field $\mathbb{F}$. For vectors $\mathbf{u} = (u_1, \ldots, u_n) \in \mathbb{F}^n$ and $\mathbf{g} = (g_1, \ldots, g_n) \in G^n$, the notation $\langle \mathbf{u}, \mathbf{g} \rangle$ denotes multiscalar multiplication, i.e., $\langle \mathbf{u}, \mathbf{g} \rangle = \sum_{i=1}^{n} u_i \cdot g_i$, where the sum denotes group addition and $u_i \cdot g_i$ denotes scalar multiplication. For two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$, we use the same notation $\langle u, v \rangle$ to denote the inner product $\sum_{i=1}^{n} u_i \cdot v_i$, where here the sum denotes field addition and the product denotes field multiplication.

The Commitment Protocol. As with the hashing-based schemes considered in Section 6.1 above, both Hyrax and Dory represent the multilinear polynomial p to be committed as a $\sqrt{n} \times \sqrt{n}$ matrix M, such that evaluating the polynomial at a point $\mathbf{r} = (r_1, \ldots, r_{\log n})$ is equivalent to computing

$$\mathbf{r}_1^T \cdot M \cdot \mathbf{r}_2, \tag{10}$$

where

$$\mathbf{r}_1 = \bigotimes_{i=1}^{\frac{\log n}{2}} (1 - r_i, r_i), \quad \mathbf{r}_2 = \bigotimes_{i=\frac{\log n}{2}+1}^{\log n} (1 - r_i, r_i). \tag{11}$$

Here, the entries of M contain all evaluations of p at inputs in $\{0,1\}^{\log n}$.

Hyrax's commitment key consists of $\sqrt{n}$ elements of an appropriate elliptic curve group, $\mathbf{g} = (g_1, \ldots, g_{\sqrt{n}})$. Each g_i is a uniform random group element.

Remark 6.1. In practice, each g_i can be produced by evaluating a PRG at (seed, i), where seed is some agreed-upon seed. This means that the elements of the commitment key can be generated one after the other in polylogarithmic space.

In Hyrax, the prover commits to each column of the matrix by performing an MSM, sending a $\sqrt{n}$-length commitment to the verifier. That is, the commitment to p is a vector of group elements $\mathbf{h} = (h_1, \ldots, h_{\sqrt{n}})$, where $h_i = \langle M_i, \mathbf{g} \rangle$, and M_i denotes the i'th column of M.

Streaming prover for committing. If the prover can stream over the entries of M in column-major order and spends $O(\sqrt{n})$ space to store the commitment key $\mathbf{g}$, then the Hyrax-commitment to p can clearly be computed in $O(\sqrt{n})$ space by applying Pippenger's algorithm independetly to each column.

By Remark 6.1, if one desires to achieve a space bound as low as polylogarithmic, one can still iteratively generate the group elements comprising the Hyrax commitment with a single pass over M in column-major order. This requires iteratively generating the commitment key per Remark 6.1 once per column of M.

Evaluation proofs: Simplest variation. In the simplest instantiation of Hyrax evaluation proofs, when the verifier requests $p(r)$, the prover simply sends a vector $\mathbf{k} \in \mathbb{F}^{\sqrt{n}}$ claimed to equal $M \cdot \mathbf{r}_2$, where $\mathbf{r}_2$ is defined in Equation (11).

The verifier computes $c^* = \langle \mathbf{r}_2, \mathbf{h} \rangle$ and confirms that $\langle \mathbf{k}, \mathbf{g} \rangle = c^*$ (in other words, the verifier checks that $\mathbf{k}$ opens c^*, which if the prover is honest is a Pedersen vector commitment to $M \cdot \mathbf{r}_2$. If so, the verifier is convinced that $p(\mathbf{r}) = \langle \mathbf{r}_1, \mathbf{k} \rangle$. In total, the evaluation proof consists of $\sqrt{n}$ field elements (the entries of $\mathbf{k}$) and the verifier performs two MSMs of size $\sqrt{n}$ and computes one inner product over $\mathbb{F}$ of size $\sqrt{n}$.

Streaming prover for simplest variation of evaluation proofs. To produce evaluation proofs as per the above protocol, the Hyrax prover merely needs to compute $M \cdot \mathbf{r}_2$. This is a linear combination of the columns of M, with coefficients given by the entries of $\mathbf{r}_2$. Via standard techniques [VSBW13], in $O(\sqrt{n})$ space and $O(\sqrt{n})$ time, the prover can explicitly compute and store the entries of $\mathbf{r}_2$, and hence the necessary linear combination of the columns of M can be computed in $O(n)$ time with a single streaming pass over M in column-major order.

By Remark 6.1, if one desires to achieve a space bound as low as polylogarithmic, one can still iteratively generate the field elements comprising the Hyrax evaluation proof, with a single pass over M in column-major order.

Evaluation proofs: Second variation. A more complicated variation of Hyrax evaluation proofs, described in [WTS+18], reduces the evaluation proof size to $O(\log n)$ group elements rather than $\sqrt{n}$ field elements (but this leads to only a constant-factor reduction in overall communication because the Hyrax commitment itself consists of $\sqrt{n}$ group elements).

In this variation, rather than explicitly sending a vector $k \in \mathbb{F}^{\sqrt{n}}$ claimed to equal $M \cdot \mathbf{r}_2$, the prover instead employs the Bulletproofs protocol [BCC+16, BBB+18] to demonstrate knowledge of such a vector. More precisely, the prover proves that it knows a vector $\mathbf{w}_1$ such that

- $\mathbf{w}_1 = M \cdot \mathbf{r}_2$, and
- $y = \langle \mathbf{r}_1, \mathbf{w}_1 \rangle$, where y is the claimed evaluation $p(\mathbf{r})$.

This application of Bulletproofs is an interactive protocol consisting of $\log(\sqrt{n})$ rounds (though it can be rendered non-interactive via the Fiat-Shamir transformation). At the beginning of round i, the prover possesses the witness vectors $\mathbf{w}_i$ and $\mathbf{u}_i$, as well as a vector G_i of group elements, of size $\frac{\sqrt{n}}{2^{i-1}}$. Additionally, the prover knows $y_i = \langle \mathbf{u}_i, \mathbf{w}_i \rangle$. In round 1, the prover sets $y_1 = y$, $\mathbf{w}_1 = M \cdot \mathbf{r}_2$, $\mathbf{u}_1 = \mathbf{r}_1$, and $G_1 = \mathbf{g}$. At the end of each round $i \in \{1, \ldots, \log \sqrt{n}\}$, the verifier samples a random scalar $\alpha_i \in \mathbb{F}$, which is used to compute the witness vectors and generators for the next round as follows:

$$\mathbf{w}_{i+1} = \alpha_i \cdot \mathbf{w}_{i,L} + \alpha_i^{-1} \cdot \mathbf{w}_{i,R}$$

$$\mathbf{u}_{i+1} = \alpha_i^{-1} \cdot \mathbf{u}_{i,L} + \alpha_i \cdot \mathbf{u}_{i,R}$$

$$G_{i+1} = \alpha_i^{-1} \cdot G_{i,L} + \alpha_i \cdot G_{i,R}$$

where $\mathbf{w}_{i,L}$ and $\mathbf{w}_{i,R}$, as well as $\mathbf{u}_{i,L}$ and $\mathbf{u}_{i,R}$, denote the left and right halves of $\mathbf{w}_i$ and $\mathbf{u}_i$, respectively. (To clarify, here the expression $\alpha_i \cdot G_i$ denotes the vector $(\alpha_i \cdot g_{i,1}, \ldots, \alpha_i g_{i,\frac{\sqrt{n}}{2^{i-1}}})$). Similarly, $G_{i,L}$ and $G_{i,R}$ are the left and right halves of G_i.

At the start of the protocol, as in the simpler variation of Hyrax evaluation proofs, the verifier on its own computes a commitment to $\mathbf{w}_1 = M \cdot \mathbf{r}_2$, as this commitment simply equals $\langle \mathbf{h}, \mathbf{r}_2 \rangle$.

In each round, the prover transmits the following values, which are referred to as *cross-terms*:

$$y_{i,L} = \langle \mathbf{u}_{i,L}^T, \mathbf{w}_{i,R} \rangle \quad y_{i,R} = \langle \mathbf{u}_{i,R}, \mathbf{w}_{i,L} \rangle, \quad \langle \mathbf{w}_{i,L}, G_{i,R} \rangle, \quad \langle \mathbf{w}_{i,R}, G_{i,L} \rangle. \tag{12}$$

Upon receiving these values, the verifier responds with α_i chosen at random from $\mathbb{F}$. Using these values, the verifier computes:

$$\mathbf{w}_{i+1} \cdot G_{i+1} = \mathbf{w}_{i,L} \cdot G_{i,R} + \alpha^2 \cdot (\mathbf{w}_{i,L} \cdot G_{i,R}) + \alpha^{-2} \cdot (\mathbf{w}_{i,R} \cdot G_{i,L})$$

$$y_{i+1} = y_i + \alpha^2 \cdot y_{i,L} + \alpha^{-2} \cdot y_{i,R}$$

After $\log \sqrt{n}$ rounds, the verifier obtains $\mathbf{w}_{\sqrt{n}+1} \cdot G_{\sqrt{n}+1}$ and $y_{\sqrt{n}+1} = \mathbf{u}_{\sqrt{n}+1} \cdot \mathbf{w}_{\sqrt{n}+1}$. At this stage, the prover sends $\mathbf{w}_{\sqrt{n}+1}$, while the verifier, having computed $\mathbf{u}_{\sqrt{n}+1}$ and $G_{\sqrt{n}+1}$ independently, verifies the correctness of the values $\mathbf{w}_{\sqrt{n}+1} \cdot G_{\sqrt{n}+1}$ and $\mathbf{u}_{\sqrt{n}+1} \cdot \mathbf{w}_{\sqrt{n}+1}$.

Streaming prover for second variation. Executing the evaluation protocol of this second variation of Hyrax in $O(\sqrt{n})$ space with a single pass over the committed matrix in column-major order is straightforward: the prover can compute and store the entries of $\mathbf{r}_2$, compute and store $M \cdot \mathbf{r}_2$ while making its pass over M, and then execute the evaluation protocol (during which the prover only needs to know $M \cdot \mathbf{r}_2$.

Executing the protocol in logarithmic space is more involved and requires streaming the matrix M in column-major order, and the commitment key once per column of M in each round of the Bulletproof protocol (note that the prover can stream the commitment key per Remark 6.1). This essentially follows from the following two observations:

- [BHR+20] showed how to implement the Bulletproofs prover in logarithmic space with one pass over the committed vector in each round of Bulletproofs.
- If the Hyrax prover streams over the entries of M in column-major order, it can iteratively produce each of the entries of the vectors to which Bulletproofs is applied, namely $\mathbf{w}_1$, $\mathbf{r}_1$ and $\mathbf{g}$.

We now provide full details for completeness.

Overview of the prover algorithm. Observe that the commitment to $M \cdot \mathbf{r}_2$ using commitment key $\mathbf{g} = (g_1, \ldots, g_{\sqrt{n}})$ can be rewritten as follows:

$$\sum_{i,j \in \{1,\ldots,\sqrt{n}\}} (m_{i,j} \cdot r_{2,j}) \cdot g_i.$$

This sum can be computed the prover in logarithmic space by maintaining an accumulator variable that aggregates the terms corresponding to each $m_{i,j}$. The entries of the matrix M are streamed column-wise while the generators are streamed sequentially.

It is trivial for the prover to iteratively generate the entries of $\mathbf{r}_2$ in lexicographic order, with $O(\sqrt{n} \log n)$ total time and $O(\log n)$ space. This fact is sufficient to achieve an $O(n \log n)$ prover runtime in computing Hyrax evaluation proofs, but, as discussed earlier in Section 4, the $\log n$ factor can be avoided with slightly more care. See [CFFZE24, Section 4.1] or [Rot24, Appendix A] for details. Hence, when processing $m_{i,j}$, $r_{2,j}$ can also be efficiently computed, allowing the calculation of $(m_{i,j} \cdot r_{2,j}) \cdot g_i$, which is then added to the accumulator variable.

Furthermore, at each round, for the prover to compute the cross-terms (Equation (12)), vector partitioning is interleaved. Instead of conventional left and right halves, we define the left half of the generator vector at round 1 as $(g_1, g_3, \ldots, g_{\sqrt{n}-1})$ and the right half as $(g_2, g_4, \ldots, g_{\sqrt{n}})$ (this is the same approach used

to render the Bulletproofs prover streaming in [BHR+20]). The entries of M are similarly interleaved.[20] This arrangement enables the prover to compute cross-terms by streaming the entries of M column-wise, and the $\sqrt{n}$ generators sequentially (once per column), and storing at most a constant number of them in memory at a time.

