Proofs for Deep Thought: Accumulation for large memories and deterministic computations

Benedikt Bünz[1] and Jessica Chen[1]

New York University

Abstract. We construct two new accumulation schemes. The first one is for checking that ℓ read and write operations were performed correctly from a memory of size T. The prover time is entirely independent of T and only requires committing to 6ℓ field elements, which is an over 100X improvement over prior work. The second one is for deterministic computations. It does not require committing to the intermediate wires of the computation but only to the input and output. This is achieved by building an accumulation scheme for a modified version of the famous GKR protocol. We show that these schemes are highly compatible and that the accumulation for GKR can further reduce the cost of the memory-checking scheme. Using the BCLMS (Crypto 21) compiler, these protocols yield an efficient, incrementally verifiable computation (IVC) scheme that is particularly useful for machine computations with large memories and deterministic steps.

Keywords: Proof system · Accumulation Scheme · Incrementally Verifiable Computation.

1 Introduction

Imagine an untrusted prover P^{vm} that is performing an unbounded machine computation. P^{vm} wants to produce a convincing certificate that it ran the computation correctly. The certificate's size and the complexity of checking its correctness should ideally be independent of the length of the computation. P^{vm} should also be able to stop the machine at any point and output its state along with a certificate, so that anyone can check the computation up to that point and even continue it. This is the problem statement of *incrementally verifiable computation* (IVC) [Val08a] [1].

IVC enables P^{vm} to produce a proof π^{IVC} that convinces a verifier $\mathsf{V}^{\mathsf{IVC}}$ that some (possibly non-deterministic) machine computation was run correctly. IVC and its generalization to graphs, PCD [CT10], have many applications, which range from outsourcing computation to untrusted servers [BCTV14b], over distributed proving [CTV15], to verifiable delay functions [BBBF18]. IVC has also

[1] The literary application of IVC is the machine Deep Thought from the Hitchhikers Guide to the Galaxy. It computes the answer to the ultimate question of the universe and life over several thousand years. Given the nonsensical answer (42), it would have been helpful to be able to efficiently verify the correctness of the computation.

been deployed in practice, mainly in the context of succinct blockchain clients [KB20; BMRS20; CCDW20] and zk-Rollups [Tea22] which use IVC to show that an entire block of transactions is valid. Recently, a line of work has shown that IVC can be constructed more efficiently and from weaker assumptions using so-called accumulation[2] or folding schemes [BGH19; BCMS20; BCLMS21; BDFG21; KST22; BC23; EG23]. Accumulation-based IVC schemes can be built from simple protocols, including protocols where the prover simply sends the witness. The resulting IVC's recursive overhead only depends on algebraic degree of the verifier and the number of rounds of the underlying protocol but importantly not the communication or verifier complexity.

Despite this progress, significant barriers for IVC remain: Most machine computations require a memory. Proving the correctness of the computation requires proving that all the memory accesses were performed correctly. This can be handled by a separate prover $\mathsf{P}^{\mathsf{mem}}$, which performs the so-called *memory-proving*. Memory-proving allows $\mathsf{P}^{\mathsf{mem}}$ to convince a verifier $\mathsf{V}^{\mathsf{mem}}$ that all reads and write instructions were executed correctly. Given this, P^{vm} can focus on proving that a computation was correctly performed *given* a list of reads and writes. Prior approaches have the prover run either online or offline memory-checking techniques in order to prove that all the reads and writes where performed correctly. Online memory-checking requires the use of Merkle Trees so each read or write uses $\log T$ hashes. Offline memory-checking can be built using multi-set hash functions [CDvGS03; SAGL18] and requires a single hash to a group plus a linear scan of the memory at the end of the computation. Spice [SAGL18] used this approach, which resulted in around 1500 constraints[3] per read and write operations. On the contrary, Protostar recently showed that for static read-only memory, there can be a prover who only performs two group scalar multiplication per read instruction; however, their approach does not support write instructions [BC23]. This naturally brings us to our first research question:

Research Question 1: Memory-proving without hashing Can we perform efficient memory-proving for large memories without hashing while keeping the prover's work independent of the memory size T?

Another limitation of IVC is that even the most efficient constructions built from accumulation schemes require cryptographically committing to a witness per intermediate wire. Practically, this results in exactly one multi-scalar multiplication (MSM), which has the size of the circuit, even if the inputs (private and public) to the circuit are significantly smaller than the circuit. Another way to phrase this limitation is that the prover's witness is different from the computation's witness even for deterministic computations. Imagine an arithmetic circuit, consisting of addition and multiplication gates, that checks $x^{16} = y$ for some input x, y. The circuit's wires consist of 5 values (y, x, x^2, x^4 and x^8) even though the input is only 2 values. One way to resolve this is to have more expressive circuits with fewer but higher degree gates. Unfortunately, this approach

[2] We use accumulation to refer to split-accumulation as defined by [BCLMS21].

[3] In group-based proof systems the prover, generally, computes at least one multi-scalar multiplication that is as large the number of constraints.

quickly hits a limit as the degree grows exponentially when combining multiple gates into one, e.g. you would need one degree 16 gate to prove that $x^{16} = y$. An alternative approach relies on the fundamental work of GKR [GKR08]. GKR enables proving that a deterministic depth d computation was executed correctly while only requiring the verifier to have access to the input and output of the computation and not any of the intermediate values. Moreover, GKR only requires committing to and sending values that are linear in the depth of the circuit and the degree of the circuit gates, but only logarithmic in the width and thus the total size of the circuit. This is helpful for both the verifier's computation and, even more importantly, the prover's computation. Subsequent work [CMT12; XZZPS19; CBBZ23], further reduced the prover cost such that proving a computation is barely more expensive than running it. Unfortunately, GKR in its plain version, due to its many rounds of interaction, is not well-suited for constructing accumulation schemes. Nevertheless, GKR's existence motivates our second research question.

Research Question 2: Breaking the witness barrier Can we build an efficient accumulation-based IVC scheme that does not require committing to each intermediate circuit wire?

In this paper we answer both research questions positively and show a connection between them.

$O(\ell)$ memory-proving. Firstly, we built a memory-proving scheme that only requires the prover to do $O(\ell)$ computation where ℓ is the number of reads and writes done in each computation step, independent of the size of the memory T. This significantly improves on prior work, which either required $\mathsf{V}^{\mathsf{mem}}$ to do a linear-scan of the entire memory [BCGTV13] or relied on Merkle trees which require $\log T$ hashes per memory read or write. These methods are especially expensive in the context of IVC, where the prover proves recursively that $\mathsf{V}^{\mathsf{mem}}$ would have accepted the memory-proving proof. At it's core, we build an accumulation-based memory-proving scheme, which builds on a prior lookup argument first introduced in [Hab22]. Protostar already notes that this lookup argument is well suited for accumulation and enables checking that a set of witness values exists in a static table of values [BC23]. We extend the argument to support reading and writing to an entirely dynamic table. A key challenge we overcame is proving only $O(\ell)$ table values were changed in a table of size $T >> \ell$ while keeping the prover cost independent of T. The lookup argument is practically efficient with the prover only having to commit to 10ℓ field values for ℓ read and write operations. Using the Protostar compiler this accumulation scheme directly yields IVC with essentially the same computational overhead. One limitation of the scheme is that the prover needs to commit to 6ℓ large field elements, i.e. of λ bit size, even if the memory entries itself are small. Our second contribution for deterministic computation resolves this overhead.

IVC for deterministic computations. We build an accumulation scheme for the GKR protocol which retains GKR's efficiency properties, in particular the ability to prove deterministic layered computations without committing to the in-

termediate values. We build an accumulation scheme for GKR by utilizing the ProtoStar compiler, which takes any interactive special-sound public coin protocol with an algebraic verifier and turns it into an accumulation scheme. The accumulation scheme, in turn, can be used to construct IVC. The specific accumulation scheme we build is based on a variant of GKR that uses a bivariate sumcheck protocol instead of relying on multi-linear sumcheck, since bivariate sumcheck has the advantage of only requiring 3 prover messages. In the context of the ProtoStar compiler, this reduced number of rounds leads to a significantly reduced recursive overhead in IVC. While a bivariate sumcheck requires committing to messages of size $O(\sqrt{n})$, where n is the width of the circuit, this is generally not a bottleneck as the prover already needs to commit to the input of the circuit, which is of size $\Theta(n)$.

Improving memory-proving with GKR. We use the GKR protocol to improve our memory-proving protocol. In the memory-proving protocol, the prover needs to send 6 vectors consisting of inverses, with each inverse as large as the field. These inverses are used to compute sums of fractions that have the form $\sum_{i=1}^{\ell} \frac{1}{r+t_i}$ where r is a constant, and each t_i is a small table entry. We rely on ideas from Lasso, and logUp [STW23; PH23] to compute this sum using formal fractions and a $\log \ell$ depth GKR protocol. The protocol never requires formally computing and committing to the inverses. Thus, if we read/write ℓ c-bit values from memory, the number of group operations goes from $O(\lambda\ell)$ to $O(c\ell)$, i.e., the actual size of the data that is read/written. We briefly describe the resulting efficiency of our protocol in Table 1.

We also introduce several optimizations for our GKR-powered memory-proving, that help to further reduce the number of GKR rounds. GKR has many applications beyond improving our memory-proving protocol. In fact, it can reduce the commitment cost of *any* deterministic computation to just the inputs and outputs. We show the utility of this by briefly describing a GKR protocol for computing group scalar multiplications, the dominant cost inside the recursive circuit.

Table 1. Efficiency Table for our Memory-Proving Protocol. See Table 3 for an explanation of the columns and symbols, and more details.

	$\mathsf{P}_{\mathsf{acc}}$ Time	$\mathsf{V}_{\mathsf{acc}}$ Time
Plain	$(6\ell, \mathbb{T})$-MSM $+(9\ell, \mathbb{F})$-MSM	$3\mathbb{G}$
Using GKR	$(6\ell, \mathbb{T})$-MSM	$O(\log T)\mathbb{G}$

1.1 Related Work

IVC and Accumulation. Incrementally verifiable computation was first introduced by Valiant [Val08b] who showed that IVC can be built from SNARKs. The core idea is that for each computation step, the prover produces a SNARK

that certifies both the computation as well as the verification of SNARKs from the previous steps. We generally refer to the latter part as the *recursive circuit*. An important line of work [BCCT13; BCTV14a; COS20] that followed Valiant improved the practicality of IVC, generalized it to arbitrary graphs (called PCD) and improved it's theoretical foundations. However, in this entire line of work, IVC was still constructed from succinct arguments with sublinear verification. Halo [BGH19] first showed that IVC can be constructed from simpler assumptions. This has led to a very active line of work on accumulation [BCMS20; BCLMS21; BDFG21; KST22; BC23; EG23]. The idea of accumulation is that one can construct IVC by simply accumulating or batching the verification of non-interactive arguments. Roughly speaking, in every IVC step, the prover produces a new argument for the current step and proves that it has correctly accumulated the current argument with the existing accumulator. This accumulation step can be as simple as taking a random linear combination between two vector commitments. Only at the end of the IVC computation, a more expensive *decision* step is run to check the correctness of the commitment. Very recently, ProtoStar introduced a new recipe (the ProtoStar compiler) for constructing accumulation schemes and IVC [BC23]. The ProtoStar compiler operates in five steps

1. It takes as input *any* k-round special-sound interactive public coin protocol with L verification checks of maximum degree d, and prover messages of size n.
2. Communication in the protocol is compressed by committing to each message using a homomorphic vector commitment.
3. The protocol is made non-interactive using the Fiat-Shamir heuristic.
4. Let V be the verifier for the underlying protocol. Let π be the proof for that protocol such that for $V(\pi) = 0$ any accepting transcript. Given a proof π and an accumulator acc, the Protostar compiler outputs an accumulation scheme which reduces checking π and acc to checking acc' by computing $V(\mathsf{acc} + X \cdot \pi) = e(X)$ for some degree d polynomial $e(X)$ and checking this equality at a random point α by setting $\mathsf{acc}' = \mathsf{acc} + \alpha\pi$.
5. If the underlying protocol can prove NP-complete relations, e.g. circuits, then we can use the compiler of [BCLMS21] to construct an IVC scheme for any function F. The compiler proves that the accumulation verifier was run correctly alongside proving F.

The recursive circuit needs to implement the accumulation verifier, thus keeping the accumulation verifier's complexity low is an important design goal. Fortunately the accumulation verifier only depends on k and d but is entirely independent of n and L. The IVC prover's cost consists mainly of committing to the interactive prover's messages which are of size n. We will utilize the ProtoStar compiler by designing interactive, algebraic protocols with small k and d, while caring less about n and L. Keeping n, the size of the committed messages, low is an important secondary goal, as it directly impacts the prover's commitment cost.

Concurrent work[APPK24] also constructs an accumulation scheme for GKR. However, they utilize the multi-linear version of GKR and batch the polynomial evaluation, similar to [BCMS20].

Memory-proving and lookup arguments. Memory-checking [BEGKN91] enables an untrusted prover to convince a verifier that a set of read and write operations is consistent with a memory. Every read and write operation consists of an address a, a value v and a timestamp t. If a value v was written to a at time t, then any read at time $t' > t$ from a shall return a with the timestamp t, unless there was another write to a in the meantime. We refer to Appendix B of Jolt [AST23] for an excellent overview of memory-checking techniques but want to highlight two constructions and their limitations. One idea is to store the memory inside a Merkle Tree [BFRSBW13; BCTV14a]. For every read operation, the prover opens the Merkle Tree at the relevant address. For every write operation, the prover shows that the Merkle Tree was correctly updated. The verification for either step requires $\log T$ hashes where T is the size of the memory, and the prover's computational work is also $\log T$. However, when using this technique within IVC, the memory-checking verifier becomes part of the recursive circuit, and $\log(T)$ hashes per read and write operation is a significant overhead. The other common approach, dating back to [BEGKN91] and later refined[CDvGS03; DNRV09; SAGL18] relies on proving that the sets of reads and writes form a permutation. While each individual read or write can be as cheap as a few field operations, the scheme requires reading the entire memory at least once. Its complexity is therefore necessarily $\Theta(T)$, which is prohibitive when the size of the memory T greatly exceeds the number of reads/writes ℓ. This is commonly the case in IVC, where each computation step is constant size but the memory may be large. Our memory-checking scheme is specifically designed for IVC (in fact it only works in the IVC context) and has $O(\ell)$ complexity, independent of T. [DNRV09] uses a list of pointers to only scan the memory in positions that where touched. We achieve a similar result where the final decider's complexity only depends on the positions that were ever touched during memory-checking.

Recently, there has been increased attention to a related primitive called lookup arguments. Lookup arguments enable verified read operation in a static, possibly preprocessed memory. A recent line of work [ZBKMNS22; PK22; GK22; ZGKMR22; EFG22] has shown that in the preprocessing setting, one can achieve lookup arguments independent of the table size and quasi-linear in the number of read operations. Lasso[STW23] improves on these ideas by enabling a fully linear prover and independence of the table size for structured table. In the context of IVC, ProtoStar [BC23] gave a lookup argument based on [Hab22] that is independent of the table size (for arbitrary tables) and only requires two group scalar multiplications per read. Unfortunately, all these lookup arguments only work for static tables and read operations. We construct a memory-proving argument (which is more general than a lookup) that is still independent of the table size and has minimal overhead.

1.2 Technical Overview

Our construction heavily relies on the ProtoStar [BC23] compiler, which we describe in Theorem 1. It takes as input any special-sound interactive, k-round[4] protocol with an algebraic degree d verifier, and outputs an accumulation scheme. The transformation first converts the interactive argument into a NARK by committing to each round of prover messages using a homomorphic vector commitment and then using the Fiat-Shamir transform. The accumulation scheme then combines the current argument with an accumulator (which has the same form as the argument) by taking a random linear combination of the committed prover messages with the accumulator messages. It also computes a new verification equation by appropriately canceling out cross terms resulting from the accumulation. The accumulation verifier's cost is $k+2$ group operations and $d+3$ hashes and field operations. Note that the NARK's verifier cost and the size of the prover messages do not affect the accumulation verifier's complexity. The accumulation prover's main cost is committing to all the special-sound prover's messages. Using the BCLMS[BCLMS21] compiler an accumulation scheme for NP directly yields an IVC, where the prover's cost for computing the predicate is proportional to the accumulation prover and the recursive overhead consists of the accumulation verifier. Following this recipe we design special-sound, algebraic protocols for memory-proving and GKR.

From a read-only lookup argument to mem-update. The starting point of our construction is the *logUp* lookup argument [Hab22] which uses the fact that if and only if a set of values $\mathbf{w} = \{\mathbf{w}_1, \ldots, \mathbf{w}_\ell\}$ is in a table $\mathbf{t} = (\mathbf{t}_1, \ldots, \mathbf{t}_T)$ then $\sum_{i=1}^{\ell} \frac{1}{X+\mathbf{w}_i} = \sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X+\mathbf{t}_i}$, where $\boldsymbol{m}_i$ is the multiplicity of $\mathbf{t}_i$ in $\mathbf{w}$ for every $i \in [T]$ and X is a random variable. LogUp checks this equality at a random point. Protostar [BC23] observes that the prover message in this protocol, e.g. $\boldsymbol{m} = (\boldsymbol{m}_1, \ldots, \boldsymbol{m}_T)$, is sparse if $\ell << T$. This means in the context of IVC and the protostar compiler, the prover only needs to do $O(\ell)$ work. We would like to use the lookup argument in the context of memory-checking, where $\mathbf{t}$ corresponds to the memory and $\mathbf{w}$ to the set of read and write operations. However, the lookup argument only performs read operations and does not support write operations. Secondly, it is not immediately clear how to update $\mathbf{t}$, especially in a manner that does not require a linear scan. Our key observation is that even if the memory itself is not sparse, the difference (Δ) between an old memory state and a new memory state is sparse if $\ell << T$. Assume $RW = [(\boldsymbol{a}_i, \boldsymbol{r}_i, \mathbf{w}_i)]_{i=1}^{\ell}$ is a vector consisting of value $\boldsymbol{r}_i$ first read from $\boldsymbol{a}_i$ and value $\mathbf{w}_i$ last written to $\boldsymbol{a}_i$; such a vector can be constructed from the memory trace by pairing any initial read with a final write and vice versa. Our first attempt is to use a logUp style argument to show that Δ is consistent with RW, i.e. to show that $\sum_{i=1}^{\ell} \frac{1}{X+Y\cdot\boldsymbol{a}_i+(\mathbf{w}_i-\boldsymbol{r}_i)} = \sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X+Y*i+\Delta_i}$ for random variable X, Y. Note that this is an indexed lookup where we ensure that the values are also matched by address. This lookup ensures that each tuple in RW is consistent with Δ but is

[4] By round we refer to the number of prover messages.

not sufficient, as it does not ensure that Δ is 0 at the positions for which there was no read or write. This is important, since an adversarial prover could use non-zero values in Δ to change the memory state arbitrarily. We must, therefore, prove that Δ is truly the sparse representation of the changes reflected by the reads and writes, while still keeping the prover time independent of T. Our idea is to ensure that the ith fraction is 0 if and only if $\Delta_i = 0$ by setting the numerator to Δ_i instead of $\boldsymbol{m}_i$, i.e. check that $\sum_{i=1}^{\ell} \frac{(\mathbf{w}_i - \boldsymbol{r}_i)}{X + Y \cdot \boldsymbol{a}_i + (\mathbf{w}_i - \boldsymbol{r}_i)} = \sum_{i=1}^{T} \frac{\Delta_i}{X + Y \cdot i + \Delta_i}$. We show that this check is still secure and indeed results in a check with $O(\ell)$ prover complexity.

LogUp powered memory-proving. The mem-update argument described above can prove that Δ is consistent with a list of read and write operations. We can then use a homomorphic commitment to Δ to update our commitment to the memory $M' \leftarrow M + \Delta$. Unfortunately, the system is still limited. Checking that Δ is computed correctly can suffice in a system where all write operations happen synchronously at the end of the computation step. If we want to write and then read from a memory cell within a computation step then we would first need to update the memory. However, this requires an expensive homomorphic commitment operation executed by the verifier, i.e. as part of the recursive circuit. To resolve this we combine our memory-update argument with the classic permutation-based memory-checking idea [BEGKN91]. The key difference is that in the classic memory-checking, the entire memory is added to the read and write sets. We instead only add the cells that are accessed by some read or write operation and use mem-update to prove the correctness of the update. We lay out the precise protocol in Section 4.