Complete description of the prover algorithm. In general, for the i-th round, the k-th element of the generator vector G_i depends on the following elements of $\mathbf{g}$ for $k \in \{1, \ldots, \sqrt{n}/2^{i-1}\}$: g_ℓ for $\ell \in \{2^{i-1} \cdot (k-1) + 1, \ldots, 2^{i-1} \cdot k\}$. Specifically, let $\ell_1, \ell_2, \ldots, \ell_i$ denote the last i bits of ℓ, with ℓ_1 being the least significant bit. Then, the generator vector G_i in the i-th round can be expressed as

$$g_{i,k} = \sum_{\ell \in \{2^{i-1} \cdot (k-1)+1, \ldots, 2^{i-1} \cdot k\}} \prod_{s \in \{1, \ldots, i-1\}} ((1 - \ell_s) \cdot \alpha_s^{-1} + \ell_s \cdot \alpha_s) \cdot g_\ell. \tag{13}$$

Similarly, the k-th element of the witness vector $\mathbf{w}_i$ at the i-th round is given by

$$w_{i,k} = \sum_{\ell \in \{2^{i-1} \cdot (k-1)+1, \ldots, 2^{i-1} \cdot k\}} \prod_{s \in \{1, \ldots, i-1\}} ((1 - \ell_s) \cdot \alpha_s + \ell_s \cdot \alpha_s^{-1}) \cdot w_{1,\ell}.$$

Since the initial witness vector is given by $\mathbf{w}_1 = M \cdot \mathbf{r}_2$, it follows that

$$w_{1,\ell} = \sum_{j \in \{1, \ldots, \sqrt{n}\}} m_{\ell,j} \cdot r_{2,j}.$$

At each round i, the prover must compute the cross-terms $\langle \mathbf{w}_{i,L}, G_{i,R} \rangle$, $\langle \mathbf{w}_{i,R}, G_{i,L} \rangle$, $y_{i,L}$, and $y_{i,R}$. We explain the computation of $\mathbf{w}_{i,L} \cdot G_{i,R}$ and $\mathbf{w}_{i,R} \cdot G_{i,L}$, and similar arguments hold for the computation of $y_{i,L}$, and $y_{i,R}$. The cross-terms $\mathbf{w}_{i,L} \cdot G_{i,R}$ and $\mathbf{w}_{i,R} \cdot G_{i,L}$ requires computing the following individual products for $k \in \{1, \ldots, \sqrt{n}/2^i\}$:

$$w_{i,2k} \cdot g_{i,2k+1} \quad \text{and} \quad w_{i,2k+1} \cdot g_{i,2k}.$$

These terms can be efficiently computed iteratively by maintaining accumulator variables that update the respective sums for $\langle \mathbf{w}_{i,L}, G_{i,R} \rangle$ and $\langle \mathbf{w}_{i,R}, G_{i,L} \rangle$.

To illustrate the process, consider the computation of $w_{i,2k} \cdot g_{i,2k+1}$; a similar procedure applies to compute $w_{i,2k+1} \cdot g_{i,2k}$. Expanding the expression using the previously defined formulae:

$$w_{i,2k} \cdot g_{i,2k+1} = \left(\sum_{\ell \in \{2^{i-1} \cdot (2k-1)+1, \ldots, 2^{i-1} \cdot 2k\}} \prod_{s \in \{1, \ldots i\}} \left((1 - \ell_s) \cdot \alpha_s^{-1} + \ell_s \alpha_s \right) \cdot \left(\sum_{j \in \{1, \ldots, \sqrt{n}\}} m_{\ell,j} \cdot r_{2,j} \right) \right) \cdot g_{i,2k+1}.$$

Rearranging the terms, we obtain:

$$w_{i,2k} \cdot g_{i,2k+1} = \sum_{j \in \{1, \ldots, \sqrt{n}\}} \left(\sum_{\ell \in \{2^{i-1} \cdot (2k-1)+1, \ldots, 2^{i-1} \cdot 2k\}} \prod_{s \in \{1, \ldots, i\}} \left((1 - \ell_s) \cdot \alpha_s^{-1} + \ell_s \alpha_s \right) \cdot m_{\ell,j} \cdot r_{2,j} \right) \cdot g_{i,2k+1}.$$

This expression reveals that the computation naturally decomposes across columns. Specifically, the entries of M can be streamed column-wise, enabling an efficient accumulation of terms. For a fixed column j, the inner sum

$$\sum_{\ell \in \{2^{i-1} \cdot (2k-1)+1, \ldots, 2^{i-1} \cdot 2k\}} \prod_{s \in \{1, \ldots, i\}} \left((1 - \ell_s) \cdot \alpha_s^{-1} + \ell_s \alpha_s \right) \cdot m_{\ell,j} \cdot r_{2,j}$$

can be computed by maintaining an accumulator that is iteratively updated as the coefficients $m_{\ell,j}$ are streamed. Similarly, the generator $g_{i,2k+1}$ can be computed by streaming the generators sequentially and accumulating the corresponding terms using the recurrence relation (Equation 13). Once all 2^{i-1} terms have been accumulated, the final product $w_{i,2k} \cdot g_{i,2k+1}$ is computed in constant space. This process requires streaming the matrix M once and the generator vector $\sqrt{n}$ times in each round of the Bulletproofs protocol.

[20] This process of generating the two halves based on the value of the least significant bit and proceeding round-by-round toward the most significant bit is similar to the approach used in the sum-check protocol.

6.3 Other Curve-Based Commitment Schemes

The second variation of Hyrax described in Section 6.2.1 above uses two techniques. The first is to leverage that any $(\log n)$-variate multilinear polynomial p, evaluated at a point $r \in \mathbb{F}^{\log n}$, can be expressed as a vector-matrix-vector product per Equation (10). Leveraging this fact typically leads to commitment keys that are only $O(\sqrt{n})$ in size, and evaluation proofs can be computed with roughly $O(\sqrt{n})$ group operations. The second technique leverages logarithmic-round protocols reminiscent of Bulletproofs/IPA to keep evaluation proofs logarithmic in size.

Many other polynomial commitment schemes leverage one or both of these techniques. Via the approach of Section 6.2.1 above, all of them are also amenable to a streaming prover. A brief summary follows.

- Bulletproofs/IPA [BCC+16, BBB+18] itself is amenable to a streaming prover as already shown by Block et al. [BHR+20]. Note that Bulletproofs/IPA has a linear-sized commitment key. If the prover does not wish to store this key, it can generated incrementally (logarithmically many times). However, this is slow, both asymptotically and concretely, as discussed in Section 1.3.1. Note that it is possible to reduce the commitment key size of Bulletproofs by a logarithmic factor with only a constant-factor increase in proof size and a low-order increase in verifier time–see [ST25, Section 3.1.1] for details.

- HyperKZG is a multilinear analog of KZG commitments [KZG10]. In HyperKZG, when committing to a polynomial p, the prover interprets the evaluations of p at all inputs in $\{0,1\}^{\log n}$ as the coefficients of a *univariate* polynomial and applies the KZG commitment procedure to that univariate polynomial. This involves a single MSM. Evaluation proofs in HyperKZG consist of a logarithmic-round protocol that is reminiscent of Bulletproofs and used to relate the requested evaluation of the multilinear polynomial p to the univariate polynomial that was actually KZG-committed. The evaluation proofs can be generated in a streaming manner using the techniques of [BHR+20].

  However, note that HyperKZG has a linear-sized SRS. Since the SRS cannot be generated locally, it must either be stored explicitly by the prover or repeatedly streamed from some repository. It is possible to reduce the size of the HyperKZG SRS to $O(T/\log T)$ with a constant-factor increase in proof size and verifier time (see [ST25, Section 3.1.1] for details). Even so, a $O(T/\log T)$ SRS size is limiting in terms of scaling to large cycle counts T without resorting to recursion. Nonetheless, our techniques can still achieve substantial concrete improvements to zkVM prover space when folding schemes are combined with HyperKZG commitments.

- Dory [Lee21] and Kopis [SL20]. In both Dory and Kopis, commitments are computed as follows. The prover first computes the Hyrax commitment, but since this commitment is large (comprising one group element per column of the $\sqrt{n} \times \sqrt{n}$ matrix representing the committed polynomial), Dory and Kopis instead have the prover *commit to the Hyrax commitment* using AFGHO commitments [AFG+10]. This entails computing an *inner pairing product*: if the Hyrax commitment is $\mathbf{h} = (h_1, \ldots, h_{\sqrt{n}})$, then the Dory and Kopis commitments equal $\prod_{i=1}^{n} e(h_i, q_i)$, where e denotes a bilinear map, $\prod$ denotes the group operation within the target group of a pairing-friendly group, and $q_1, \ldots, q_{\sqrt{n}}$ are randomly generated elements of $\mathbb{G}_2$ (i.e., the AFGHO commitment key). Just as the Hyrax commitment is computable in small space, so is an AFGHO commitment thereof.

  The evaluation protocol of Kopis explicitly invokes Bulletproofs/IPA twice (Kopis achieves logarithmic-size proofs but has $O(\sqrt{n})$ verifier time). Dory applies a Bulletproofs-like protocol to achieve logarithmic verifier time. As with the second variant of Hyrax, both Dory and Kopis evaluation proofs can be computed in small space. If $O(\sqrt{n})$ space is available to the prover, no modification to the linear-space prover implementation is needed. Alternatively, evaluation proofs can be computed in logarithmic space by streaming the committed polynomial once per round (so $O(\log n)$ times in total) and the commitment key once per column per round (so, $\sqrt{n} \cdot \log(\sqrt{n})$ times in total).

  All of the above also holds for BMMTV [BMM+21], a predecessor of Dory that also achieves logarithmic verifier costs and $O(\sqrt{n})$ commitment key size, but is not transparent. As with any non-transparent commitment scheme, the commitment key (i.e., the SRS) cannot be generated locally, so

the prover either needs to store it explicitly (using $O(\sqrt{n})$ space), or else stream it repeatedly from some repository.

7 Jolt Proving in Small Space

In this section, we present a space-efficient honest prover algorithm for the Jolt zkVM, demonstrating that verification can be achieved with minimal memory requirements while accepting a trade-off in runtime performance.

Theorem 7.1. *For any RISC-V program executed in T cycles that uses K words of RAM, there exists an honest prover algorithm for Jolt that uses $S = O(K + \log T)$ space and runs in $O(T \log T)$ time, plus the time to produce $O(1)$ polynomial evaluation proofs for the relevant polynomial commitment scheme. Here, the evaluation proof is for a committed polynomial of size $O(T)$ (for the version of Jolt using Lasso and Spice) or $O(K^{1/d} \cdot T)$ (for the version of Jolt using Twist and Shout; here, d is the dimensionality parameter used in Twist and Shout, see Section 3.3 for details).*

The Jolt proof system consists of three interconnected SNARKs, as described in Section 3.2: Spartan applied to Jolt's R1CS instance, a lookup argument (either Shout or Lasso) used to prove correct primitive instruction execution and used to process lookups into bytecode), and a memory-checking argument for read/write memory (either Twist or Spice), used to process the VM's reads and writes to registers and RAM.

Per Observation 3.5, the Jolt prover first executes the RISC-V program and iteratively generates the necessary witness vectors for these three component proof systems. As the witness vectors are generated, they are committed using a PCS for multilinear polynomials. We discuss polynomial commitment schemes that enable small-space committing and evaluation proof computation in Section 6.

Hence, to establish the Jolt prover's space efficiency, we need only demonstrate that all three component arguments possess small-space prover algorithms. Small-space Spartan proving is discussed in Section 4. Small-space proving for Twist and Shout is discussed in Section 5, while small-space proving for Lasso and Spice is discussed in Appendices B-F.

7.1 Estimated Concrete Slowdowns

Recall (Section 3.2 that Jolt consists of three components: (1) an R1CS instance, to which Spartan is applied (appropriately optimized for the highly uniform nature of the constraint system arising in Jolt), (2) a lookup argument, used to prove correct execution of primitive instructions and to perform reads into program bytecode and (3) a memory-checking argument for read/write memory, used to ensure that the prover correctly maintains the VM's registers and RAM. We now estimate the concrete slowdown that arises from implementing the prover for these three components in small space rather than space $O(T)$.

Let us consider $T = 2^{35}$, $K = 2^{25}$ and focus on the setting of Jolt with Twist and Shout as the memory-checking arguments, using an elliptic curve commitment scheme such as Dory [Lee21]. For this discussion, we consider $\Theta(\sqrt{KT})$ space to be acceptable, justified via the discussion in Section 1.2. Note that with only a modest increase in prover time, our techniques can reduce the prover's space usage considerably further, say to below $\Theta(K^{1/4} \cdot T^{1/2})$ field elements.

High-level summary. What is the extra work required by the small-space Jolt prover compared to the linear-space prover? It consists primarily of: (1) repeated witness generation and (2) invocation of the small-space sum-check prover algorithm (Theorem 3.3) in the final $\log(T)$ rounds of any invocation of Twist and Shout within Jolt, and in Spartan-proving. The cost of committing to data and computing evaluation proofs is unchanged given the $O(\sqrt{KT})$ space bound.

As explained below, the extra cost of repeated witness generation is minimal primarily due to the fact that the first time witness generation is performed it is typically done serially while subsequent runs of

witness generation can be parallelized. The use of the small-space sum-check prover algorithm contributes about $12T \log T$ extra field multiplications for the prover in total. For $T \approx 2^{35}$, this is roughly $400T$ extra field multiplications.

As we explain below, the linear-space prover performs at least about $500T$ and at most about $900T$ field operations in total.[21] Hence, this slowdown is a small relative increase in total prover time, far less than a factor of 2. Details on the overall cost and slowdown for each component of Jolt are provided below.

Cost of repeated witness generation. The subroutine that determines the number of times witness generation must occur is Twist applied to prove correct execution of the machine's RAM. (Spartan would be the bottleneck if not for the optimization described in Remark 4.1).

Implementing the Twist prover in $O(K + T^{1/2})$ (much less then the $O(K + \sqrt{KT})$ space bound we target here) requires performing witness generation at most $\log K + (1/2) \cdot \log T \approx 40$ times. As discussed in Section 4, the first time witness generation occurs it is typically done sequentially, but subsequent runs of witness generation can be heavily parallelized. If there are 16 threads, then running witness generation 40 times leads to an increase in total (parallel) runtime by a factor of about $40/16 < 3$. Since witness generation is currently less than 5% of Jolt's prover time, this entails less than a 15% increase in overall prover time.

Spartan. In Spartan as applied in Jolt, a linear-space proving algorithm performs roughly $250T$ field operations. (More precisely, the current Spartan implementation in Jolt performs about $400T$ field operations, as there are about 80 constraints per CPU cycle and Spartan can be implemented with about 5 field operations per constraint [DT24]. However, further optimizations (see Section 3.1.3) are possible–and in the process of being implemented within Jolt—by leveraging that all values arising in all constraints fit in a single machine word. These optimizations reduce the Spartan prover cost from about $400T$ field operations down to about $250T$. More importantly for our work, as discussed in Section 3.1.3, these "small value" sum-check optimizations are directly compatible with a small-space prover (for the first several rounds of the protocol). Hence, the slowdown in Spartan proving that arises from keeping the prover's space usage bounded is modest. The precise slowdown depends on how many rounds the "small-value sum-check" technique of Bagad et al. [BDT24] (detailed in Section 3.1.3 above) can be used before becoming slower than the standard linear-time (and space) prover algorithm. Preliminary experiments suggest this crossover point does not happen for at least 8 rounds. This ensures that extra prover work done by the small-space implementation is bounded by at most about $40T$ field multiplications (this comes from using the small-space sum-check proving algorithm of Theorem 1 for rounds, say, 9-18 in the first invocation of the sum-check protocol in Spatan, before switching over to the linear-space, linear-time algorithm).

Commitment costs and evaluation proofs. The $\Theta(\sqrt{KT})$ space bound is enough to run the Dory commitment scheme and compute evaluation proofs in exactly the same manner as in the linear-space case. The overall cost of commitments can be estimated as follows. The Jolt prover today (with Spice and Lasso) commits to up to 61 non-zero values per cycle, but this will fall to less than 30 non-zero committed values per cycle after the switch to Twist and Shout. Of these roughly 30 committed non-zero values per cycle, at least 8 are equal to 1. Of the remaining 22 (or fewer) committed non-zero values, all but one are in the set of $\{0, 1, \ldots, 2^{32} - 1\}$, and most are in the smaller set $\{0, 1, \ldots, 2^{16} - 1\}$. Hence, a crude estimate for total commitment costs is at most 50 group operations per RISC-V cycle, translating to roughly 350 field operations per RISC-V cycle [GW20]. The cost of computing evaluation proofs in Jolt when Dory is used as the polynomial commitment scheme is at most $30T$ field operations plus $O(1)$ multi-pairings of size $O(\sqrt{KT})$ and a similar number of scalar multiplications in $\mathbb{G}_1$ and $\mathbb{G}_2$. For $K = 2^{25}$ and $T = 2^{35}$, these cryptographic group operations will not be a dominant cost for the prover.

[21] A 256-bit field multiplication requires less than 80 CPU cycles. So a prover requiring 900 field operations per cycle translates to a prover slowdown relative to native execution of about $900 \cdot 80 = 72,000$. This is broadly consistent with the observed performance of Jolt today combined with anticipated speedups due to incorporation of Twist and Shout in place of Spice and Lasso, and other known optimizations.

Shout for instruction execution. The Shout prover for instruction execution lookups performs about $40T$ field multiplications[22] in the case of linear-space proving. The first 64 (out of $64 + \log(T) \approx 100$) rounds of this protocol is naturally space-efficient for the prover. The only prover runtime overhead when running in small space comes in the final $\log(T)$ rounds (specifically, the added cost of using the small-space proving algorithm from Theorem 3.3 for the first half of the final $\log T$ rounds is at most $4 \cdot (1/2) \cdot T \cdot \log T = 2T \log T$ field operations). Overall, the linear-space prover implementation of Shout incurs roughly $40T$ field multiplications for the prover [ST25], with the small-space implementation requiring about $40T + 2T \log T \approx 110T$ field multiplications when $T = 2^{35}$.

Shout for bytecode lookups. As long as the number of instructions in the RISC-V bytecode is noticeably less than T, then the Shout prover for bytecode lookups performs only about $5T$ field operations in total when using linear space [ST25]. A small-space prover implementation would involve at most an additional $2T \log T$ field operations.

Twist for registers. The Twist prover applied to the 32 RISC-V registers performs, in the worst case, about $35T$ field operations when using linear space [ST25]. This increases by at most an additional $4T \log T$ field operations when run in small space ($2T \log T$ for the read-checking sum-check and $2T \log T$ for the write-checking sum-check).

Twist for RAM. In the worst case, Twist for RAM incurs less than $150T$ field multiplications for the prover when the RAM size K equals 2^{25}. The Jolt prover only actually incurs field multiplications within Twist-for-RAM for RISC-V cycles performing a load or store operation (i.e., a read or write to/from RAM). Moreover, for such cycles, the Jolt prover pays much less than 110 field multiplications within Jolt's application of Shout to instruction-execution. On top of the above, the Twist prover incurs fewer field multiplications when load and store operations are local (meaning many of them access memory cells that were recently read or written).

For all of these reasons, Jolt's application of Twist to RAM is likely to cost substantially less than $150T$ field multiplications for the prover. To be conservative, we add these $150T$ field multiplications to our prover time estimate anyway.

A small-space implementation of Twist for reads and writes to RAM increases the number of field operations the prover performs by an additional $4T \log T$ field operations.

In total, we conservatively estimate that the Jolt prover (once Twist and Shout are incorporated), when run in linear space and applied to a RISC-V executions with $T = 2^{35}$ cycles and a memory size of $k = 2^{25}$, incurs about 900 field operations per cycle in the worst case (and fewer in typical cases). A small-space prover implementation increases this cost by roughly $12 \log T \approx 400$ field operations per cycle.

Disclosures. Justin Thaler is a Research Partner, and Michael Zhu is a research engineer, at a16z crypto. They are investors in various blockchainbased platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see https: //www.a16z.com/disclosures/.)

[22] This $40T$ estimate of field multiplications within Shout-for-instruction-execution ignores a cost that is independent of T. Let us call this ignored quantity the "fixed cost" of Shout. This fixed cost is a multiple of $C2^{64/C}$ for a parameter C used in the sparse-dense sum-check protocol (see Section 5 for details) or the prefix-suffix inner product protocol of Appendix A. For $T \geq 2^{20}$, C can be set to 4 and the fixed cost will not be a significant contributor to prover time. Accordingly, our $40T$ estimate assumes $C = 4$ and that the fixed cost is insignificant. For smaller CPU executions (i.e., $T < 2^{20}$), C can be set to a larger number, such as 8. This increases the non-fixed cost of Shout proving to roughly $80T$ but substantially reduces the fixed cost. Our general focus in this work is on scaling to large CPU executions (i.e., $T \approx 2^{30}$ and up), and in this setting the $40T$ estimate for Shout is accurate.

References

[AFG+10] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-preserving signatures and commitments to group elements. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*, pages 209–236. Springer, 2010. 33

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, pages 2087–2104, 2017. 28

[AS24] Arasu Arun and Srinath T. V. Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. *IACR Cryptol. ePrint Arch.*, page 1605, 2024. 2

[AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2024. 4

[AT+24] Arasu Arun, Justin Thaler, et al. Simplified second-Spartan-sum-check (via simpler uniform R1CS product-vector MLE) #347. https://github.com/a16z/jolt/issues/347, May 2024. GitHub issue opened on May 7, 2024. Accessed: 28 Feb 2025. 24, 26

[AY25] Gal Arnon and Eylon Yogev. Towards a white-box secure fiat-shamir transformation. Cryptology ePrint Archive, Paper 2025/329, 2025. 9

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. 2018. 9, 30, 33

[BBHV22] Laasya Bangalore, Rishabh Bhadauria, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. On black-box constructions of time and space efficient sublinear arguments from symmetric-key primitives. In *Theory of Cryptography - 20th International Conference, TCC 2022, Chicago, IL, USA, November 7-10, 2022, Proceedings, Part I*, volume 13747 of *Lecture Notes in Computer Science*, pages 417–446. Springer, 2022. 28

[BC23] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special-sound protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 77–110. Springer, 2023. 2

[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*, pages 269–301. Springer, 2024. 2

[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. 2016. 9, 30, 33

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic snarks for diverse environments. In *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 427–457. Springer, 2022. 10

[BCPR16] Éric Brier, Cyrille Chevallier-Mames, David Page, and Matthieu Rivain. Indifferentiable hashing to elliptic and hyperelliptic curves. *Journal of Cryptology*, 29(1):102–146, 2016. 7

[BDT24] Suyash Bagad, Yuval Domb, and Justin Thaler. The sum-check protocol over fields of small characteristic. *IACR Cryptol. ePrint Arch.*, page 1046, 2024. 6, 11, 17, 18, 20, 35

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from DARK compilers. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 677–706. Springer, 2020. 10

[BHR+20] Alexander R Block, Justin Holmgren, Alon Rosen, Ron D Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In *Theory of Cryptography Conference*, pages 168–197. Springer, 2020. 5, 7, 9, 10, 31, 32, 33

[BHR+21] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV*, volume 12828 of *Lecture Notes in Computer Science*, pages 123–152. Springer, 2021. 10

[BMM+21] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 65–97. Springer, 2021. 33

[BMM+24] Anubhav Baweja, Pratyush Mishra, Tushar Mopuri, Karan Newatia, and Steve Wang. Scribe: Low-memory snarks via read-write streaming. *IACR Cryptol. ePrint Arch.*, page 1970, 2024. 10

[BMNW24] Benedikt Bünz, Pratyush Mishra, Wilson Nguyen, and William Wang. Arc: Accumulation for reed–solomon codes. *Cryptology ePrint Archive*, 2024. 2

[BTVW14] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. *IACR Cryptol. ePrint Arch.*, page 846, 2014. 3, 4, 5, 9, 10, 14

[But25] Vitalik Buterin. Yes, it's possible to scale blockchains to billions of users without sacrificing decentralization. https://vitalik.eth.limo/general/2025/02/14/l1scaling.html, February 2025. Accessed: 2025-02-26. 5

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 499–530. Springer, 2023. 10

[CCH+19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N Rothblum, Ron D Rothblum, and Daniel Wichs. Fiat-shamir: from practice to theory. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1082–1090, 2019. 12

[CFFZE24] Alessandro Chiesa, Elisabetta Fedele, Giacomo Fenzi, and Andrew Zitek-Estrada. A time-space tradeoff for the sumcheck prover. *Cryptology ePrint Archive*, 2024. 26, 27, 31, 45

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 90–112. ACM, 2012. 3, 4, 5, 8, 9, 14

[CTY11] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. *Proc. VLDB Endow.*, 5(1):25–36, 2011. 3, 4, 5, 6, 7, 9, 14, 15

[DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. *IACR Cryptol. ePrint Arch.*, page 1784, 2023. 2, 23, 24, 28

[DP24] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. *IACR Cryptol. ePrint Arch.*, page 504, 2024. 23, 28

[DT24] Quang Dao and Justin Thaler. More optimizations to sum-check proving. *Cryptology ePrint Archive*, 2024. 5, 26, 35

[EGS+24] Liam Eagen, Ariel Gabizon, Marek Sefranek, Patrick Towa, and Zachary J. Williamson. Stackproofs: Private proofs of stack and contract execution using protogalaxy. Cryptology ePrint Archive, Paper 2024/1281, 2024. 2

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1987. 8

[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. 8

[GLS+23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II*, volume 14082 of *Lecture Notes in Computer Science*, pages 193–226. Springer, 2023. 28

[Gro24] Jens Groth. Memory checking in IVC-based zkVMs, 2024. ZKSummit 12, available at https://www.youtube.com/watch?v=kzSYNFh4uQ0. 2

[Gru24] Angus Gruen. Some improvements for the PIOP for zerocheck. *IACR Cryptol. ePrint Arch.*, page 108, 2024. 15, 17

[GW20] Ariel Gabizon and Zachary J Williamson. Proposal: The turbo-plonk program syntax for specifying snark programs, 2020. 35

[Ham13] Mike Hamburg. Elligator: Elliptic-Curve Points Indistinguishable from Uniform Random Strings. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 710–727. Springer, 2013. 7

[KNS24] Tohru Kohrita, Maksim Nikolaev, and Javier Silva. BOIL: Proof-carrying data from accumulation of correlated holographic iops. *Cryptology ePrint Archive*, 2024. 2

[KRS25] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. How to prove false statements: Practical attacks on fiat-shamir. *IACR Cryptol. ePrint Arch.*, page 118, 2025. 8

[KS24] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. In *Annual International Cryptology Conference*, pages 345–379. Springer, 2024. 2

[KST22] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 359–388. Springer, 2022. 2

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010. 10, 13, 33

[Lee21] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part II*, volume 13043 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 2021. 3, 5, 6, 8, 33, 34

[LS24] Hyeonbum Lee and Jae Hong Seo. On the security of nova recursive proof system. Cryptology ePrint Archive, Paper 2024/232, 2024. 2

[NS25] Wilson Nguyen and Srinath Setty. Neo: Lattice-based folding scheme for ccs over small fields and pay-per-bit commitments. *Cryptology ePrint Archive*, 2025. 4

[PP24] Christodoulos Pappas and Dimitrios Papadopoulos. Sparrow: Space-efficient zksnark for data-parallel circuits and applications to zero-knowledge decision trees. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3110–3124. ACM, 2024. 10

[Rot24] Ron D Rothblum. A note on efficient computation of the multilinear extension. *Cryptology ePrint Archive*, 2024. 26, 31, 45

[RR25] Gyumin Roh and Ron Rothblum. SP1 V4 Turbo: Memory argument via elliptic curve based multiset hashing, 2025. Available at https://github.com/succinctlabs/sp1/blob/dev/book/static/SP1_Turbo_Memory_Argument.pdf. 2

[SAGL18] Srinath T. V. Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 339–356. USENIX Association, 2018. 4, 20, 48

[Set20] Srinath T. V. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020. 8, 9, 20

[SL20] Srinath T. V. Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zksnarks. *IACR Cryptol. ePrint Arch.*, page 1275, 2020. 8, 33, 49

[ST25] Srinath T. V. Setty and Justin Thaler. Twist and shout: Faster memory checking arguments via one-hot addressing and increments. *IACR Cryptol. ePrint Arch.*, page 105, 2025. 2, 3, 4, 5, 7, 8, 20, 26, 27, 28, 33, 36, 44

[STW23] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Customizable constraint systems for succinct arguments. *IACR Cryptol. ePrint Arch.*, page 552, 2023. 25, 45

[STW24] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Unlocking the lookup singularity with lasso. In *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI*, volume 14656 of *Lecture Notes in Computer Science*, pages 180–209. Springer, 2024. 4, 6, 7, 20, 26, 27, 42, 44, 53