Accumulation scheme for GKR. The lookup and memory-checking protocols have *almost* optimal parameters. They require committing[5] to just 10 vectors of length ℓ. However, 6 of these vectors consist of $\log |\mathbb{F}|$-bit elements even if the memory itself only consist of, say 32-bit entries. Using homomorphic commitments requires fields of size at least 256 so this is a factor 8 blowup. Removing this blowup motivates the second orthogonal but highly compatible contribution of this paper. To this end, we built an efficient accumulation scheme for the GKR protocol. This can prove low-depth deterministic computations while only committing to the computation's inputs and outputs but not the intermediate values. Note that GKR is a special-sound interactive protocol with an algebraic verifier, which means it can directly be compiled with the ProtoStar compiler to an accumulation scheme. Unfortunately, GKR has $O(k \cdot \log n)$ rounds where k is the depth of the circuit and $\log n$ its width. This results in an accumulation verifier with $k \cdot \log n$ group scalar multiplications. In the context of IVC, the accumulation verifier becomes part of the recursive circuit, and this is a significant overhead, especially when compared with other accumulation schemes

[5] Committing is by far the dominant prover cost in these systems. Committing to a message is between 100 and 1000 times as expensive as doing field operations on the same message. See `https://zka.lc/`.

which only have 1 to 3 group operations [KST22; KS23; BC23]. Our goal is therefore, to reduce the number of rounds of GKR while maintaining the attractive efficiency properties and the compatibility with the ProtoStar compiler.

In every round, GKR runs a multivariate sumcheck protocol, which has $\log n$ rounds. As a strawman, we can replace this multivariate sumcheck with a univariate one. This immediately reduces the number of GKR rounds from $k \cdot \log n$ to just k. Univariate sumcheck requires sending a quotient polynomial that is as large as the domain of the sumcheck, in our case $O(n)$. Committing to this polynomial would be at least as expensive as directly committing to the intermediate wires of the circuit, thus removing any benefit of using GKR. Fortunately, the idea of using a higher degree sumcheck can still help. Moving to a bivariate sumcheck reduces the communication to $O(\sqrt{n})$ while being only a 3-round protocol. The $O(\sqrt{n})$ commitment cost is, in most applications, dominated by the cost of committing to the input and output layers; even if not, we show that one can use a c-variate sumcheck to ensure that the sumchecks commitment cost is marginal. Using a bivariate sumcheck presents us with a couple of challenges. First, the verifier needs to evaluate a $O(\sqrt{n})$ degree polynomial, which is a $O(\sqrt{n})$ degree check if done naively. To resolve this we built a polynomial evaluation protocol, where with aid from the prover, the verification degree reduces to merely 3, independent of the degree of the polynomial.

Additionally, GKR batches polynomial evaluations, after each sumcheck, in order to only evaluate the next layer at a single point. In bivariate sumcheck, this would require computing a high-degree interpolation polynomial. We show that it is much simpler and more efficient to directly batch the resulting sumchecks. This observation is also applicable to multivariate sumchecks. We then construct a specific GKR for computing the sum of fractions, e.g. $\sum_{i=1}^{n} \frac{n_i}{d_i}$, similar to [PH23]. We also give specific optimizations for this instantiation, such as breaking up the circuit into multiple parts, while still maintaining the asymptotic properties. This optimization takes advantage of the circuit structure of sums of fractions, where the number of sums halves in every layer.

2 Preliminaries

Notation. For $n \in \mathbb{N}$, we use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. We denote λ as the security parameter and use $\mathbb{F}$ to denote a field of prime order p such that $\log(p) = \Omega(\lambda)$. For list of tuples $ltup = [(a_i, b_i, c_i, \ldots)]_{i=1}^{k}$ of arbitrary length k, we use $ltup.\boldsymbol{a}$ to denote the list $[a_i]_{i=1}^{k}$, and $ltup.(\boldsymbol{a}, \boldsymbol{b})$ to denote the list $[(a_i, b_i)]_{i=1}^{k}$. For function f, $\widetilde{f}$ denotes the bivariate extension of f.

2.1 Special-sound Protocols

We take the definition of special-soundness from [AFK22; BC23].

Definition 1 (Public-coin interactive proof). *An interactive proof $\Pi = (\mathsf{P}, \mathsf{V})$ for relation $\mathcal{R}$ is an interactive protocol between two probabilistic machines, a prover P, and a polynomial time verifier V. Both P and V take as*

public input a statement pi *and, additionally,* P *takes as private input a witness* $\mathbf{w} \in \mathcal{R}(\mathsf{pi})$ *. The verifier* V *outputs* 0 *if it accepts and a non-zero value otherwise. It's output is denoted by* $(\mathsf{P}(\mathbf{w}), \mathsf{V})(\mathsf{pi})$*. Accordingly, we say the corresponding transcript (i.e., the set of all messages exchanged in the protocol execution) is accepting or rejecting. The protocol is* public coin *if the verifier randomness is public. The verifier messages are referred to as challenges.* Π *is a* $(2k-1)$*-move protocol if there are* k *prover messages and* $k-1$ *verifier messages.*

Definition 2 (Tree of transcript). *Let* $\mu \in \mathbb{N}$ *and* $(a_1, \ldots, a_\mu) \in \mathbb{N}^\mu$*. An* $(a_1, \ldots, a_\mu)$*-tree of transcript for a* $(2\mu+1)$*-move public-coin interactive proof* Π *is a set of* $a_1 \dot{a}_2 \ldots a_\mu$ *accepting transcripts arranged in a tree of depth* μ *and arity* $a_1, \ldots, a_\mu$ *respectively. The nodes in the tree correspond to the prover messages and the edges to the verifier's challenges. Every internal node at depth* $i-1$ $(1 \le i \le \mu)$ *has* a_i *children with distinct challenges. Every transcript corresponds to one path from the root to a leaf node. We simply write the transcripts as an* (a^μ)*-tree of transcript when* $a = a_1 = a_2 = \cdots = a_\mu$*.*

Definition 3 (Special-sound Interactive Protocol). *Let* $\mu, N \in \mathbb{N}$ *and* $(a_1, \ldots, a_\mu) \in \mathbb{N}^\mu$*. A* $(2\mu+1)$*-move public-coin interactive proof* Π *for relation* $\mathcal{R}$ *where the verifier samples its challenges from a set of size* N *is* $(a_1, \ldots, a_\mu)$*-out-of-*N *special-sound if there exists a polynomial time algorithm that, on input* pi *and any* $(a_1, \ldots, a_\mu)$*-tree of transcript for* Π *outputs* $\mathbf{w} \in \mathcal{R}(\mathsf{pi})$*. We simply denote the protocol as an* a^μ*-out-of-*N *(or* a^μ*) special-sound protocol if* $a = a_1 = a_2 = \cdots = a_\mu$*.*

2.2 Commitment Scheme

Definition 4 (Commitment Scheme). *(Definition 6 from [BC23])* $\mathsf{cm} = (\mathsf{Setup}, \mathsf{Commit})$ *is a binding commitment scheme, consisting of two algorithms:*
$\mathsf{Setup}(\lambda) \to \mathsf{ck}$ *takes as input the security parameter and outputs a commitment key* ck*.*
$\mathsf{Commit}(\mathsf{ck}, \boldsymbol{m} \in \mathcal{M}) \to C \in \mathcal{C}$*, takes as input the commitment key* ck *and a message* $\boldsymbol{m}$ *in* $\mathcal{M}$ *and outputs a commitment* $C \in \mathcal{C}$*.*
The scheme is binding if for all polynomial-time randomized algorithms P^**:*

$$\Pr\left[\begin{array}{c}\mathsf{Commit}(\mathsf{ck}, \boldsymbol{m}) = \mathsf{Commit}(\mathsf{ck}, \boldsymbol{m}') \\ \wedge \\ \boldsymbol{m} \neq \boldsymbol{m}'\end{array}\middle|\begin{array}{c}\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda) \\ \boldsymbol{m}, \boldsymbol{m}' \leftarrow \mathsf{P}^*(\mathsf{ck})\end{array}\right] = \mathsf{negl}(\lambda)$$

Homomorphic commitment. We say the commitment is homomorphic if $(\mathcal{C}, +)$ *is an additive group of prime order* p*.*

2.3 Lookup Relation

Definition 5. *(Definition 12 of [BC23]) Given configuration* $\mathcal{C}_{LK} := (T, \ell, \boldsymbol{t})$ *where* ℓ *is the number of lookups and* $\boldsymbol{t} \in \mathbb{F}^T$ *is the lookup table, the relation* $\mathcal{R}_{LK}$ *is the set of tuples* $\mathbf{w} \in \mathbb{F}^\ell$ *such that* $\mathbf{w}_i \in \boldsymbol{t}$ *for all* $i \in [\ell]$*.*

Lemma 1. *(Lemma 5 of [Hab22])* [6] *Let $\mathbb{F}$ be a field of characteristic $p > \max(\ell, T)$. Given two sequences of field elements $[\mathbf{w}_i]_{i=1}^{\ell}$ and $[\boldsymbol{t}_i]_{i=1}^{T}$, we have $\{\mathbf{w}_i\} \subseteq \{\boldsymbol{t}_i\}$ as sets (with multiples of values removed) if and only if there exists a sequence $[\boldsymbol{m}_i]_{i=1}^{T}$ of field elements such that*

$$\sum_{i=1}^{\ell} \frac{1}{X + \mathbf{w}_i} = \sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X + \boldsymbol{t}_i}\,. \tag{1}$$

2.4 Vector-valued lookup

Definition 6. *(Definition 13 in [BC23]) Consider configuration $\mathcal{C}_{VLK} := (T, \ell, v \in \mathbb{N}, \boldsymbol{t})$ where ℓ is the number of lookups, and $\boldsymbol{t} \in (\mathbb{F}^v)^T$ is a lookup table in which the ith $(1 \leq i \leq T)$ entry is*

$$\boldsymbol{t}_i := (\boldsymbol{t}_{i,1}, \ldots, \boldsymbol{t}_{i,v}) \in \mathbb{F}^v\,.$$

A sequence of vectors $\mathbf{w} \in (\mathbb{F}^v)^\ell$ is in relation $\mathcal{R}_{VLK}$ if and only if for all $i \in [\ell]$,

$$\mathbf{w}_i := (\mathbf{w}_{i,1}, \ldots, \mathbf{w}_{i,v}) \in \boldsymbol{t}\,.$$

As noted in Section 3.4 of [Hab22], we can extend Lemma 1 and replace (1) with

$$\sum_{i=1}^{\ell} \frac{1}{X + w_i(Y)} = \sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X + t_i(Y)} \tag{2}$$

where the polynomials are defined as

$$w_i(Y) := \sum_{j=1}^{v} \mathbf{w}_{i,j} \cdot Y^{j-1}\,, \quad t_i(Y) := \sum_{j=1}^{v} \boldsymbol{t}_{i,j} \cdot Y^{j-1}\,,$$

which represent the witness vector $\mathbf{w}_i \in \mathbb{F}^v$ and the table vector $\boldsymbol{t}_i \in \mathbb{F}^v$.