[Sze24] Alan Szepieniec. DEEP commitments and their applications. Cryptology ePrint Archive, Paper 2024/1752, 2024. 2

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2013. 8, 9, 15

[Tha22] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. 2022. 14

[VSBW13] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 223–237. IEEE, 2013. 30, 44, 45

[WHV24] Ruihan Wang, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. Ligetron: Lightweight scalable end-to-end zero-knowledge proofs post-quantum zk-snarks on a browser. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 1760–1776. IEEE, 2024. 3, 10

[WJB+17] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2071–2086, 2017. 9

[WJSS23] Riad Wahby, Simon Josefsson, Douglas Stebila, and Benjamin Smith. Hashing to Elliptic Curves. Internet Engineering Task Force, RFC 9380, April 2023. IRTF Crypto Forum Research Group. 7

[WTS+18] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943. IEEE, 2018. 6, 8, 29, 30

[XZZ+19] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019. 9

[ZCL+24] Shuangjun Zhang, Dongliang Cai, Yuan Li, Haibin Kan, and Liang Zhang. Epistle: Elastic succinct arguments for plonk constraint system. *IACR Cryptol. ePrint Arch.*, page 872, 2024. 10

A Prefix-Suffix Inner Product Protocol

In this section we consider the sum-check protocol applied to compute the inner product of two vectors u and a in $\mathbb{F}^N$. That is, the sum-check protocol is applied to compute the sum

$$\sum_{x \in \{0,1\}^{\log N}} \widetilde{u}(x) \cdot \widetilde{a}(x), \tag{14}$$

where $\widetilde{u}$ is the multilinear extension of u and $\widetilde{a}$ is the multilinar extension of a. We seek a linear-time streaming prover. To achieve this, we can let u be an arbitrary vector in $\mathbb{F}^N$, but we must assume that $\widetilde{a}$ has some structure to it. We refer to the structural property we require of $\widetilde{a}$ as *prefix-suffix structure.*

Definition A.1. Let $i \leq \log n$ be an integer. A multilinear polynomial $\widetilde{a}(x_1, \ldots, x_{\log N})$ has *prefix-suffix structure for cutoff i with k terms* if there exist multilinear polynomials $\mathsf{prefix}_1, \ldots, \mathsf{prefix}_k \colon \mathbb{F}^i \to \mathbb{F}$ and $\mathsf{suffix}_1, \ldots, \mathsf{suffix}_k \colon \mathbb{F}^{\log(N)-i} \to \mathbb{F}$, such that the following property holds:

$$\widetilde{a}(x_1, \ldots, x_{\log N}) = \sum_{j=1}^{k} \mathsf{prefix}_j(x_1, \ldots, x_i) \cdot \mathsf{suffix}_j(x_{i+1}, \ldots, x_{\log N}). \tag{15}$$

Consider the sum-check protocol applied to compute $\sum_{x \in \{0,1\}^{\log N}} \widetilde{u}(x) \cdot \widetilde{a}(x)$. Let $C \geq 1$ be a positive integer. Suppose for simplicity that C divides $\log N$ and that $\widetilde{a}(x)$ has prefix-suffix structure with k terms, for the cutoffs $i = \frac{\log(N)}{C}, \ldots, \frac{(C-1)\log(N)}{C}$. Below, we describe an algorithm we call the *prefix-suffix inner product protocol*. This algorithm implements the honest prover in the sum-check protocol applied to compute Expression (14) when $\widetilde{a}$ meets Definition A.1. It uses space $O(k \cdot C \cdot N^{1/C})$ and makes C passes over the

vectors u and a. In our key applications in this paper (the pcnext and $\widetilde{M}$-evaluation sum-checks of Sections 4.2 and 3.3 respectively), the prover performs $O(C \cdot k \cdot N)$ field multiplications in general. In fact, if m denotes the number of non-zero entries of u, then the prover's runtime is $O(C \cdot k \cdot m)$.

Remark A.2. The number of non-zero entries m of u is called the *sparsity* of u. The prover's runtime scales linearly with the sparsity of u (so long as C is constant and $\widetilde{a}$ is prefix-suffix structured). This property is the motivation behind the name "sparse-dense sum-check protocol" coined in [STW24, Appendix G]. Due to the fact that they are small-space for the prover, both the sparse-dense sum-check prover algorithm of [STW24] and the prefix-suffix inner product prover algorithm are useful even when u is *dense* (i.e., $m = \Theta(N)$). This is the case in its applications to the pcnext-evaluation and $\widetilde{M}$-evaluation sum-checks in this paper.

Algorithm overview. Let us focus on the case $k = 1$. The case of general k follows from linearity. That is, when applying the sum-check protocol to a sum of k polynomials, say to compute $\sum_{j=1}^{k} \sum_{x \in \{0,1\}^{\log N}} g_k(x)$, the prover's message in each round is simply the sum of the messages obtained by applying the sum-check protocol independently in parallel to each term $g_1, \ldots, g_k$.

The case $C = 1$ follows from the standard linear-time sum-check prover implementation (Section 3.1.1). We now turn to the case of $C \geq 2$.

The protocol proceeds in C stages, with each stage covering $\log(N)/C$ rounds of the sum-check protocol. In each stage, the prover first makes a single pass over u and a, and builds an array Q with $N^{1/C}$ entries, each labeled by a vector $y \in \{0,1\}^{\log(N)/C}$. The array Q can be built at the start of each stage with constant time spent per non-zero entry of u. Once Q is built, the prover can get through all $\log(N)/C$ rounds of the stage in $O(N^{1/C})$ time.

Description of Stage 1. Since $\widetilde{a}$ is prefix-suffix structured for cutoff $\log(N)/C$ with $k = 1$ term, we can write $\widetilde{a}(y, x) = \mathsf{prefix}(y) \cdot \mathsf{suffix}(x)$ for any $y \in \mathbb{F}^{\log(N)/C}$ and $x \in \mathbb{F}^{\log(N) - \log(N)/C}$.

In Stage 1 the array-building procedure works as follows. The y'th entry of Q stores the sum of all $\widetilde{u}(x) \cdot \mathsf{suffix}(x)$ evaluations, as x ranges over all vectors in $\{0,1\}^{\log(N)}$ whose first $\log(N)/C$ bits are equal to y. In other words,

$$Q[y] = \sum_{x=(x_1,\ldots,x_C) \in \left(\{0,1\}^{\log(N)/C}\right)^C \colon x_1 = y} \widetilde{u}(x) \cdot \mathsf{suffix}(x_2, \ldots, x_C).$$

The prover also builds an array P of size $N^{1/C}$ that stores all values $\mathsf{prefix}(y)$ for $y \in \{0,1\}^{\log(N)/C}$.

We claim the prover's message in each of the first $\log(N)/C$ rounds of the sum-check protocol applied to compute

$$\sum_{x \in \{0,1\}^{\log N}} \widetilde{u}(x) \cdot \widetilde{a}(x)$$

is identical to the prover's messages in the sum-check protocol applied to compute

$$\sum_{y \in \{0,1\}^{\log(N)/C}} \widetilde{P}(y) \cdot \widetilde{Q}(y) \tag{16}$$

Here, $\widetilde{P} = \mathsf{prefix}$ and $\widetilde{Q}(y)$ denote the multilinear extensions of the functions that maps $y \in \{0,1\}^{\log(N)/C}$ to $P[y]$ and $Q[y]$ respectively.

Indeed, if $r_1, \ldots, r_{i-1}$ denotes the random verifier challenges selected in rounds $1, \ldots, i$, then the sum-check prover's message in round i specifies, for each $c \in \{0,2\}$, the value:

$$\sum_{x \in \{0,1\}^{\log N - i}} \widetilde{u}(r_1, \ldots, r_{i-1}, c, x) \cdot \widetilde{a}(r_1, \ldots, r_{i-1}, c, x)$$

$$= \sum_{y \in \{0,1\}^{\log(N)/C-i}} \sum_{x_2,\ldots,x_C \in \{0,1\}^{\log(N)/C}} \widetilde{u}(r_1,\ldots,r_{i-1},c,y,x_2,\ldots,x_C) \cdot \mathsf{prefix}(r_1,\ldots,r_{i-1},c,y) \cdot \mathsf{suffix}(x_2,\ldots,x_C)$$

$$= \sum_{y \in \{0,1\}^{\log(N)/C}} \widetilde{Q}(r_1,\ldots,r_{i-1},c,y) \cdot \widetilde{P}(r_1,\ldots,r_{i-1},c,y). \tag{17}$$

Expression (17) is precisely the prover's message in round i of the sum-check protocol applied to compute Expression (16).

Accordingly, the prover's messages in rounds $1,\ldots,\log(N)/C$ can be computed by applying the standard linear-time sum-check proving algorithm (Section 3.1.1) to the polynomial $\widetilde{P}(y) \cdot \widetilde{Q}(y)$. Once the arrays P and Q are initialized, this algorithm takes only $O(N^{1/C})$ time with no additional passes over the vectors u and a needed.

Stage $j > 1$. For simplicity of notation let us focus on Stage 2, as all other stages are similar. Let $r = (r_1,\ldots,r_{\log(N)/c})$ denote the verifier's random challenges chosen over the course of the first stage. Since $\widetilde{a}$ is prefix-suffix structured for cutoff $2\log(N)/C$ with $k = 1$ term, we can write

$$\widetilde{a}(r,y,x) = \mathsf{prefix}(r,y) \cdot \mathsf{suffix}(x)$$

for any $y \in \mathbb{F}^{\log(N)/C}$ and $x \in \mathbb{F}^{\log(N)-2\log(N)/C}$. Note here that prefix and suffix are not the same functions that appeared in Stage 1: there, prefix was a function of

$$\log(N)/C$$

variables and suffix of

$$\log(N) - \log(N)/C$$

variables, while here in Stage 2 prefix is a function of

$$2\log(N)/C$$

variables and suffix of

$$\log(N) - 2\log(N)/C$$

variables.

At the start of Stage 2, the array-building procedure works as follows. The y'th entry of Q stores the sum of all $\widetilde{u}(r,y,x) \cdot \mathsf{suffix}(x)$ evaluations, as x ranges over all vectors in $\{0,1\}^{\log(N)-2\log(N)/C}$. In other words,

$$Q[y] = \sum_{x=(x_3,\ldots,x_C) \in \left(\{0,1\}^{\log(N)/C}\right)^{C-2}} \widetilde{u}(r,y,x) \cdot \mathsf{suffix}(x).$$

The prover also builds an array P of size $N^{1/C}$ that stores all values $\mathsf{prefix}(r,y)$ for all $y \in \{0,1\}^{\log(N)/C}$.

Similar to Stage 1, the prover's message in each of the $\log(N)/C$ rounds of Stage 2 is identical to the prover's messages in the sum-check protocol applied to compute

$$\sum_{y \in \{0,1\}^{\log(N)/C}} \widetilde{P}(y) \cdot \widetilde{Q}(y) \tag{18}$$

Accordingly, the prover's messages in Stage 2 can be computed by applying the standard linear-time sum-check proving algorithm (Section 3.1.1) to the polynomial $\widetilde{P}(y) \cdot \widetilde{Q}(y)$. Once the arrays P and Q are initialized, this algorithm takes only $O(N^{1/C})$ time with no additional passes over the vectors u and a needed.

Algorithm runtime. Aside from initializing the arrays P and Q are the start of each of the C stages, the prover spends $O(C \cdot N^{1/C})$ time and space in total.

Initializing Q in each stage j requires evaluating $\widetilde{u}(r, y, x)$ and $\mathsf{suffix}(x)$ for fixed r as (y, x) ranges over $\{0,1\}^{\log(N)/C} \times \{0,1\}^{\log(N) - j\log(N)/C}$. In general, $\widetilde{u}(r, y, x)$ can be computed for all relevant values of y and x in time $O(j \cdot N^{1/C} + m)$ and space $O(C \cdot N^{1/C})$ where recall that m is the number of non-zero entries of u (see [STW24, Section 3.1] for details). The cost of initializing $\mathsf{suffix}(x)$ is application-dependent but typically can be done in constant amortized time per relevant value of x. Initializing P in Stage i requires evaluating $\mathsf{prefix}(r, y)$ as y ranges over $\{0,1\}^{\log(N)/C}$.

We now show that in the applications we consider, both the time and space required only $O(C \cdot N^{1/C})$.

Runtime analysis for $\widetilde{M}$-evaluation sum-check. In the $\widetilde{M}$-evaluation sum-check of Twist (Section 3.3), $\widetilde{u}(j) = \widetilde{\mathsf{Inc}}(r, j)$ and $\widetilde{a}(j) = \widetilde{\mathrm{LT}}(r', j)$. For simplicity, let us consider $C = 2$ (which is sufficient to keep the prover space bounded by $O(T^{1/2})$) and write $j = (j_1, j_2) \in \mathbb{F}^{\log(T)/2} \times \mathbb{F}^{\log(T)/2}$ and similarly write $r' = (r'_1, r'_2)$. Then

$$\widetilde{\mathrm{LT}}(r', j) = \mathsf{prefix}_1(j_1) \cdot \mathsf{suffix}_1(j_2) + \mathsf{prefix}_2(j_1) \cdot \mathsf{suffix}_2(j_2) \tag{19}$$

where

$$\begin{aligned}
\mathsf{prefix}_1(j_1) &= \widetilde{\mathrm{LT}}(r'_1, j_1), \\
\mathsf{suffix}_1(j_2) &= \widetilde{\mathsf{eq}}(r'_2, j_2), \\
\mathsf{prefix}_2(j_1) &= 1, \text{ and} \\
\mathsf{suffix}_2(j_2) &= \widetilde{\mathrm{LT}}(r'_2, j_2).
\end{aligned}$$

To see this, observe that the right hand side of Equation (19) is clearly multilinear and agrees with the right hand side at all inputs in $\{0,1\}^{\log T}$. Indeed, this amounts to the observation that $\mathsf{val}(j) < \mathsf{val}(r')$ if and only if one of the following two situations holds:

- the high order bits of j, namely j_2, are less than the high order bits of r, namely r_2, or
- the high order bits of j and r are equal while the low order bits of j are less than those of r.

To finish showing that the prover runs in $O(T)$ time and $O(\sqrt{T})$ space, we must analyze the time and space required to evaluate suffix_1, suffix_2, prefix_1, and prefix_2 at the relevant inputs to initialize the arrays P and Q. $\mathsf{suffix}_1(j_2) = \widetilde{\mathsf{eq}}(r'_2, j_2)$ can be computed in $O(\sqrt{T})$ time and $O(\sqrt{T})$ space for all $j_2 \in \{0,1\}^{\log(T)/2}$ via standard techniques [VSBW13]. The same goes for $\mathsf{suffix}_1(j_2) = \widetilde{\mathrm{LT}}(r'_2, j_2)$ and $\mathsf{prefix}_1(j_1) = \widetilde{\mathrm{LT}}(r'_1, j_1)$ [ST25, Appendix B], and trivially holds for $\mathsf{prefix}_2(j_1) = 1$.

Runtime analysis for $\widetilde{\mathsf{pcnext}}$-evaluation sum-check. In the $\widetilde{\mathsf{pcnext}}$-evaluation sum-check of Section 4.2, $\widetilde{u}(j) = \widetilde{\mathsf{PC}}(j)$ and $\widetilde{a} = \widetilde{\mathsf{shift}}(r, j)$. For simplicity, let us consider $C = 2$ (which is sufficient to keep the prover space bounded by $O(T^{1/2})$) and write $j = (j_1, j_2) \in \mathbb{F}^{\log(T)/2} \times \mathbb{F}^{\log(T)/2}$ and similarly write $r' = (r'_1, r'_2)$. Then

$$\widetilde{\mathsf{shift}}(r, j) = \mathsf{prefix}_1(j_1) \cdot \mathsf{suffix}_1(j_2) + \mathsf{prefix}_2(j_1) \cdot \mathsf{suffix}_2(j_2) \tag{20}$$

where

$$\begin{aligned}
\mathsf{prefix}_1(j_1) &= \widetilde{\mathsf{shift}}(r_1, j_1), \\
\mathsf{suffix}_1(j_2) &= \widetilde{\mathsf{eq}}(r_2, j_2), \\
\mathsf{prefix}_2(j_1) &= \left(\prod_{\ell=1}^{\log(T)/2} (1 - r_\ell) \cdot j_{1,\ell} \right), \text{ and} \\
\mathsf{suffix}_2(j_2) &= \widetilde{\mathsf{shift}}(r_2, j_2).
\end{aligned}$$

To see this, observe that the right hand side of Equation (19) is clearly multilinear and agrees with the right hand side at all inputs in $\{0,1\}^{\log T}$. Indeed, this amounts to the observation that $\mathsf{val}(j)+1 = \mathsf{val}(r)$ if and only if one of the following two situations holds:

- the low-order bits r_1 of r are equal to $\mathsf{val}(j_1)+1$, and the high-order bits of r and j are equal, or
- the low order bits of j are all-1, the low-order bits of r are all 0, and the high-order bits of r are equal to $\mathsf{val}(j_2)+1$.

To finish showing that the prover runs in $O(T)$ time and $O(\sqrt{T})$ space, we must analyze the time and space required to evaluate suffix_1, suffix_2, prefix_1, and prefix_2 at the relevant inputs to initialize the arrays P and Q. $\mathsf{suffix}_1(j_2) = \widetilde{\mathsf{eq}}(r_2, j_2)$ can be computed in $O(\sqrt{T})$ time and $O(\sqrt{T})$ space for all $j_2 \in \{0,1\}^{\log(T)/2}$ via standard techniques [VSBW13]. The same trivially holds for $\mathsf{prefix}_2(j_1) = \left(\prod_{\ell=1}^{\log(T)/2}(1-r_\ell)\cdot j_{1,\ell}\right)$, since this equals 0 for all $j \in \{0,1\}^{\log(T)/2} \setminus \{\mathbf{1}\}$, and $\mathsf{prefix}_2(\mathbf{1})$ can be computed in $O(\log T)$ time.

Now consider $\mathsf{suffix}_2(j_2) = \widetilde{\mathsf{shift}}(r_2, j_2)$ and $\mathsf{prefix}_1(j_1) = \widetilde{\mathsf{shift}}(r_1, j_1)$. To simplify notation, for the remainder of this section let us consider shift to be defined over $2\log T$ inputs, the first $\log T$ of which are denoted simply by r (instead of r_1 or r_2) and the second by j (instead of j_1 or j_2). Per [STW23, Section 5.2], we utilize the following explicit expression for $\widetilde{\mathsf{shift}}$ from [STW23]:

$$\widetilde{\mathsf{shift}}(r,j) = h(r,j) + g(r,j), \tag{21}$$

where

$$h(r,j) = (1-j_1)r_1 \cdot \widetilde{\mathsf{eq}}(j_2,\ldots,j_{\log T}, r_2,\ldots,r_{\log T}), \tag{22}$$

and

$$g(r,j) = \sum_{k=1}^{\log(T)-1}\left(\prod_{i=1}^{k} j_i\cdot(1-r_i)\right)(1-j_{k+1})r_{k+1}\cdot\widetilde{\mathsf{eq}}(j_{k+2},\ldots,j_{\log T}, r_{k+2},\ldots,r_{\log T}). \tag{23}$$

Confirming that Equality (21) holds. Intuitively, $h(r,j)$ covers the case where the low-order bit j_1 of j is 0. In this case, if all entries of r and j are in $\{0,1\}$, then $\mathsf{int}(r) = \mathsf{int}(j)+1$ if and only if the low-order bit of r is 1 and all other bits of r and j are equal bit-by-bit.

Meanwhile, $g(r,j)$ handles the case where the low-order bit j_1 of j is 1. In this case, $\mathsf{int}(r) = \mathsf{int}(j)+1$ if and only for some $k \geq 1$:

- The first k bits of j are 1 the $(k+1)$'st bit of j is zero.
- The first k bits of r are 0, and the $(k+1)$'st bit of r is 1.
- r and j agree bit-by-bit on all higher order bits, i.e., $r_{k+2} = j_{k+2}, \ldots, r_{\log T} = j_{\log T}$.

We actually show something stronger than the claim that $\widetilde{\mathsf{shift}}(r,j)$ can be computed for all $j \in \{0,1\}^{\log T}$ in linear time and space: we show that all such evaluations can be *enumerated* in linear time and *polylogarithmic space*.

Enumerating evaluations of $g(r,j)$ by exploiting Expression (23). As noted earlier, it is known that the evaluations of $\widetilde{\mathsf{eq}}(j_2,\ldots,j_{\log T}, r_2,\ldots,r_{\log T})$ for all $(j_2,\ldots,j_{\log T}) \in \{0,1\}^{\log(T)-1}$ can be generated in lexicographic order in $O(\log T)$ space and $O(T)$ time (see [CFFZE24, Section 4.1] or [Rot24, Appendix A] for details). This ensures that all evaluations of $h(r,j)$ can be enumerated in lexicograhic order in $O(T)$ total time and $O(\log T)$ space. This algorithm amounts to a depth-first traversal of a binary tree whose leaves are labeled by $(j_2,\ldots,j_{\log T}) \in \{0,1\}^{\log(T)-1}$.

A similar statement holds for $g(r, j)$ evaluations, though the enumeration procedure is more complicated. To enumerate $g(r, j)$ as j ranges over strings $j = (j_1, \ldots, j_{\log T}) \in \{0,1\}^{\log T}$ with $j_1 = 1$ in lexicographic order, one can perform a depth-first traversal of a binary tree of depth $\log(T) - 2$ with leaves labeled by $\{0,1\}^{\log(T)-2}$. Each layer of the tree corresponds to a value k indexing a term of the sum on the right hand side of Equality (23). Tree leaves corresponding to $k = 1$ and the root corresponds to $k = \log(T) - 1$. We think of a depth-first traversal of this tree as performing a left-to-right traversal of each layer of the tree, with the traversals for different layers synchronized in the natural way (i.e., the traversal of layer k advances one node to the right only when all of the descendants of the current layer-k node have been traversed).

At each layer of the tree, the left-to-right traversal applies the algorithm that enumerates the values $\widetilde{\mathrm{eq}}(j_{k+2}, \ldots, j_{\log T}, r_{k+2}, \ldots, r_{\log T})$ as $(j_{k+2}, \ldots, j_{\log T})$ ranges over $\{0,1\}^{\log(T)-(k+1)}$ in lexicographic order.

In this manner, the root-to-leaf path for the current leaf $(j_3, \ldots, j_{\log T})$ under consideration identifies all $\log(T) - 2$ evaluations

$$\widetilde{\mathrm{eq}}(j_{k+1}, \ldots, j_{\log T}, r_{k+1}, \ldots, r_{\log T})$$

for $k \in \{1, \ldots, \log(T) - 1\}$. This enables the prover to evaluate the right hand side of Equation (23) for $j = (j_1 = 1, j_2, j_3, \ldots, j_{\log T})$.

The traversal at layer k costs $O(T/2^{k+1})$ field multiplications. Hence, the total number of multiplications to perform the traversals for all layers is $O(\sum_{k=1}^{\log(T-1)} T/2^k) = O(T)$. Given the values produced by these traversals, the prover performs $O(T)$ total field multiplications to iteratively produce the desired values $g(r, j)$ as j ranges over bitstrings in $\{0,1\}^{\log T}$ with $j_1 = 1$. A naive implementation that treats each layer's traversal independently uses $O(\log^2 T)$ space, but it is straightforward to reduce this to $O(\log T)$ space as the traversals at each layer are tightly related to each other. We omit details of this optimization for brevity.

B Overview of Lasso as Used in Jolt

In Jolt, the correctness of most operations is verified using a unified lookup table. This table is constructed to handle all RISC-V instructions, where each entry corresponds to the operation and its operands. For instance, a lookup table for a 32-bit XOR operation would have 2^{64} entries. The value at index (i, j)—where $i, j \in \{0,1\}^{32}$—is the XOR of i and j. The lookup tables for all operations are combined into a single, unified lookup table for RISC-V. During each cycle, the prover computes the relevant index based on the operands, and the correctness of the output is verified using Lasso.

Lasso is an indexed lookup argument that efficiently verifies lookups into large but decomposable tables. A table is deemed decomposable if its entries can be represented as a multilinear expression of values from smaller sub-tables (see Definition E.3 in Section 7). Jolt demonstrates that the lookup table for each RISC-V operation meets this criterion, enabling verification via Lasso. The lookup argument operates in two stages:

1. Sum-Check Protocol: Establishes that the outputs from the large lookup table correspond to a predefined multilinear expression involving outputs from the sub-tables.

2. Read-Only Memory Checking: Verifies the correctness of lookups into the sub-tables through a specialized version of the offline memory-checking argument.

The prover commits to the indexes of the sub-table lookups, ensuring the correctness of these indexes through an R1CS constraint. This constraint captures both the index computation and its usage in the lookup process.

The elements of the witness vectors required in both protocols can be enumerated in constant space and time per element (see Observation 3.5). In Section 3.1, we demonstrate how an honest prover, utilizing this witness generation algorithm, can execute the sum-check protocol while operating in logarithmic space.

The read-only memory checking in part 2 is verified using a specialized version of the offline memory-checking argument from Spice, where operations consist only of reads. In the next section, we describe the offline memory-checking argument, show that it reduces to a grand-product check. In Section D, we provide an honest prover algorithm for the grand-product check protocol that leverages the witness generation algorithm to operate in logarithmic space.

C Overview of Spice used in Jolt

Jolt leverages Spice to prove the correctness of reads and writes into memory. In this section, we outline how Spice reduces the problem of proving the consistency of memory operations to a grand-product check (see Appendix D). Let M represent the memory of size N, and let T denote the number of reads and writes performed on the memory. The memory is augmented with two additional columns, and each memory entry is represented as a tuple $(\mathsf{index}, \mathsf{value}, \mathsf{timestamp})$. Here:

- The index and value uniquely identify the content's position in memory.
- The timestamp records the temporal state of the memory. It is initialized to zero for all entries and is updated as reads and writes are performed.

The update process is described below. The prover is tasked with constructing the following sets: Reads, Writes, Memory_Init, and Memory_Fin. These four sets also consist of tuples of the same form as the memory entries. Algorithm 2 is employed to compute these sets.

Algorithm 2 accepts as input M, the initial state of the memory; write_val, the value written into memory at every time-step $t \in [0, T-1]$; and address, the index of the memory where the value is written at each time-step.[23] At Steps 1-5, the array Mem is computed, and at Step 6, Memory_Init is set to Mem. Steps 9-16 compute the sets Reads and Writes. At Step 11, the tuple currently at location address$[j]$ in Mem is pushed into the set Reads, and at Steps 12-13, the tuple in Mem at the same location is updated. The entries of the tuple corresponding to value and timestamp are updated with write_val$[j]$ and universal_timestamp, respectively. At Step 14, the updated tuple at location address$[j]$ in Mem is pushed into the set Writes, and at Step 15, the universal_timestamp is incremented by 1. The final state of Mem at the end of T rounds is set as Memory_Fin at Step 17.

Algorithm 2 Construction of Sets for Offline Memory Checking

$\mathcal{P}$**'s Input:** address $\in \mathbb{F}^T$, write_val $\in \mathbb{F}^T$, $M \in \mathbb{F}^N$
/* Here T is the execution length, and M is the memory of size N. */
Output: The four sets Reads, Writes, Memory_Init, and Memory_Fin.