2.5 Incremental Verifiable Computation (IVC)

In the following, we take an adapted from of the definition from [BCLMS21; KST22; BC23].

Definition 7 (IVC). *(Definition 7 from [BC23]) An incremental verifiable computation (IVC) scheme for function predicates expressed in a circuit-satisfiability relation $\mathcal{R}_{\mathsf{NP}}$ is a tuple of algorithms $\mathsf{IVC} = (\mathsf{P}_{\mathsf{IVC}}, \mathsf{V}_{\mathsf{IVC}})$ with the following syntax and properties:*

- $\mathsf{P}_{\mathsf{IVC}}(m, z_0, z_m, z_{m-1}, \mathbf{w}_{\mathsf{loc}}, \pi_{m-1}]) \rightarrow \pi_m$. *The IVC prover $\mathsf{P}_{\mathsf{IVC}}$ takes as input a program output z_m at step m, local data $\mathbf{w}_{\mathsf{loc}}$, initial input z_0, previous program output z_{m-1} and proof π_{m-1} and outputs a new IVC proof π_m.*

[6] This lookup argument is unofficially referred to as *logUp*.

– $\mathsf{V}_{\mathsf{IVC}}(m, z_0, z_m, \pi_m) \to b$. *The IVC verifier* $\mathsf{V}_{\mathsf{IVC}}$ *takes the initial input* z_0*, the output* z_m *at step* m*, and an IVC proof* π_m*, 'accepts' by outputting* $b = 0$ *and 'rejects' otherwise.*

The scheme IVC *has perfect adversarial completeness if for any function predicate* ϕ *expressible in* $\mathcal{R}_{\mathsf{NP}}$*, and any, possibly adversarially created,* $(m, z_0, z_m, , z_{m-1}, \mathbf{w}_{\mathsf{loc}}, \pi_{m-1})$ *such that*

$$\phi(z_0, z_m, z_{m-1}, \mathbf{w}_{\mathsf{loc}}) \wedge (\mathsf{V}_{\mathsf{IVC}}(m-1, z_0, z_{m-1}, \pi_{m-1}) = 0)$$

it holds that $\mathsf{V}_{\mathsf{IVC}}(m, z_0, z_m, \pi_m)$ *accepts for proof* $\pi_m \leftarrow \mathsf{P}_{\mathsf{IVC}}(m, z_0, z_{m-1}, z_m, \mathbf{w}_{\mathsf{loc}}, \pi_{m-1})$.

The scheme IVC *has knowledge soundness if for every expected polynomial-time adversary* P^**, there exists an expected polynomial-time extractor* $\mathsf{Ext}_{\mathsf{P}^*}$ *such that*

$$\Pr\left[\begin{array}{c} \mathsf{V}_{\mathsf{IVC}}(m, z_0, z, \pi_m) = 0 \wedge \\ ([\exists i \in [m], \neg\phi(z_0, z_i, z_{i-1}, \mathbf{w}_i)] \\ \vee z \neq z_m) \end{array} \middle| \begin{array}{c} [\phi, (m, z_0, z, \pi_m)] \leftarrow \mathsf{P}^* \\ [z_i, \mathbf{w}_i]_{i=1}^m \leftarrow \mathsf{Ext}_{\mathsf{P}^*} \end{array}\right] \leq \mathsf{negl}(\lambda)\,.$$

Here m *is a constant.*

Efficiency. The runtime of $\mathsf{P}_{\mathsf{IVC}}$ *and* $\mathsf{V}_{\mathsf{IVC}}$ *as well as the size of* π_{IVC} *only depend on* $|\phi|$ *and are independent on the number of iterations.*

Definition 8 (Fiat-Shamir Heuristic). *(Definition 9 from [BC23]) The Fiat-Shamir Heuristic, relative to a secure cryptographic hash function* H*, states that a random oracle NARK with negligible knowledge error yields a NARK that has negligible knowledge error in the standard (CRS) model if the random oracle is replaced with* H*.*

Theorem 1 (ProtoStar compiler). *(Theorem 3 from [BC23]) Let* $\mathbb{F}$ *be a finite field, such that* $|\mathbb{F}| \geq 2^\lambda$ *and* $\mathsf{cm} = (\mathsf{Setup}, \mathsf{Commit})$ *be a binding homomorphic commitment scheme for vectors in* $\mathbb{F}$*. Let* $\Pi_{\mathsf{sps}} = (\mathsf{P}_{\mathsf{sps}}, \mathsf{V}_{\mathsf{sps}})$ *be a special-sound protocol for an NP-complete relation* $\mathcal{R}_{\mathsf{NP}}$ *with the following properties:*

- *It's* $(2k-1)$ *move.*
- *It's* $(a_1, \ldots, a_{k-1})$*-out-of-*$|\mathbb{F}|$ *special-sound. Such that the knowledge error* $\kappa = 1 - \prod_{i=1}^{k-1}(1 - \frac{a_i}{|\mathbb{F}|}) = \mathsf{negl}(\lambda)$
- *The inputs are in* $\mathbb{F}^{\ell_{\mathsf{in}}}$
- *The verifier is degree* $d = \mathsf{poly}(\lambda)$ *with output in* $\mathbb{F}^\ell$

Then, under the Fiat-Shamir heuristic for a cryptographic hash function H *(Definition 8), there exist two IVC schemes* $\mathsf{IVC} = (\mathsf{P}_{\mathsf{IVC}}, \mathsf{V}_{\mathsf{IVC}})$ *and* $\mathsf{IVC}_{\mathsf{CV}} = (\mathsf{P}_{\mathsf{CV,IVC}}, \mathsf{V}_{\mathsf{CV,IVC}})$ *with predicates expressed in* $\mathcal{R}_{\mathsf{NP}}$ *with the efficiencies shown in Table 2.*

In Table 2, $|\mathbf{m}_i|$ *denotes the prover message length;* $|\mathbf{m}_i^*|$ *is the number of non-zero elements in* $\mathbf{m}_i$*;* $\mathbb{G}$ *for rows 1-3 is the total length of the messages committed using* Commit*.* $\mathbb{F}$ *are field operations.* H *denotes the total input length to a cryptographic hash, and* H_{in} *is the hash to the public input and accumulator instance.* $\mathsf{P}_{\mathsf{sps}}$ *(and* $\mathsf{V}_{\mathsf{sps}}$*) is the cost of running the prover (and the algebraic*

Table 2. Efficiency of IVC schemes compiled from sps protocol

$\mathsf{P}_{\mathsf{IVC}}$ native	$\mathsf{P}_{\mathsf{IVC}}$ recursive	$\mathsf{V}_{\mathsf{IVC}}$	$\lvert\pi_{\mathsf{IVC}}\rvert$
$\sum_{i=1}^{k} \lvert\mathbf{m}_i^*\rvert\mathbb{G}$ $\mathsf{P}_{\mathsf{sps}} + L'(\mathsf{V}_{\mathsf{sps}}, d+2)$	$k + 2\mathbb{G}$ $k + \ell_{\mathsf{in}} + d + 1\mathbb{F}$ $(k + d + O(1))\mathsf{H} + 1\mathsf{H}_{\mathsf{in}}$	$\sum_{i=1}^{k} \lvert\mathbf{m}_i\rvert\mathbb{G}$ $O(\ell) + \mathsf{V}_{\mathsf{sps}}$	$k + \ell_{\mathsf{in}} + 1\mathbb{F}$ $k + 2\mathbb{G}$ $\sum_{i=1}^{k} \lvert\mathbf{m}_i\rvert$

verifier) of the special-sound protocol, respectively. $L'(\mathsf{V}_{\mathsf{sps}}, d+2)$ *is the cost of computing the coefficients of the degree* $d+2$ *polynomial*

$$e(X) := \sum_{a=0}^{\sqrt{\ell}-1} \sum_{b=0}^{\sqrt{\ell}-1} (X \cdot \pi.\beta_a + \mathsf{acc}.\beta_a)(X \cdot \pi.\beta'_b + \mathsf{acc}.\beta'_b) \\ \sum_{j=0}^{d} (\mu + X)^{d-j} \cdot f_{j,a+b\sqrt{\ell}}^{\mathsf{V}_{\mathsf{sps}}}(\mathsf{acc} + X \cdot \pi)\,, \tag{3}$$

where all inputs are linear functions in a formal variable X[7]*, and* $f_{j,i}^{\mathsf{V}_{\mathsf{sps}}}$ *is the* i*th* $(0 \le i \le \ell - 1)$ *component of* $f_j^{\mathsf{V}_{\mathsf{sps}}}$*'s output. For the proof size,* $\mathbb{G}$ *and* $\mathbb{F}$ *are the number of commitments and field elements, respectively.*

3 Special Sound Subrelations for Read/Write Lookup

We introduce the three lookup subprotocols that will be combined later to build the Read/Write Memory-proving algorithm.

Handling Tuples. For simplicity, we describe the protocols as lookups and permutations on vectors of single values. However, when applied to memory-checking the entries could be tuples of addresses, values, and/or timestamps. Fortunately, this can be handled using a simple random linear combination, akin to the transformation from vector lookups to lookups (Lemma 6 of [BC23]). For sequence $\boldsymbol{b} = [\boldsymbol{b}_i]_{i=1}^n$ where each entry is a tuple of $k > 1$ element (i.e. $\boldsymbol{b}_i = (\boldsymbol{b}_{(i,j)})_{j=1}^k$ for every $i \in [k]$), $\boldsymbol{b}_i$ will implicitly denote the random linear combination of the elements, i.e. $\sum_{j=1}^k Y^{j-1}\boldsymbol{b}_{(i,j)}$, whenever it appears in a formula. For example,

$$\frac{1}{X + \boldsymbol{b}_i} = \frac{1}{X + \sum_{j=1}^k Y^{j-1}\boldsymbol{b}_{(i,j)}}\,.$$

This is a k-special-sound transformation, so a previously $(a_1, \ldots, a_\mu)$-special-sound protocol becomes $(k, a_1, \ldots, a_\mu)$-special sound after it.