1. **Initialize** Mem as a vector of size N, each entry being a tuple with three elements.
2. /* Each tuple corresponds to index, value, and timestamp, respectively. */
3. **for** $i \in [0, N-1]$ **do**
4. Set Mem$[i].0 \leftarrow i$, Mem$[i].1 \leftarrow M[i]$, and Mem$[i].2 \leftarrow 0$.
5. **end for**
6. **Initialize** universal_timestamp $\leftarrow 0$.
7. /* Compute the sets Memory_Init, Memory_Fin, Reads, and Writes. */
8. Set Memory_Init $\leftarrow$ Mem.
9. **Initialize** two empty sets: Reads and Writes.
10. **for** $j \in [0, T-1]$ **do**
11. Add Mem[address$[j]$] to Reads.
12. Update Mem[address$[j]$].1 $\leftarrow$ write_val$[j]$. /* Update the memory at the specified location with the new value. */

[23] If the operation is a read, then the value written is equal to the value read.

13. Update Mem[address[j]].2 $\leftarrow$ universal_timestamp. /* Update the timestamp at the specified location. */
14. Add the updated Mem[address[j]] to Writes.
15. Increment universal_timestamp $\leftarrow$ universal_timestamp $+ 1$.
16. **end for**
17. Set Memory_Fin $\leftarrow$ Mem.
18. **Output** Reads, Writes, Memory_Init, and Memory_Fin.

We now explain how the sets Reads, Writes, Memory_Init, and Memory_Fin computed by Algorithm 2 are used in the offline-memory checking argument. Let idx be a vector of length T such that $\text{idx}[j] = j$ for $j \in [0, T-1]$. The reads and writes into memory M are consistent if and only if the following conditions hold (see Section 3.1 in [SAGL18]):

1. The following two sets are equal: Reads $\cup$ Memory_Fin and Writes $\cup$ Memory_Init, where $\cup$ denotes the union.
2. For all $j \in [0, T-1]$, $\text{reads\_ts}[j] \leq \text{idx}[j]$, where $\text{reads\_ts}[j] = \text{Reads}[j].2$.

We state the additional witness vectors used in the arguments used to prove (1) and (2), and then present the main idea behind these arguments. Let reads_index, writes_index, reads_val, writes_val, and writes_ts be length-T vectors such that for $j \in [0, T-1]$:

$$\text{reads\_index}[j] = \text{Reads}[j].0, \quad \text{writes\_index}[j] = \text{Writes}[j].0, \quad \text{reads\_val}[j] = \text{Reads}[j].1,$$

$$\text{writes\_val}[j] = \text{Writes}[j].1, \quad \text{reads\_ts}[j] = \text{Reads}[j].2, \quad \text{writes\_ts}[j] = \text{Writes}[j].2.$$

Condition (2) above is proven using a lookup-table argument, where the operands are derived from reads_ts and idx. Proving condition (1) is reduced to a grand-product check, which we explain next. Once the required vectors are committed to, the verifier samples $\gamma, \tau \in \mathbb{F}$ uniformly at random. Let gpr_reads_vector be the vector of length $T + N$ whose elements are $(a + \gamma \cdot v + \gamma^2 \cdot t - \tau)$ for a tuple $(a, v, t) \in$ Reads $\cup$ Memory_Fin. Similarly, let gpr_writes_vector be the vector of length $T + N$ whose elements are $(a + \gamma \cdot v + \gamma^2 \cdot t - \tau)$ for a tuple $(a, v, t) \in$ Writes $\cup$ Memory_Init. Using a simple Schwartz-Zippel argument, it can be shown that, with high probability over the randomness of γ and τ, the sets Reads $\cup$ Memory_Fin and Writes $\cup$ Memory_Init are equal if and only if the two products, prod_reads and prod_writes, are equal:

$$\text{prod\_reads} = \prod_{j \in [0, T+N-1]} \text{gpr\_reads\_vector}[j],$$

$$\text{prod\_writes} = \prod_{j \in [0, T+N-1]} \text{gpr\_writes\_vector}[j].$$

The prover and verifier use the grand-product check argument (see Appendix D) to prove that prod_reads and prod_writes are computed correctly. The final step in the grand-product check argument requires the verifier to check the evaluations of the MLEs corresponding to gpr_reads_vector and gpr_writes_vector.

The evaluations of the MLEs corresponding to gpr_reads_vector and gpr_writes_vector can be obtained from the evaluations of the MLEs corresponding to reads_index, writes_index, reads_val, writes_val, reads_ts, writes_ts, and idx. Hence, the offline-memory checking argument reduces to the evaluations of the MLEs corresponding to these vectors. It is evident from the construction of these vectors that reads_index[j], writes_index[j], and address[j] are equal for all $j \in [0, T-1]$, and hence their corresponding MLEs are equal. Similarly, $\text{writes\_ts}[j] = \text{idx}[j] + 1$ for all $j \in [0, T-1]$. Hence, the prover at the beginning of the protocol commits to the MLEs corresponding to the following vectors address, reads_val, writes_val, and reads_ts, and

later uses the evaluation protocol of PCS to validate their evaluations. The evaluation of the MLE corresponding to idx can be computed in $O(\log T)$ time using its succinct expression.[24]

D Grand Product Check in Small Space

The grand product relation,

$$\mathcal{R} \quad = \quad \left\{ P \in \mathbb{F}, V \in \mathbb{F}^m \mid P = \prod_{i \in \{1,\ldots,m\}} v_i \text{ , where } v_i \text{ is the } i\text{-th component of } V \right\}$$

is a fundamental component in many SNARK constructions, where it is commonly used to verify that two vectors are permutations of each other. This section presents a protocol for verifying the grand product relation, featuring an honest prover algorithm that achieves quasi-linear time complexity while maintaining only logarithmic memory usage. To construct this protocol, we employ a lemma from Quarks [SL20], which allows for a sum-check-based protocol tailored to the grand product relation.

Lemma D.1 ([SL20]). $P = \prod_{x \in \{0,1\}^n} v(x)$ if and only if there exists a multilinear polynomial f in $n+1$ variables such that

1. $f(0,1,\ldots,1) = P$,
2. $f(x,0) = v(x)$ for all $x \in \{0,1\}^n$,
3. $f(x,1) = f(0,x) \cdot f(1,x)$ for all $x \in \{0,1\}^n$.

A protocol for $\mathcal{R}$ can be designed if the verifier is able to check Items (a), (b), and (c) from the lemma above. Item (a) is checked directly, while Item (b) requires the verifier to check that $f(\mathbf{u},0) = v(\mathbf{u})$ for a uniformly random $\mathbf{u} \in \mathbb{F}^n$. For Item (c), it is sufficient for the verifier to check that Equation 24 holds for a $\mathbf{u} \in \mathbb{F}^n$ sampled uniformly at random:

$$0 \quad = \quad \sum_{x \in \{0,1\}^n} \widetilde{\text{eq}}(\mathbf{u}, x) \cdot (f(x,1) - f(0,x) \cdot f(1,x)). \tag{24}$$

This check is reduced to evaluating the polynomials $\widetilde{\text{eq}}(\mathbf{u}, X)$, $f(X,1)$, $f(0,X)$, and $f(1,X)$ at a random point using the sum-check protocol, where $X = \{X_1, \ldots, X_n\}$ are the n variables.

Denote $g_1(X), g_2(X), g_3(X)$ as the n-variate polynomials $f(X,1)$, $f(0,X)$, and $f(1,X)$ respectively. For convenience, also denote $g_0(X)$ as the n-variate polynomial $\widetilde{\text{eq}}(\mathbf{u}, X)$. To leverage the small-space honest prover algorithm (see Algorithm 1 in Section 3.1.2) in the sum-check protocol, an algorithm is required that can iteratively compute the evaluations of $g_k(X)$ over $\{0,1\}^n$ for $k \in \{0,1,2,3\}$ using minimal space. In Jolt, however, the witness generation algorithm computes the values of v iteratively over $\{0,1\}^n$ (see Observation 3.5), and the values of $g_k(X)$ over $\{0,1\}^n$ must be derived from the evaluations of v. While storing the entire evaluation vector of v in memory is an option, this would require a prohibitive amount of space. In Section D.1, we describe a space-efficient, honest prover algorithm for the sum-check instance in Equation 24. This algorithm takes as input an oracle for v and queries it sequentially in the order: $v(\text{tobits}(0))$, $v(\text{tobits}(1))$, …, $v(\text{tobits}(2^n - 1))$. Assuming each entry of v can be computed in $O(1)$ time and space, as is the case in Jolt, the honest prover algorithm operates in $O(n \cdot 2^n)$ time and $O(n)$ space. This approach significantly reduces space complexity, making it feasible for small-space computations in protocols that utilize grand-product relations. Before proceeding to the next section, we recall that a binary string $x \in \{0,1\}^n$ is denoted as $(x_1, \ldots, x_n)$, where x_1 represents the least significant bit and x_n represents

[24] The MLE $\widetilde{\text{idx}}(x_0, \ldots, x_{\log T - 1})$ is equal to $\sum_{j \in [0, \log T - 1]} 2^j \cdot x_j$.

the most significant bit. Hence, when we refer to the least significant j bits (or trailing j bits) of x, we mean $x_1, \ldots, x_j$. Conversely, when referring to the most significant j bits (or leading j bits) of x, we mean $x_{n-j+1}, \ldots, x_n$.

D.1 Small-Space Prover Algorithm

Algorithm 3 provides a small-space honest prover algorithm for the sum-check instance presented in Equation 24. We first outline the core idea of the algorithm before delving into the details. Recall that in round $i \in \{1, \ldots, n\}$ of the sum-check protocol, the prover must compute the univariate polynomial $f_i(X_i)$ (see Section 3.1 for more details), after which the verifier responds with a random value r_i. The coefficients of f_i are obtained by interpolating its evaluations over a set $S = \{\alpha_0, \ldots, \alpha_3\}$. The evaluations of f_i are computed using techniques similar to those in Algorithm 1. However, in this case, we do not have oracle access to g_k. Instead, we compute the evaluations of g_k in $O(n)$ space by leveraging an intrinsic structure in f which we explain next.[25]

The polynomial f represents the computation of the product P using a depth-n binary-tree circuit composed solely of product gates. The leaves of this tree, designated as layer 0, are labeled by the evaluations of v over $\{0,1\}^n$, and each intermediate node computes the product of its two child nodes. This circuit contains $n+1$ layers, with layer j containing 2^{n-j} nodes for $j \in \{0, \ldots, n\}$, and the value of the node at layer n is equal to P. Let $\mathbf{1}^j$ denote a binary string of length j where all bits are equal to 1. It follows that the evaluations of f: $\left\{f(x, 0, \mathbf{1}^j)\right\}_{x \in \{0,1\}^{n-j}}$ correspond to the values of the nodes in layer j of the circuit for $j \in \{0, \ldots, n\}$. Moreover, for any $x \in \{0,1\}^{n-j}$, $f(x, 0, \mathbf{1}^j)$ equals the product of $f(0, x, 0, \mathbf{1}^{j-1})$ and $f(1, x, 0, \mathbf{1}^{j-1})$. We can express this relationship in terms of g_1, g_2, g_3 as follows: for every $j \in \{1, \ldots, n\}$, if $x' = (x, 0, \mathbf{1}^{j-1})$ for some $x \in \{0,1\}^{n-j}$, then $g_1(x') = g_2(x') \cdot g_3(x')$. Algorithm 3 leverages this structure to iteratively compute the evaluations of g_1, g_2, g_3 over $\{0,1\}^n$. Also, we set $f(\mathbf{1}^{n+1})$ to 0, and hence, $g_1(\mathbf{1}^n)$ and $g_3(\mathbf{1}^n)$ is equal to 0, whereas $g_2(\mathbf{1}^n)$ is equal to $g_1(0, \mathbf{1}^{n-1}) = P$. Below, we describe the different steps of the algorithm in detail.

Steps 3-39 correspond to a single round of the sum-check protocol. We describe the algorithm for round i. For convenience, let $G(X)$ denote the expression $g_0(X)(g_1(X) - g_2(X) \cdot g_3(X))$. As in Algorithm 1, at Step 3, accumulator is used to accumulate the summands corresponding to $f_i(\alpha_s)$, as stated below:

$$f_i(\alpha_s) = \sum_{m \in \{0, \ldots, 2^{n-i}-1\}} G(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m)) \tag{25}$$

The stack st, initialized in Step 4, stores intermediate values computed by the algorithm. Specifically, st holds $g_2(x')$ for some $x' \in \{0,1\}^n$ until $g_3(x')$ is computed by the algorithm. As will become evident in the following discussion, st never stores more than $n+1$ elements at a time. The three-dimensional array *g_evals*, initialized in Step 5, is used to compute $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ for $m \in \{0, \ldots, 2^{n-i}-1\}$ and $k, s \in \{0, 1, 2, 3\}$. This is similar to *witness_evals* in Algorithm 1, except that here we compute the values of $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ simultaneously for multiple values of m at a time. Specifically, *g_evals*$[j][k][s]$ accumulates terms corresponding to $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ based on the number of leading ones in m[26]. The algorithm partitions the values of $m \in \{0, \ldots, 2^{n-i}-1\}$ according to their number of leading ones, where *g_evals*$[j][k][s]$ stores the accumulated terms for all m having exactly j leading ones, for $j \in \{0, \ldots, n-i\}$. This approach is necessitated by the sequence in which Algorithm 3 computes evaluations of g_k over $\{0,1\}^n$. We emphasize here that Algorithm 3 does not compute the evaluations of g_k over $\{0,1\}^n$ in the lexicographic order. Steps 6-31 are explained next.

[25] Here f is as stated in Lemma D.1.

[26] Note that the rightmost bit of m is the most significant bit. The number of leading ones, in this sense, refers to the count of consecutive ones starting from the right in the binary representation of m.

At Step 6, the next n-bit string x is chosen according to the lexicographic order. The algorithm then evaluates $v(x)$ at Step 7 and pushes this value onto stack st. Steps 8-37 comprise a while loop that iterates exactly for the number of trailing ones in x. Specifically, the loop terminates at Step 11 if $x_{j+1} \neq 1$. When $x_{j+1} = 1$ at Step 11, the algorithm constructs the string $x^{(j)} = (x_{j+2}, \ldots, x_n, 0, \mathbf{1}^j)$ at Step 12, where the suffix consists of j ones. The algorithm then pops $g_3(x^{(j)})$ and $g_2(x^{(j)})$ from the stack to compute $g_1(x^{(j)})$. The correctness of Steps 13-15 follows from the following lemma, which characterizes the state of stack st throughout the algorithm's execution (proof in Appendix I).

Lemma D.2 (State of Stack During Algorithm Execution)**.** For any binary string $x \in \{0,1\}^n$ and index $j < n$, define $x^{(j)} = (x_{j+2}, \ldots, x_{n-1}, 0, \mathbf{1}^j) \in \{0,1\}^n$, where $\mathbf{1}^j$ denotes j consecutive ones. Let $J_x \subseteq \{1, \ldots, n\}$ denote the set of indices where $x_{j+1} = 1$. When Algorithm 3 processes input x at Step 6 in lexicographic order, the stack st at Step 8 contains:

- If $1 \in J_x$: The values $g_2(x^{(j)})$ for all $(j+1) \in J_x$ in decreasing order of j, followed by $g_3(x^{(0)})$,
- If $1 \notin J_x$: The values $g_2(x^{(j)})$ for all $(j+1) \in J_x$ in decreasing order of j.

The proof of Lemma D.2 relies on the following claim (proof in Appendix H).

Claim D.3. Let $x \in \{0,1\}^n$ and $j < n$. For a given x, define $x^{(j)}$ as described in the preceding lemma, and let $x^{(n)} = \mathbf{1}^n$. Consider the least significant bit of $x^{(j)}$. If this bit is 0, then $g_1(x^{(j)}) = g_2(x^{(j+1)})$. Otherwise, if this bit is 1, then $g_1(x^{(j)}) = g_3(x^{(j+1)})$.

By Lemma D.2, for any $x \in \{0,1\}^n$ with least significant bit 1, we have $1 \in J_x$. At Steps 13 and 14 with $j = 0$, the algorithm pops $g_3(x^{(0)})$ and $g_2(x^{(0)})$ respectively. At Step 15, $g_1(x^{(0)})$ is computed as the product of $g_3(x^{(0)})$ and $g_2(x^{(0)})$. From Claim D.3, if the second least significant bit x_2 is 1, then $g_1(x^{(0)})$ equals $g_3(x^{(1)})$; otherwise, it equals $g_2(x^{(1)})$. Therefore, by Lemma D.2, when $x_2 = 1$, at Steps 13 and 14 with $j = 1$, the algorithm pops $g_3(x^{(1)})$ and $g_2(x^{(1)})$ respectively. This argument extends inductively for each j where $x_{j+1} = 1$, proving the correctness of Steps 13-15.

As described previously, the algorithm accumulates terms corresponding to $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ in *g_evals* for multiple values of m simultaneously. At Step 17, t determines the appropriate index in *g_evals* based on the number of leading ones in $x^{(j)}$, capped at $n - i$. At Step 18, the value of the i-th bit of $x^{(j)}$ dictates which term is to be added to $g\_evals[t][k][s]$. This process is analogous to determining u_{odd} and u_{even} and appropriately combining the evaluations of the witness polynomials at u_{odd} and u_{even} in Algorithm 1 (see Steps 7, 9, and 11). In particular, observe that in Algorithm 1 for u_{even} the i-th bit is always zero, whereas for u_{odd} it is always one. At Step 23, the algorithm checks if 2^i terms have been added to $g\_evals[t][k][s]$. If this condition is met, it concludes that $g\_evals[t][k][s]$ is equal to $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ for $m = \text{value}(x^{(j)}_{n-i+1}, \ldots, x^{(j)}_n)$. The correctness of this step follows from the lexicographic ordering of strings, which ensures that all strings with t leading ones that contribute to the same m are processed consecutively. At Step 25, $\mathsf{g\_eval}[t][k][s]$ is reset to 0, for $k, s \in \{0,1,2,3\}$. Steps 27-31 handles the last-case corresponding to the string $\mathbf{1}^n$. Since we set $f(1^{n+1}) = g_1(\mathbf{1}^n) = g_3(\mathbf{1}^n) = 0$, we only have to set $g_2(\mathbf{1}^n)$ equal to $g_1(0, \mathbf{1}^{n-1})$. Steps 29-30 are similar to Steps 19, and 24 for the special case of $\mathbf{1}^n$. Finally, since the algorithm iterates over all $x \in \{0,1\}^n$ in the lexicographic order, it must be that at at Steps 24 and 30:

1. For $s \in \{0,1,2,3\}$, the value added to $\mathsf{accumulator}[s]$ equals $G(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ for some $m \in 2^{n-i}$.
2. For each $m \in 2^{n-i}$, $G(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ is added to $\mathsf{accumulator}[s]$ for $s \in \{0,1,2,3\}$.

Hence, at Step 38, $\mathsf{accumulator}[s]$ is equal to $f_i(\alpha_s)$ for $s \in \{0,1,2,3\}$. Finally, from Lemma D.2 it follows that st at any given time stores at most $n+1$ elements, and hence, Algorithm 3 is an honest prover algorithm

for the grand-product check operating in $O(n)$ space and $O(n \cdot 2^n)$ time. We make a note of this in Theorem D.4.

Theorem D.4. *Algorithm 3 operating in $O(n)$ space and $O(n \cdot 2^n)$ time is an honest prover algorithm for the sum-check instance corresponding to the grand-product check.*

Algorithm 3 Log-Space Honest Prover Algorithm for Grand-Product Check

$\mathcal{P}$'s input: Oracle $\mathcal{A}_v$ that take as input $i \in \{0, \ldots 2^n - 1]\}$ and returns $v(\text{tobits}(i))$.

/* We use A_k to denote the array that contains the Lagrange coefficients of g_k. Specifically, the j-th entry of A_k is equal to $g_k$tobits(j).*/
for $i \in \{1, \ldots, n-1\}$ **do**
  Initialise an array accumulator of size 4 and set all the entries to zero.
  Also, initialize an empty stack st.
  Initialise a three-dimensional array g_evals of size $(n+1-i) \times 4 \times 4$. /* g_evals$[j][k][s]$ is used to accumulate the summands corresponding to $g_k(r_1, \ldots, r_{i-1}, \alpha_s, \text{tobits}(m))$ for some $m \in \{0, 2^{n-i}\}$*/
  for $x \in \{0,1\}^n$ **do**
    Query $\mathcal{A}_v$ at x, and push the output of the oracle in st.
    Set $j = 0$
    while $j < n$ **do**
      Let x_{j+1} denote the $(j+1)$-th least significant bit of x, and $\mathbf{1}^j$ denote a binary string of length j with all bits being 1.
      if x_{j+1} is equal to 1 **then**
        Let $x^{(j)} = (x_{j+2}, \ldots, 0, \mathbf{1}^j)$ be a binary string in $\{0,1\}^n$.
        Set $g_3(x^{(j)}) = \text{pop}(\text{st})$,
        Set $g_2(x^{(j)}) = \text{pop}(\text{st})$,
        Set $g_1(x^{(j)}) = g_2(x^{(j)}) \cdot g_3(x^{(j)})$ and push it in st.
        Compute $g_0(x^{(j)})$ as $\widetilde{eq}(\mathbf{u}, x^{(j)})$ /*Here $\mathbf{u}$ is as in Equation 24 */
        Set $t = \min(j, n-i)$
        if $x_i^{(j)} == 1$ **then**
          For $k \in \{0,1,2,3\}$ add the following expression to g_eval$[t][k][s]$,

$$(\widetilde{eq}((r_1, \ldots, r_{i-1}), (x_1^{(j)}, \ldots, x_{i-1}^{(j)})) \cdot \alpha_s \cdot g_k(x^{(j)})$$

        else
          For $k \in \{0,1,2,3\}$ add the following expression to g_eval$[t][k][s]$,

$$(\widetilde{eq}((r_1, \ldots, r_{i-1}), (x_1^{(j)}, \ldots, x_{i-1}^{(j)})) \cdot (1 - \alpha_s) \cdot g_k(x^{(j)})$$

        end if
        if $x^{(j)}$ has last i bits equal to 1 **then**
          For $s \in \{0,1,2,3\}$ add the following to accumulator$[s]$:

$$\text{g\_eval}[t][0][s] \; (\; \text{g\_eval}[t][1][s] \; - \; \text{g\_eval}[t][2][s] \cdot \text{g\_eval}[t][3][s] \;)$$

          For $k, s \in \{0,1,2,3\}$ set g_eval$[t][k][s]$ to 0.
        end if
        if j is equal to $n-1$ **then**
          Set $g_2(\mathbf{1}^n) = g_1(x^{(j)})$, and compute $g_1(\mathbf{1}^n)$.
          For $k \in \{0,2\}$ add the following expression to g_eval$[t][k][s]$,

$$(\widetilde{eq}((r_1, \ldots, r_{i-1}), (x_1^{(j)}, \ldots, x_{i-1}^{(j)})) \cdot \alpha_s \cdot g_k(\mathbf{1}^n)$$

30. For $s \in \{0,1,2,3\}$ add the following to accumulator$[s]$:

$$\mathsf{g\_eval}[t][0][s]\ (\ \mathsf{g\_eval}[t][1][s]\ -\ \mathsf{g\_eval}[t][2][s] \cdot \mathsf{g\_eval}[t][3][s]\ )$$

31. **end if**
32. Increment j by 1.
33. **else**
34. Break;
35. **end if**
36. **end while**
37. **end for**
38. Set $f_i(\alpha_s) = \mathsf{accumulator}[s]$ for $s \in \{0,1,2,3\}$.
39. Interpolate the coefficients of $f_i(X_i)$ from its $\ell + 1$ evaluations $\{f_i(\alpha_s)\}_{s\in\{0,1,2,3\}}$, and send it to $\mathcal{V}$.
40. **end for**

E Small-Space Proving for Lasso

Jolt employs Lasso as its indexed lookup argument system. In this subsection, we show how to implement Lasso's prover algorithm in small space. We begin by formally defining indexed lookup arguments and the required table properties to apply Lasso in Jolt.

Definition E.1 (Indexed Lookup Argument)**.** Let $T \in \mathbb{F}^N$, and let $(\mathsf{commit}, \mathsf{open}, \mathsf{eval})$ be an extractable PCS for multilinear polynomials. Then, a lookup argument for T is a SNARK for the following relation:

$$\left\{(\mathsf{pp}, C_{\tilde{a}}, C_{\tilde{b}}) \mid \exists \mathbf{a}, \mathbf{b} \in \mathbb{F}^m \text{ such that } a_i = T[b_i], \forall i \in [0, n-1] \text{ and } \mathsf{open}(pp, C_{\tilde{a}}, \mathbf{a}) = 1, \mathsf{open}(pp, C_{\tilde{b}}, \mathbf{b}) = 1\right\}.$$

In this context, $\mathbf{a}$ represents the lookup vector and $\mathbf{b}$ the lookup indices vector. A key feature of Lasso is that it eliminates the need for cryptographic commitments to T when the table is either MLE-structured or decomposable.

Definition E.2 (MLE-Structured)**.** A table $T \in \mathbb{F}^N$ is MLE-structured if, for any $\mathbf{r} \in \mathbb{F}^{\log N}$, $\tilde{T}(\mathbf{r})$ can be evaluated using $O(\log N)$ field operations.

Definition E.3 (Decomposable Tables)**.** A table $T \in \mathbb{F}^N$ is c-decomposable if there exists a constant $k \in \mathbb{N}$, and $\alpha \leq k \cdot c$ tables $T_1, \ldots, T_\alpha \in \mathbb{F}^{N^{1/c}}$, and an α-variate polynomial G such that: for all $\mathbf{r} \in \mathbb{F}^{\log N}$,

$$\tilde{T}(\mathbf{r}) = G(\tilde{T_1}(\mathbf{r}_1), \ldots, \tilde{T}_k(\mathbf{r}_1), \tilde{T}_{k+1}(\mathbf{r}_2), \ldots, \tilde{T}_{2k}(\mathbf{r}_2), \ldots, \tilde{T}_\alpha(\mathbf{r}_c)).$$

Let $n = \log N$. Then $\mathbf{r}$ and $\mathbf{r}_1, \ldots, \mathbf{r}_c$ are related as follows: if $\mathbf{r} = (r_0, \ldots, r_{n-1}) \in \mathbb{F}^N$, then:

$$\mathbf{r}_1 = (r_0, \ldots, r_{n/c-1}), \quad \mathbf{r}_2 = (r_{n/c}, \ldots, r_{2n/c-1}), \quad \ldots, \quad \mathbf{r}_c = (r_{n-n/c}, \ldots, r_{n-1}).$$

The tables $T_1, \ldots, T_\alpha$ are called sub-tables.

We note that decomposability is a weaker requirement than being MLE-structured (see Section 2.3 in [STW24]). For c-decomposable tables, Lasso's prover commits to $3cm + c \cdot N^{1/c}$ elements, with the verifier performing $O(\log m)$ hashes and field operations, along with checking the evaluation proofs of a few $\log m$-variate multilinear polynomials at a random point using a PCS.

Space-Efficient Implementation

For a c-decomposable table T and vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}^m$ satisfying $a_i = T[b_i]$ for all $i \in [0, m-1]$, we can decompose the lookup operation as follows:

$$a_i = T[b_i] = G(T_1[b_{1,i}], \ldots, T_k[b_{1,i}], T_{k+1}[b_{2,i}], \ldots, T_{2k}[b_{2,i}], \ldots, T_\alpha[b_{c,i}]),$$

where $b_i \in \mathbb{F}$ and $\text{tobits}(b_i) \in \{0,1\}^{\log N}$. Here, $b_{1,i}$ represents the first $\log N/c$ bits, $b_{2,i}$ the next $\log N/c$ bits, and so on.