[7] For example if $f_d = \prod_{i=1}^d (a_i + b_i \cdot X)$ then a naive algorithm takes $O(d^2)$ time but using FFTs it can be computed in time $O(d \log^2 d)$ [CBBZ22].

Achieving Perfect Completeness. The three protocols we introduce will not yet have perfect completeness since the prover will be sending over vectors of fractions of the form $\mathbf{h}_j = \frac{\boldsymbol{n}_j}{\boldsymbol{d}_j} \quad \forall j \in [|\mathbf{h}|]$, where the computation of the denominator d is dependent on values in the given witness or lookup table. If there exists any value in some entry of the witness or lookup table such that $d = 0$, then the prover message will be undefined. We can achieve perfect completeness by following the same strategy for achieving perfect completeness in Π_{LK} in [BC23], which is to have the verifier set $\mathbf{h}_j = 0$ in this case and changing the verification equation from $\mathbf{h}_j \cdot \boldsymbol{d}_j \stackrel{?}{=} \boldsymbol{n}_j$ to

$$\boldsymbol{d}_j \cdot (\mathbf{h}_j \cdot \boldsymbol{d}_j - \boldsymbol{n}_j) \stackrel{?}{=} 0$$

The new check ensures that either $\mathbf{h}_j = \frac{\boldsymbol{n}_j}{\boldsymbol{d}_j}$ or $\boldsymbol{d}_j = 0$. This increases the verifier degree in all of the three subprotocols to 3. Without these checks, the protocol has a negligible completeness error of $(\frac{\sum_i |\boldsymbol{h}_i|}{|\mathbb{F}|})$, where $\mathbf{h}_1, \mathbf{h}_2, \ldots$ are the vectors of fractions sent by the prover. This completeness error is negligible. However, IVC and thus accumulation from which IVC is constructed require the protocols to be perfectly complete [BCLMS21] because IVC is designed for distributed computations where the continuance of computation is important, even on adversarially generated inputs.

3.1 Checking Permutation Using Lookup Relation

Definition 9. *(Definition 10 from [BC23]) Two sequences of field elements $\mathbf{w} = [\mathbf{w}_i]_{i=1}^n$, $\mathbf{t} = [\mathbf{t}_i]_{i=1}^n$ are in $\mathcal{R}_{perm}$ if there exists permutation $\sigma : [n] \to [n]$ such that for all $i \in [n]$, $\mathbf{w}_i = \mathbf{t}_{\sigma(i)}$.*

Lemma 2. *Let $\mathbb{F}$ be a field of characteristic $p > \max(\ell, T)$. Given two sequences of field elements $\mathbf{w} = [\mathbf{w}_i]_{i=1}^{\ell}$ and $\mathbf{t} = [\boldsymbol{t}_i]_{i=1}^{T}$, we have $\mathbf{w}, \mathbf{t}$ are permutations of each other (i.e. $\mathbf{w}, \mathbf{t}$ are in $\mathcal{R}_{perm}$) if and only if $\ell = T$ and*

$$\sum_{i=1}^{\ell} \frac{1}{X + \mathbf{w}_i} = \sum_{i=1}^{T} \frac{1}{X + \mathbf{t}_i}\,. \tag{4}$$

Proof. Suppose $\ell = T$ and (4) holds. This implies there exists a sequence $[\boldsymbol{m}_i]_{i=1}^{T}$ of field elements where $\boldsymbol{m}_i = 1 \;\; \forall i \in [T]$ such that

$$\sum_{i=1}^{\ell} \frac{1}{X + \mathbf{w}_i} = \sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X + \mathbf{t}_i} \text{ and } \sum_{i=1}^{\ell} \frac{\boldsymbol{m}_i}{X + \mathbf{w}_i} = \sum_{i=1}^{T} \frac{1}{X + \mathbf{t}_i}$$

By Lemma 1, this means $\{\mathbf{w}_i\} \subseteq \{\mathbf{t}_i\}$ and $\{\mathbf{t}_i\} \subseteq \{\mathbf{w}_i\}$ as sets. Hence, it must be that $\mathbf{w} = \boldsymbol{t}$. The converse direction is trivial.

We can therefore describe a special-sound protocol Π_{perm} for $\mathcal{R}_{perm}$ by simply adding the check $\ell \stackrel{?}{=} T$ and removing the need to compute $\boldsymbol{m}$ from Π_{LK} for $\mathcal{R}_{\mathrm{LK}}$ in [BC23].

Special-sound protocol Π_{perm} for $\mathcal{R}_{perm}$		
Prover $\mathsf{P}(\mathbf{t} \in \mathbb{F}^T, \mathbf{w} \in \mathbb{F}^\ell)$		**Verifier** $\mathsf{V}(\mathbf{t} \in \mathbb{F}^T)$
	$\xrightarrow{\mathbf{w}}$	
Compute $\mathbf{h} \in \mathbb{F}^\ell, \mathbf{g} \in \mathbb{F}^T$	$\xleftarrow{x_1}$	$x_1 \leftarrow\$ \mathbb{F}$
$\mathbf{h}_j \leftarrow \frac{1}{x_1 + \mathbf{w}_j} \quad \forall j \in [\ell]$		
$\mathbf{g}_i \leftarrow \frac{1}{x_1 + \mathbf{t}_i} \quad \forall i \in [T]$	$\xrightarrow{\mathbf{h}, \mathbf{g}}$	$\ell \stackrel{?}{=} T$
		$\sum_{j=1}^{\ell} \mathbf{h}_j \stackrel{?}{=} \sum_{i=1}^{T} \mathbf{g}_i$
		$\mathbf{h}_j \cdot (x_1 + \mathbf{w}_j) \stackrel{?}{=} 1 \ \forall j \in [\ell]$
		$\mathbf{g}_i \cdot (x_1 + \mathbf{t}_i) \stackrel{?}{=} 1 \ \forall i \in [T]$

Complexity. Π_{perm} is a 3-move protocol (i.e. $k = 2$); the degree of the verifier is 2; the number of non-zero elements in the prover message is at most $2\ell + T$.

Special-Soundness. Just as Π_{LK} from [BC23], the perfect complete version of Π_{perm} is $2(\ell + T)$-special-sound, assuming each entry $\mathbf{w}_j, \mathbf{t}_i$ is a single value for all $j \in [\ell], i \in [T]$.

3.2 Indexed-Vector Lookup Relation

Definition 10. *(Indexed-Vector Lookup Relation) Given configuration $\mathcal{C}_{ivlk} := (T, \ell, \mathbf{t})$ where ℓ is the number of lookups and $\mathbf{t} \in \mathbb{F}^T$ is the lookup table, the triple $(\mathbf{t}, \mathbf{w} \in \mathbb{F}^\ell, \boldsymbol{b} \in \mathbb{F}^\ell)$ are in the relation $\mathcal{R}_{ivlk}$ if for all $j \in [\ell]$, $\boldsymbol{b}_j \in [T]$ and $\mathbf{w}_j = \mathbf{t}_{\boldsymbol{b}_j}$.*

Lemma 3 and 4 in the following are extensions on Lemma 4 and 5 from [Hab22], respectively.

Lemma 3. *Let $\mathbb{F}$ be an arbitrary field and $f_1, f_2 : \mathbb{F}^2 \to \mathbb{F}$ any functions. Then*

$$\sum_{\substack{z_1, z_2 \in \mathbb{F}^2, \\ X - z_1 \cdot Y - z_2 \neq 0}} \frac{f_1(z_1, z_2)}{X - z_1 \cdot Y - z_2} = \sum_{\substack{z_1, z_2 \in \mathbb{F}^2, \\ X - z_1 \cdot Y - z_2 \neq 0}} \frac{f_2(z_1, z_2)}{X - z_1 \cdot Y - z_2} \tag{5}$$

in the rational function field $\mathbb{F}(X, Y)$, if and only if $f_1(z_1, z_2) = f_2(z_1, z_2)$ for every $z_1, z_2 \in \mathbb{F}^2$.

Proof. Our proof strategy follows the proof of Lemma 4 in [Hab22].

Suppose that Equation (5) holds. Then

$$\sum_{\substack{z_1,z_2\in\mathbb{F}^2,\\ X-z_1\cdot Y-z_2\neq 0}} \frac{f_1(z_1,z_2)-f_2(z_1,z_2)}{X-z_1\cdot Y-z_2}=0$$

Fix Y at any arbitrary point $y\in\mathbb{F}$, we get $\sum_{\substack{z_1,z_2\in\mathbb{F}^2,\\ X-z_1\cdot y-z_2\neq 0}} \frac{f_1(z_1,z_2)-f_2(z_1,z_2)}{X-z_1\cdot y-z_2}$, and therefore have the polynomial

$$\begin{aligned} p(X,y) &= \prod_{q\in\mathbb{F}}(X-q)\cdot \sum_{\substack{z_1,z_2\in\mathbb{F}^2,\\ X-z_1\cdot y-z_2\neq 0}} \frac{f_1(z_1,z_2)-f_2(z_1,z_2)}{X-z_1\cdot y-z_2} \\ &= \sum_{z_1\in\mathbb{F},z_2\in\mathbb{F}} (f_1(z_1,z_2)-f_2(z_1,z_2))\cdot \prod_{q\in\mathbb{F}\setminus\{z_1\cdot y-z_2\}}(X-q)=0 \end{aligned}$$

In particular, for every pair $z_1\in\mathbb{F}, z_2\in\mathbb{F}$,

$$p(z_1\cdot y-z_2,y)=(f_1(z_1,z_2)-f_2(z_1,z_2))\cdot \prod_{q\in\mathbb{F}\setminus\{z_1\cdot y-z_2\}}(z_1\cdot y-z_2-q)=0$$

Since $\prod_{q\in\mathbb{F}\setminus\{z_1\cdot y-z_2\}}(z_1\cdot y-z_2-q)$ is not zero, it must be that $f_1(z_1,z_2)=f_2(z_1,z_2)$ for every pair $z_1,z_2\in\mathbb{F}^2$. The other direction is trivial.