Let $\mathbf{a}_j \in \mathbb{F}^m$ be defined such that $a_{j,i} = T_j[b_{\lfloor j/k \rfloor, i}]$ for $j \in [1, \alpha]$. Then for all $i \in [0, N-1]$:

$$a_i = G(a_{1,i}, \ldots, a_{\alpha,i}).$$

The Lasso proof consists of two main components:

1. A sum-check protocol proving the above equation (detailed in Section E.1)
2. Indexed lookup arguments for sub-tables, implemented as read-only memory checking arguments, proving $a_{j,i} = T_j[b_{\lfloor j/k \rfloor, i}]$ for $j \in [1, \alpha]$ (detailed in Section E.2)

Theorem E.4. *Let T, $\mathbf{a}$, $\{\mathbf{a}_j\}_{j\in[1,\alpha]}$, $\mathbf{b}$, and $\{\mathbf{b}_j\}_{j\in[1,c]}$ be as described above. If there exists a witness generation algorithm that computes these vectors in their natural order with $O(1)$ space and time per element, then there exists an honest prover algorithm for the Lasso protocol that operates in $O(\log m + N^{1/c})$ space and $O(m \cdot \log m + N^{1/c})$ time.*

E.1 Sum-checks in Lasso

Assume $(\mathsf{commit}, \mathsf{open}, \mathsf{eval})$ is an extractable PCS for multilinear polynomials. In Lasso, the sum-check protocol is used to reduce the following relation to opening the MLEs corresponding to $\mathbf{a}$ and $\mathbf{a}_j \in \mathbb{F}^m$ for all $j \in [1, \alpha]$ at a random point, which is then verified using the PCS:

$$\Big\{(\mathsf{pp}, \left\{C_{\tilde{a}_j}\right\}_{j\in[1,\alpha]}, C_a) \mid \exists \mathbf{a} \in \mathbb{F}^m, \mathbf{a}_j \in \mathbb{F}^m, \forall j \in [1,\alpha] \text{ such that } a_i = G(a_{1,i}, \ldots, a_{\alpha,i})\ \forall i \in [0, m-1], \mathsf{open}(\mathsf{pp}, C_{\tilde{a}}, \mathbf{a}) = 1, \text{ and } \mathsf{open}(\mathsf{pp}, C_{\tilde{a}_j}, \mathbf{a}_j) = 1\ \forall j \in [1,\alpha]\Big\}.$$

The polynomial G in the context of Jolt is a sum of $O(1)$ products of variables.[27] Therefore, the results in Section 3.1 can be adapted to the sum-check instance above, yielding an honest prover algorithm that operates in $O(m)$ space and $O(m \cdot \log m)$ time. In Algorithm 1, calls to the oracles are replaced by the execution of the witness generation algorithm of Jolt (see Observation 3.5).

E.2 Grand-Product Checks in Lasso

The lookups into the sub-tables in Lasso are proved using a read-only memory checking argument adapted from Spice. Formally, we aim to provide a SNARK for the following relation for $j \in [1, \alpha]$:

$$\Big\{(\mathsf{pp}, C_{\tilde{a}_j}, C_{\tilde{b}_j}) \mid \exists \mathbf{a}_j, \mathbf{b}_j \in \mathbb{F}^m \text{ such that } a_{j,i} = T_j[b_j[i]]\ \forall i \in [0, N-1], \mathsf{open}(\mathsf{pp}, C_{\tilde{a}_j}, \mathbf{a}_j) = 1, \text{ and } \mathsf{open}(\mathsf{pp}, C_{\tilde{b}_j}, \mathbf{b}_j) = 1\Big\}.$$

We refer the reader to Section C for an overview of the offline-memory checking argument from Spice and how the above check is reduced to two grand-product checks. The following simplifications to the discussion in Section C are applicable since lookups are only a read-only memory checking argument:

[27] Concretely, the number of such terms is at most 4.

1. The vectors reads_val and writes_val in Section C correspond to $\mathbf{a}_j$.
2. The address vector committed to by the prover corresponds to $\mathbf{b}_j$.
3. In Algorithm 2, the universal_timestamp is not required. Furthermore, in Step 13, the current value of the counter in memory is incremented by one, instead of storing the value of universal_timestamp.
4. To prove the correctness of lookup values, it suffices for the prover to demonstrate that Reads $\cup$ Memory_Fin is equal to Writes $\cup$ Memory_Init. There is no need to additionally show that $\mathsf{reads\_ts}[j] \leq \mathsf{idx}[j]$ for all $j \in [0, T-1]$.

The Spice protocol reduces the check that Reads $\cup$ Memory_Fin equals Writes $\cup$ Memory_Init to two grand-product checks. Thus, using Algorithm 3 from Section D, we obtain an honest prover algorithm for proving the correctness of lookups into the sub-tables. As in Section E.1, in Algorithm 1, calls to the oracles are replaced by the execution of the witness generation algorithm of Jolt.

F Small-space Prover Algorithm for Spice used in Jolt

Jolt utilizes Spice to ensure the consistency of memory reads and writes performed in each RISC-V cycle. Appendix C provides an overview of Spice, explaining how it reduces the problem of verifying memory consistency to grand-product checks. In this section, we demonstrate that the prover algorithm for Spice can be executed in a small-space setting by leveraging the small-space prover algorithm for grand-product checks, as outlined in Algorithm 3 in Appendix D.

As discussed in Appendix C, the Spice prover establishes the equality of prod_reads and prod_writes, where prod_reads is the product of elements in the vector gpr_reads_vector, and prod_writes is the product of elements in gpr_writes_vector. To apply the small-space prover algorithm from Appendix D, we need to show that the elements of these vectors can be computed sequentially in $O(1)$ space and time.

The elements of gpr_reads_vector and gpr_writes_vector are derived from the tuples in Reads $\cup$ Memory_Fin and Writes $\cup$ Memory_Init, respectively. The sizes of Memory_Init and Memory_Fin is equal to the memory footprint of the RISC-V program and are stored in memory. In each RISC-V cycle, a new element from Reads and Writes is realized in $O(1)$ space and time (see Observation 3.5), allowing the corresponding elements of gpr_reads_vector and gpr_writes_vector to be computed in constant $O(1)$ space and time.

This demonstrates that gpr_reads_vector and gpr_writes_vector can be streamed sequentially, enabling an efficient small-space proof for the consistency of memory operations.

G Proof of Claim 3.2

We prove the claim by induction. For context, we refer to the discussion in Section 3.1 regarding the linear-time honest prover algorithm for the sum-check protocol. Recall that $A_{k,i}$ denotes the state of the arrays maintained by the honest prover at the beginning of round i for $k \in [1, \ell]$ and $i \in [1, n]$. Initially, $A_{k,1}$ contains the evaluations of g_k over $\{0,1\}^n$. For $i \in [1, n-1]$, the arrays are updated according to the relation:

$$A_{k,i+1}[m] = (1 - r_i) \cdot A_{k,i}[2m] + r_i \cdot A_{k,i}[2m+1] \tag{26}$$

Base case ($i = 1$): For $m \in [0, 2^{n-1} - 1]$, applying Equation 26:

$$\begin{aligned} A_{k,2}[m] &= (1 - r_1) \cdot A_{k,1}[2m] + r_1 \cdot A_{k,1}[2m+1] \\ &= \sum_{j \in \{0,1\}} \widetilde{eq}(r_1, j) \cdot A_{k,1}[2m + j] \end{aligned}$$

Inductive step: Assume the claim holds for some $i \in [2, n-1]$ and $\mathbf{r}_i = (r_1, \ldots, r_i)$, that is:

$$A_{k,i}[m] = \sum_{j \in \{0,1\}^i} \widetilde{eq}(\mathbf{r}_i, j) \cdot A_{k,1}[2^i \cdot m + j]$$

We prove it holds for $i+1$. Using the inductive hypothesis and Equation 26:

$$\begin{aligned} A_{k,i+1}[m] &= (1-r_i)\cdot \sum_{j\in\{0,1\}^i} \widetilde{eq}(\mathbf{r}_i, j) A_{k,1}[2^{i+1}\cdot m + j] \quad + \quad r_i \cdot \sum_{j\in\{0,1\}^i} \widetilde{eq}(\mathbf{r}_i, j) A_{k,1}[2^{i+1}\cdot m + 2^i + j] \\ &= \sum_{\substack{j\in\{0,1\}^{i+1} \\ \text{last bit of } j \text{ is } 0}} \widetilde{eq}(\mathbf{r}_i, j) A_{k,1}[2^{i+1}\cdot m + j] \quad + \quad \sum_{\substack{j\in\{0,1\}^{i+1} \\ \text{last bit of } j \text{ is } 1}} \widetilde{eq}(\mathbf{r}_i, j) A_{k,1}[2^{i+1}\cdot m + j] \\ &= \sum_{j\in\{0,1\}^{i+1}} \widetilde{eq}(\mathbf{r}_i, j) A_{k,1}[2^{i+1}\cdot m + j] \end{aligned}$$

This completes the proof.

H Proof of Claim D.3

Fix $x \in \{0,1\}^n$ and $0 \le j < n-1$. By the definitions of g_2 and g_3, we have

$$f(x^{(j+1)}, 0) = g_2(x^{(j+1)}) \quad \text{and} \quad f(x^{(j+1)}, 1) = g_3(x^{(j+1)}).$$

Note that $x^{(j+1)}$ has exactly $j+1$ leading ones, implying its most significant bit is 1. Furthermore, $x^{(j)}$ and $x^{(j+1)}$ are related as follows:

$$(x^{(j)}, 1) = (x_{j+2}, x^{(j+1)}).$$

Suppose $x_{j+2} = 0$. Then

$$g_2(x^{(j+1)}) = f(x_{j+2}, x^{(j+1)}) = f(x^{(j)}, 1) = g_1(x^{(j)}),$$

where the last equality follows from the definition of g_1. A similar argument holds when $x_{j+2} = 1$, yielding

$$g_3(x^{(j+1)}) = f(x_{j+2}, x^{(j+1)}) = f(x^{(j)}, 1) = g_1(x^{(j)}).$$

For the special case when $j = n-1$, we have $x^{(n-1)} = (0, \mathbf{1}^{n-1})$ and $x^{(n)} = \mathbf{1}^n$. By definition,

$$f(0, \mathbf{1}^n) = g_2(\mathbf{1}^n) \quad \text{and} \quad f(1, \mathbf{1}^n) = g_3(\mathbf{1}^n).$$

Using the same reasoning as above, we get

$$f(1, \mathbf{1}^{n-1}, 1) = g_1(x^{(n-1)}) = g_2(x^{(n)}).$$

I Proof of Lemma D.2

Proof. We prove this lemma by induction on the lexicographic ordering of strings in $\{0,1\}^n$.

Base case: Let $\mathbf{0}$ denote the all-zeros string. At Step 7, the algorithm queries the oracle to obtain $v(\mathbf{0})$. By definition of g_2, we have $g_2(\mathbf{0}) = v(\mathbf{0})$. Therefore, at Step 8, st contains only $g_2(\mathbf{0})$. Since $\mathbf{0}^{(0)} = \mathbf{0}$ and $J_{\mathbf{0}} = \emptyset$, this matches the lemma statement.

Inductive step: Assume the lemma holds for all strings lexicographically smaller than $x \in \{0,1\}^n$. Let y be the immediate predecessor of x in lexicographic order. We consider two cases:
Case 1: The least significant bit of y is 0. In this case:

1. The least significant bit of x must be 1 (by lexicographic ordering).

2. By the induction hypothesis, at Step 8, st contains $g_2(y^{(j)})$ for $(j+1) \in J_y$ in decreasing order.
3. Since $y_1 = 0$, the check at Step 11 fails for $j = 0$, exiting the while loop.
4. In the next iteration, $v(x)$ is pushed onto st.
5. Since $x_1 = 1$, we have $v(x) = f(0, x) = g_3(x^{(0)})$.
6. Note that $J_y = J_x \cup \{1\}$ and $y^{(j)} = x^{(j)}$ for all $j \in \{0, \ldots, n-1\}$.

Therefore, st satisfies the lemma's conditions for x.
Case 2: The least significant bit of y is 1. Let y have ℓ consecutive trailing ones ($\ell \in \{1, \ldots, n-1\}$). In particular, $y_1, \ldots, y_\ell$ are equal to ones and $y_{\ell+1}$ is equal to zero. Then:

1. By the induction hypothesis, st contains $g_2(y^{(j)})$ for $(j+1) \in J_y$ in decreasing order, followed by $g_3(y^{(0)})$.
2. By Claim D.3:
   - $g_1(y^{(j)}) = g_3(y^{(j+1)})$ for $j \in \{0, \ldots, \ell-2\}$,
   - $g_1(y^{(\ell-1)}) = g_2(y^{(\ell)})$.
3. The while loop (Steps 9-36) executes ℓ times:
   - First $\ell - 1$ iterations: For each $j \in \{0, \ldots, \ell-1\}$, $g_3(y^{(j)})$ and $g_2(y^{(j)})$ are popped, and $g_1(y^{(j)}) = g_3(y^{(j+1)})$ is pushed.
   - Final iteration: $g_3(y^{(\ell-1)})$ and $g_2(y^{(\ell-1)})$ are popped, and $g_1(y^{(\ell-1)}) = g_2(y^{(\ell)})$ is pushed.
4. After the while loop, st contains $g_2(y^{(j)})$ for $(j+1) \in J_y \setminus \{1, \ldots \ell\}$ in decreasing order, followed by $g_2(y^{(\ell)})$.
5. When x is processed, $v(x)$ is pushed onto st.
6. By lexicographic ordering, x has ℓ trailing zeros and $x_{\ell+1} = 1$.
7. Therefore, $J_x = (J_y \setminus \{1, \ldots, \ell\}) \cup \{\ell + 1\}$.
8. By definition of g_2, $v(x) = g_2(x^{(0)})$.

Thus, at Step 8, st contains $g_2(x^{(j)})$ for $(j+1) \in J_x$ in decreasing order satisfying the lemma's conditions. □