Lemma 4. *Let $\mathbb{F}$ be a field of characteristic $p>\max\{\ell,T\}$. Given a sequence of field elements $\mathbf{w}\in\mathbb{F}^\ell, \boldsymbol{b}\in\mathbb{F}^\ell, \mathbf{t}\in\mathbb{F}^T$, we have $(T,\ell,\mathbf{t},\mathbf{w},\boldsymbol{b})\in\mathcal{R}_{ivlk}$ if and only if the following equation holds in the function field $F(X,Y)$*

$$\sum_{j=1}^{\ell}\frac{1}{X+Y\cdot\boldsymbol{b}_j+\mathbf{w}_j}=\sum_{i=1}^{T}\frac{\boldsymbol{m}_i}{X+Y\cdot i+\mathbf{t}_i} \tag{6}$$

where $\boldsymbol{m}=\{\boldsymbol{m}_i\}_{i=1}^T$ is the counter vector such that $\boldsymbol{m}_i$ is the count of $(i,\mathbf{t}_i)$ in $(\boldsymbol{b},\mathbf{w})$.

Proof. Our proof strategy follows the proof of Lemma 5 in [Hab22].

Suppose $(T,\ell,\mathbf{t},\mathbf{w},\boldsymbol{b})\in\mathcal{R}_{ivlk}$, then the equation is guaranteed to be true.

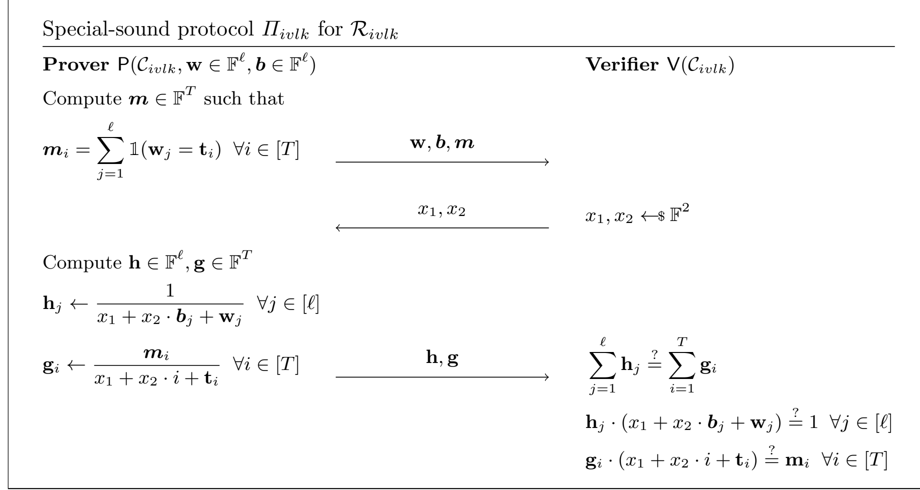
$$\sum_{i=1}^{T}\frac{\boldsymbol{m}_i}{X+Y\cdot i+\mathbf{t}_i}=\sum_{j=1}^{\ell}\frac{\boldsymbol{m}_{\boldsymbol{b}_j}}{X+Y\cdot\boldsymbol{b}_j+\mathbf{t}_{\boldsymbol{b}_j}}=\sum_{j=1}^{\ell}\frac{1}{X+Y\cdot\boldsymbol{b}_j+\mathbf{w}_j}$$

Conversely, suppose (6) holds. Collect fractions with the same denominator for the left side and re-expressing the right side we obtain,

$$\sum_{j=1}^{\ell}\frac{1}{X+Y\cdot\boldsymbol{b}_j+\mathbf{w}_j}=\sum_{z_1\in\mathbb{F},z_2\in\mathbb{F}}\frac{\mu_{\mathbf{w}}(z_1,z_2)}{X+Y\cdot z_1+z_2}=\sum_{i=1}^{T}\frac{\boldsymbol{m}_i}{X+Y\cdot i+\mathbf{t}_i}$$

where $\mu_{\mathbf{w}}(z_1, z_2)$ is the count of the tuple (z_1, z_2) in $(\boldsymbol{b}, \mathbf{w})$. By the uniqueness of bivariate fractional representations from Lemma 3, we have that for every non-zero $\frac{\mu_{\mathbf{w}}(z_1,z_2)}{X+Y\cdot z_1+z_2}$, there must exist a fraction in the $\sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X+Y\cdot i+\mathbf{t}_i}$ decomposition with equivalent numerator and denominator. This implies that for non-zero $\mu_{\mathbf{w}}(z_1, z_2) = \boldsymbol{m}_{z_1}$, and $\mathbf{t}_{z_1} = z_2$. Thus, for all $j \in [\ell]$ such that $\mu_{\mathbf{w}}(z_1, z_2) \neq 0$, $\mathbf{t}_{\boldsymbol{b}_j} = \mathbf{w}_j$ and $\boldsymbol{m}_{\boldsymbol{b}_j}$ is the count of $(\boldsymbol{b}_j, \mathbf{w}_j) = (i, \mathbf{t}_i)$ in $(\boldsymbol{b}, \mathbf{w})$.

We can therefore describe a special-sound protocol for the indexed-vector lookup relation.



Complexity. Π_{ivlk} is a 3-move protocol (i.e. $k = 2$); the degree of the verifier is 3; the number of non-zero elements in the prover message is at most 5ℓ.

Lemma 5. *Π_{ivlk} is $((\ell + T), 2(\ell + T))$-special-sound.*

Proof. We construct an extractor Ext that outputs $\mathbf{w}, \boldsymbol{b}$. We look at the $(\ell + T)$ transcripts that all have $\mathbf{w}, \boldsymbol{b}, \boldsymbol{m}$ as the first message but different $(x_1^{(p)})$ as the first challenge in the second message; then for each fixed $x_1^{(p)}$, we look at $2(\ell+T)$ transcripts that have $x_1^{(p)}$ as the first challenge but different $(x_2^{(q)}, \mathbf{h}^{(q)} \in \mathbb{F}^\ell, \mathbf{g}^{(q)} \in \mathbb{F}^T)$ as the rest of the transcript, totalling $2(\ell + T)^2$ transcripts.

By the pigeonhole principle, for each $p \in [\ell + T]$, there must exist a subset of $S \subseteq [2(\ell + T)]$ transcripts such that $|S| = \ell + T$ and $x_1^{(p)} + x_2^{(q)} \cdot \boldsymbol{b}_j + \mathbf{w}_j \neq 0$ for all $j \in [\ell]$ and $q \in S$, and $x_1^{(p)} + x_2^{(q)} \cdot i + \mathbf{t}_i \neq 0$ for all $i \in [T]$ and $q \in S$. For these transcripts, we have $\mathbf{h}_j = \frac{1}{x_1^{(p)}+x_2^{(q)}\cdot \boldsymbol{b}_j+\mathbf{w}_j}$ and $\mathbf{g}_i = \frac{\boldsymbol{m}_i}{x_1^{(p)}+x_2^{(q)}\cdot i+\mathbf{t}_i}$. Define the degree $\ell + T - 1$ polynomial

$$
\begin{aligned}
f(X, Y) = &\prod_{p=1}^{\ell} (X + Y \cdot \boldsymbol{b}_p + \mathbf{w}_p) \cdot \prod_{q=1}^{T} (X + Y \cdot 1 + \mathbf{t}_q) \\
&\cdot \left(\sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} - \sum_{i=1}^{T} \frac{\mathbf{t}_i}{X + Y \cdot i + \mathbf{t}_i} \right)
\end{aligned}
$$

If $f(X, Y)$ is the zero polynomial then $\sum_{j=1}^{\ell} \frac{1}{X+Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} = \sum_{i=1}^{T} \frac{\boldsymbol{m}_i}{X+Y \cdot i + \mathbf{t}_i}$ and by Lemma 6 $(\mathcal{C}_{ivlk}; \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{memup}$. Since we have $(\ell + T)$ points $(x_1^{(p)}, x_2^{(q)})$ at which $f(x_1^{(p)}, x_2^{(q)}) = 0$, we get $f = 0$ and thus that the extracted witness $(\mathbf{w}, \boldsymbol{b})$ is valid.

3.3 Mem-Update Relation

Definition 11 (Mem-Update Relation). *Given configuration* $\mathcal{C}_{memup} := (T, \ell, \Delta)$ *where* ℓ *is the number of lookups and* $\Delta \in \mathbb{F}^T$ *is the update table, the triple* $(\mathbf{w} \in \mathbb{F}^\ell, \boldsymbol{b} \in \mathbb{F}^\ell, \Delta)$ *are in the relation* $\mathcal{R}_{memup}$ *if for all* $j \in [\ell]$*, if* $\mathbf{w}_j \neq 0$ *then* $\mathbf{w}_j = \Delta_{\boldsymbol{b}_j}$*, and for all* $i \in [T]$*, if* $\Delta_i \neq 0$ *then there exists* $j \in [\ell]$ *such that* $\boldsymbol{b}_j = i$ *and* $\Delta_i = \mathbf{w}_j$*.*

Lemma 6. *Let* $\mathbb{F}$ *be a field of characteristic* $p > \max\{\ell, T\}$*. Given the sequences of field elements* $\mathbf{w} \in \mathbb{F}^\ell, \boldsymbol{b} \in \mathbb{F}^\ell, \Delta \in \mathbb{F}^T$*, we have* $(T, \ell, \Delta, \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{memup}$ *if and only if the following equation holds in the function field* $F(X, Y)$

$$\sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} = \sum_{i=1}^{T} \frac{\Delta_i}{X + Y \cdot i + \Delta_i} \tag{7}$$

Proof. Suppose $(T, \ell, \Delta, \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{memup}$, then the equation is guaranteed to be true.

$$\sum_{i=1}^{T} \frac{\Delta_i}{X + Y \cdot i + \Delta_i} = \sum_{j=1}^{\ell} \frac{\Delta_{\boldsymbol{b}_j}}{X + Y \cdot \boldsymbol{b}_j + \Delta_{\boldsymbol{b}_j}} = \sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j}$$

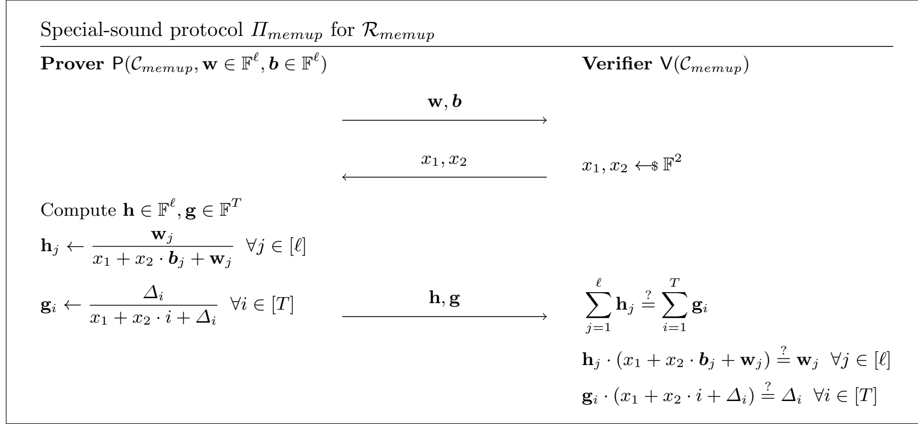
Conversely, suppose (7) holds. Collect fractions with the same denominator for the left side and re-expressing the right side we obtain,

$$\sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} = \sum_{z_1 \in \mathbb{F}, z_2 \in \mathbb{F}} \frac{z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2)}{X + Y \cdot z_1 + z_2} = \sum_{i=1}^{T} \frac{\Delta_i}{X + Y \cdot i + \Delta_i}$$

where $\mu_{\mathbf{w}}(z_1, z_2)$ is the count of the tuple (z_1, z_2) in $(\boldsymbol{b}, \mathbf{w})$. By the uniqueness of bivariate fractional representations from Lemma 3, we have that for every non-zero $\frac{z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2)}{X+Y \cdot z_1 + z_2}$, there must exist a fraction in the $\sum_{i=1}^{T} \frac{\Delta_i}{X+Y \cdot i + \Delta_i}$ decomposition with equivalent numerator and denominator. This implies that for non-zero $z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2) = \Delta_{z_1} = z_2$, so $\mu_w(z_1, z_2) = 1$. Thus, for all $j \in [\ell]$ such that $\mathbf{w}_j \neq 0$, $\Delta_{\boldsymbol{b}_j} = \mathbf{w}_j$.

For every $\frac{\Delta_i}{X+Y \cdot i + \Delta_i}$, there must exist a fraction in the $\sum_{z_1 \in \mathbb{F}, z_2 \in \mathbb{F}} \frac{z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2)}{X+Y \cdot z_1 + z_2}$ decomposition with the equivalent numerator and denominator, which implies $\Delta_i = z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2) = z_2$ for every Δ_i. Therefore if $\Delta_i \neq 0$ then $\mu_{\mathbf{w}}(z_1, z_2) \neq 0$, and thus there exists $j \in [\ell]$ such that $\boldsymbol{b}_j = i$ and $\Delta_i = \mathbf{w}_j$.

We describe a $((\ell + T), 2(\ell + T))$-special-sound protocol for the mem-update relation.



Complexity. Π_{memup} is a 3-move protocol (i.e. $k = 2$); the degree of the verifier is 3; the number of non-zero elements in the prover message is at most 4ℓ.

Lemma 7. *Π_{memup} is $((\ell + T), 2(\ell + T))$-special-sound, assuming each entry $\mathbf{w}_j, \Delta_i$ for all $j \in [\ell], i \in [T]$ is a single value.*

Proof. We construct an extractor Ext that outputs $\mathbf{w}, \boldsymbol{b}$. We look at the $(\ell + T)$ transcripts that all have $\mathbf{w}, \boldsymbol{b}$ as the first message but different $(x_1^{(p)})$ as the first challenge in the second message; then for each fixed $x_1^{(p)}$, we look at $2(\ell + T)$ transcripts that have $x_1^{(p)}$ as the first challenge but different $(x_2^{(q)}, \mathbf{h}^{(q)} \in \mathbb{F}^\ell, \mathbf{g}^{(q)} \in \mathbb{F}^T)$ as the rest of the transcript, totalling $2(\ell + T)^2$ transcripts.

By the pigeonhole principle, for each $p \in [\ell + T]$, there must exist a subset of $S \subseteq [2(\ell + T)]$ transcripts such that $|S| = \ell + T$ and $x_1^{(p)} + x_2^{(q)} \cdot \boldsymbol{b}_j + \mathbf{w}_j \neq 0$ for all $j \in [\ell]$,and $q \in S$, and $x_1^{(p)} + x_2^{(q)} \cdot i + \Delta_i \neq 0$ for all $i \in [T]$ and $q \in S$. For these transcripts, we have $\mathbf{h}_j = \frac{\mathbf{w}_j}{x_1^{(p)} + x_2^{(q)} \cdot \boldsymbol{b}_j + \mathbf{w}_j}$ and $\mathbf{g}_i = \frac{\Delta_i}{x_1^{(p)} + x_2^{(q)} \cdot i + \Delta_i}$. Define the degree $\ell + T - 1$ polynomial

$$f(X, Y) = \prod_{p=1}^{\ell} (X + Y \cdot \boldsymbol{b}_p + \mathbf{w}_p) \cdot \prod_{q=1}^{T} (X + Y \cdot q + \Delta_q) \cdot \left(\sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} - \sum_{i=1}^{T} \frac{\Delta_i}{X + Y \cdot i + \Delta_i} \right)$$

If $f(X, Y)$ is the zero polynomial then $\sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} = \sum_{i=1}^{T} \frac{\Delta_i}{X + Y \cdot i + \Delta_i}$ and by Lemma 6 $(\mathcal{C}_{memup}; \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{memup}$. Since we have $(\ell + T)$ points $(x_1^{(p)}, x_2^{(q)})$ at which $f(x_1^{(p)}, x_2^{(q)}) = 0$, we get $f = 0$ and thus that the extracted witness $(\mathbf{w}, \boldsymbol{b})$ is valid.

Efficiency in Accumulation. We refer to Table 3 for an overview over the efficiency of the protocol. Importantly the prover time is entirely independent of

T. The protocol can also be combined with our GKR protocol as layed out in Section 6. This reduces the prover time by eliminating the multi-scalar multiplication with full field elements. It is, thus, a useful option when the size of the table elements is significantly smaller than the field, e.g. 32-bit elements vs a 256-bit field.

Table 3. Efficiency Table for Accumulating Π_{mu}. We only list the dominant efficiency factors, ignoring the cost for $\mathsf{P}_{\mathsf{acc}}$ to compute the vectors. Column 2 refers to the total size of the prover messages. Here $\mathbb{T}$ is the set of small elements that are stored in the table, whereas $\mathbb{F}$ refers to full field elements. Column 3 is the verifier degree. Column 5 is the number of prover messages. Note that the number of messages in the GKR case can be further reduced with the optimizations mention in Section 6. Column 6 is the dominant factor in the prover time. An (a, B)-MSM refers to a multi-scalar multiplication of a scalars that are each within the set B. The MSM scales roughly linear in $|\log B|$. Column 7 is the number of group scalar multiplications the accumulation verifier performs.

	P Time	$\lvert\mathsf{P}$ Msg$\rvert$	deg(V)	# P Msgs	$\mathsf{P}_{\mathsf{acc}}$ Time	$\mathsf{V}_{\mathsf{acc}}$ Time
Plain	$O(\ell)$	$2\ell\,\mathbb{F} + 2\ell\mathbb{T}$	5	2	$(2\ell, \mathbb{T})$-MSM + $(2\ell, \mathbb{F})$-MSM	$4\mathbb{G}$
With GKR	$O(\ell)$	$2\ell\,\mathbb{F} + 2\ell\,\mathbb{T}$	7	$(c+1)\log T$	$(2\ell, \mathbb{T})$-MSM + $O(\ell \log \ell)\mathbb{F}$	$(c+1)\log T\mathbb{G}$

4 The Lookup-Powered Memory-Proving Algorithm

4.1 Offline Memory-Checking

In our memory-proving algorithm, we assume that the list of "reads" and the list of "writes" we are given were constructed according to the offline memory-checking process described in [BEGKN91; CDvGS03; SAGL18]. More importantly, our algorithm makes specific use of the "initial reads" and "final writes" in the memory-checking process. We explicitly define them here below.

The offline checker locally intializes two lists, RS and WS, to empty list. As in [BEGKN91], we assume both a value and a discrete timestamp of when the value was written are stored at each memory address. The local timestamp t^* is only incremented when some write operation takes place on the data structure.

When a read operation from address a is performed, and the memory responds with a value-timestamp pair (v, t), the checker updates its local state as follows:

1 : checks that $t^* > t$
2 : append (a, v, t) to RS
3 : stores (v, t^*) at the memory
4 : append (a, v, t^*) to WS
5 : $t^* \leftarrow t^* + 1$

When a write operation of value v' to address a occurs, the checker first reads from address a. Suppose the memory responds with a value-timestamp pair (v, t). The checker updates its local state as follows:

$$
\begin{aligned}
&1: \quad \text{checks that } t^* > t \\
&2: \quad \text{append } (a, v, t) \text{ to } \mathsf{RS} \\
&3: \quad \text{stores } (v', t^*) \text{ at the memory} \\
&4: \quad \text{append } (a, v', t^*) \text{ to } \mathsf{WS} \\
&5: \quad t^* \leftarrow t^* + 1
\end{aligned}
$$

Naturally, the entries in RS and WS would be sorted in increasing order of t after performing all the read and write operations. Then we compute the "initial reads" R and "final writes" W as follows:

$$
\begin{aligned}
&R, W, A_R, A_W \leftarrow \{\} \\
&\textbf{for } (a, v, t) \in \mathsf{RS} \textbf{ do} \\
&\quad \textbf{if } a \notin A_R \textbf{ then do} \\
&\quad\quad \text{append } (a, v, t) \text{ to } R \\
&\quad\quad A_R \leftarrow A_R \cup \{a\} \\
&\textbf{for } (a, v, t) \in \mathsf{WS}.rev \textbf{ do} \\
&\quad \textbf{if } a \notin A_W \textbf{ then do} \\
&\quad\quad \text{append } (a, v, t) \text{ to } W \\
&\quad\quad A_W \leftarrow A_W \cup \{a\}
\end{aligned}
$$

Finally, we sort R, W by addresses in the same order, and return $\mathsf{Rd} := \mathsf{RS}||W$ and $\mathsf{Wr} := \mathsf{WS}||R$.

Lemma 8. *(Contrapositive of Lemma 1 from [BEGKN91]) If* Rd *and* Wr *are permutations of each other, then for every* $(a, v, t) \in \mathsf{RS}$*, a read operation read value* v *and timestamp* t *from address* a*; and for every* $(a, v, t) \in \mathsf{WS}$*, a write operation wrote value* v *at time* t *into address* a*.*

Remark 1. The protocol guarantees that $|\mathsf{RS}| = |\mathsf{WS}|$ and $\mathsf{RS}.\boldsymbol{a} = \mathsf{WS}.\boldsymbol{a}$ if the memory functions correctly. It is therefore clear that if Rd and Wr were to be permutations of each other, then it must be $|W| = |R|$, and $W.\boldsymbol{a}, R.\boldsymbol{a}$ are equal as sets.

4.2 Using Lookup Relations for Memory-Proving

The full Read/Write Memory-Proving Algorithm Π_{MV} is given in Appendix A.

Given the old memory $\mathsf{OM} = [\boldsymbol{v}_i]_{i=1}^T$; $\mathsf{Rd} = \mathsf{RS}||W = [(\boldsymbol{a}_i, \boldsymbol{v}_i, \boldsymbol{t}_i)]_{i=1}^k$ and $\mathsf{Wr} = \mathsf{WS}||R = [(\boldsymbol{a}_i, \boldsymbol{v}_i, \boldsymbol{t}_i)]_{i=1}^k$, which were constructed as described in Section 4. Let $\ell := |W| = |R|$.

In Π_{MV}, the prover takes as input $(\mathsf{OM}, \mathsf{Rd} = \mathsf{RS}||W, \mathsf{Wr} = \mathsf{WS}||R)$, and the verifier takes as input $(\mathsf{OM}^{\mathsf{V}}, \mathsf{RS}, \mathsf{WS})$, where OM^{V} is the verifier's stored

state of the memory. At the start of the protocol, the prover sends R, W to the verifier, and the verifier checks that they are sorted in the same order by addresses, i.e. $R.\boldsymbol{a} \stackrel{?}{=} W.\boldsymbol{a}$. The rest of the protocol is composed of the following three lookup-style protocols:

1. Use Π_{perm} to show that $(k, \mathsf{Rd}, \mathsf{Wr})$ are in $\mathcal{R}_{perm}$.
2. Use Π_{ivlk} to show that $(T, \ell, \mathsf{OM}, \mathbf{r}, \boldsymbol{b})$ are in $\mathcal{R}_{ivlk}$, where $\boldsymbol{b} := R.\boldsymbol{a}$ and $\boldsymbol{r} := R.\boldsymbol{v}$.
3. Suppose W, R are all ordered by the addresses of the entries. The prover computes $\mathbf{w} := W.\boldsymbol{v} - R.\boldsymbol{v} \in \mathbb{F}^{\ell}$, $\boldsymbol{b} = R.\boldsymbol{a} \in \mathbb{F}^{\ell}$, and then use them to efficiently compute $\Delta \in \mathbb{F}^T$ as follows.

$$\forall i \in [T], \quad \Delta_i = \begin{cases} \mathbf{w}_j & \text{if } i = \boldsymbol{b}_j \ \ \exists j \in [\ell] \\ 0 & \text{otherwise} \end{cases}$$

which the prover then use to efficiently compute the updated memory NM as follows:

$$\forall i \in [T], \quad \mathsf{NM}_i = \begin{cases} \mathsf{OM}_i + \Delta_i & \text{if } i = \boldsymbol{b}_j \ \ \exists j \in [\ell] \\ \mathsf{OM}_i & \text{otherwise} \end{cases}$$

This update only takes time linear in ℓ and sublinear in the total memory size T.

The prover then sends the NM to the verifier, who will compute $\mathbf{w}, \boldsymbol{b}$ from R, W and Δ from $\mathsf{NM}, \mathsf{OM}^{\mathsf{V}}$ by himself, and they run Π_{mu} to show that $(T, \ell, \Delta, \mathbf{w}, \boldsymbol{b})$ are in $\mathcal{R}_{\text{mu}}$.

If all the check passes, the verifier accepts NM as the correctly updated memory. In the next round, the previously computed NM becomes the new $\mathsf{OM}, \mathsf{OM}^{\mathsf{V}}$ for the prover and the verifier, respectively.

Complexity. It is a 3-move protocol (i.e. $k = 2$); the degree of the verifier is 4; the number of non-zero elements in the prover message is at most $8k+6\ell$. This is important because the prover pays linearly in the number of non-zero elements when computing the commitments. It is important to note that the total time of running the protocol is sublinear in T: running Π_{perm} is linear in k, and Π_{ivlk} and Π_{memup} are linear in ℓ; the final step of computing the updated memory can also be done in $O(\ell)$ time. As we assume $k << T$, i.e. the total number of entries in Rd, Wr are much smaller than the total size of the memory, the time it costs to run this memory-proving algorithm is $o(T)$.

Security. In this algorithm, Π_{perm} is $(3, 4k)$-special-sound, Π_{ivlk} is $((\ell+T), 2(\ell+T))$-special-sound, and Π_{memup} is $((\ell+T), 2(\ell+T))$-special-sound. Therefore, the algorithm is $((\ell+T), 2(\ell+T))$-special-sound overall.

Given the initial memory and the list of reads and writes, the algorithm verifies that they are consistent with the memory, and correctly updates the memory as instructed. First, the verifier checks that $R.\boldsymbol{a} = W.\boldsymbol{a}$, namely the initial read list R and W cover the exact same set of addresses and are sorted by

addresses in the same order, which is important for computing $\mathbf{w} := W.\boldsymbol{v} - R.\boldsymbol{v}$. V uses Π_{perm} to check that Rd, which contains W as the final ℓ entries, and Wr, which contains R as the final ℓ entries, are permutations of each other, which is a necessary indicator to show that offline memory-checking was performed correctly. Then, it uses Π_{ivlk} to check the consistency between the "initial read" list R and OM; more specifically, this shows that the first value of every address that appears in RS is indeed the value of that address in the old memory. It is important to note that the verifier keeps its own state of the old memory (which is the updated memory from the previous round) instead of relying on the prover to provide OM, so the prover cannot cheat by sending a fabricated state of memory. Finally, the verifier uses Π_{memup} to check that Δ, the list of change-in-values proposed by the prover, is simply a sparse representation of $\mathbf{w}$; this ensures that only the addresses recorded in some write entry are updated by the correct amounts reflected in the list of reads and writes, and the addresses never written to remain unchanged.

Speeding up Memory-Proving with logUp-GKR (described in Section 6). In the memory-proving protocol the provers messages are either $O(\ell)$ sized or $O(\ell)$ sparse. However, a more finegrained view looks at the actual bit-length of the messages. When compiling to an IVC, the prover needs to commit to all the messages and this operation is linear in the bit-length of messages. In the first round of the protocol the prover sends $R, W, \boldsymbol{m}, \Delta$. These values are representations of values read or written to memory, or their addresses and timestamps respectively. If the memory architecture only supports λ'-bit values, e.g. $\lambda' = 32$, then these values are all much smaller then the size of the field (which is proportional to the security parameter). In the second prover message, the prover sends multiple inverses. These values are large, even if the denominator itself is small. Note that all vectors are either $O(\ell)$ sized or are $O(\ell)$ spares.

Instead of sending the second round values and having the verifier perform the sum over the fractions, we will take the approach of logup [PH23], where the sum of fractions is computed using formal fractions. Importantly this does not require sending the fractions itself. This can significantly reduce the prover cost as it now does not need to commit to λ-bit "full" field elements.

The bivariate GKR protocol for logup as described in Section 6, requires the prover to commit to messages of size $c \cdot T^{1/c}$ for any parameter c. We can set c such that $T^{1/c}$ is a marginal cost, compared to committing to the "small" numerators and denominators.

In Π_{MP}, some of the vectors of fractions sent by the prover are sparse (E.g. $\boldsymbol{g}^{ivlk}, \boldsymbol{g}^{memup}$). Even though they contain T entries in total, at most ℓ of them are non-zero. We can take advantage of this sparseness in logUp GKR by setting d_i to 1 whenever $n_i = 0$ for all $i \in [T]$, and the prover will store $d_i - 1 = 0$ in its head to facilitate computation. [CMT12] shows that sumcheck is linear in the sparseness of the vector, which implies that GKR is also linear in the sparseness. Therefore, the time it takes to run logUp-GKR for those sparse polynomials will be sublinear in its size.

It is not necessary to run logUp-GKR from the sum over the entire vector. We can break the overall summation into a sum of several smaller summations, and run logUp-GKR for each. This reduces the rounds of GKR, and we can then check the final sum in a straightforward manner.

After running GKR, we check that the two fractions are equal by checking the products of one numerator and the other denominator are equal.

Overall prover efficiency We display the effciency metric of both the resulting plain protocol as well as the GKR-version in Table 4. The key prover efficiency is the $\mathsf{P}_{\mathsf{acc}}$ Time. In the plain protocol (see Appendix A), the prover first commits to R, W and $\boldsymbol{m}$. It also commits to Δ in order to compute the commitment to the updated memory NM. R, and W are each of size ℓ and contain tuples of three elements (a, v, t). Note that the a values will be exactly the same in R and W, so committing to R, W takes an MSM of size 5ℓ. Committing to Δ is an additional sparse MSM with ℓ non-zero elements. Committing to $\boldsymbol{m}$ is a negligble cost as $\boldsymbol{m}$ is a bit-vector. The prover also needs to commit to the vectors of fractions in the second round of the protocol. There are 6 such vectors that are either of size ℓ (for simplicity we assume $k = \ell$) or ℓ-sparse. Finally the accumulation prover needs to compute the cross terms for accumulation. We show how to do this in Section 4.3 and it requires an additional 3 ℓ-sparse MSMs. This results in a prover time that only requires committing to 15ℓ elements. We can replace the second round of the plain protocol using GKR. The GKR protocol requires committing to $O(T^{1/c})$ for an arbitrary constant c. This reduces the overall accumulation prover complexity to only 6ℓ elements, each of which is only as large as the elements stored in the table. Note that this is almost minimal, as even just recording a single read or write, already requires 3 elements, the address, the value and the timestamp.

Table 4. Efficiency Table for Accumulating Memory-Proving Protocol. See Table 3 for an explanation of the columns and symbols. For simplicity we assume that $k = \ell$. They are of the same order.

	P Time	$\lvert\mathsf{P}$ Msg$\rvert$	deg(V)	# P Msgs	$\mathsf{P}_{\mathsf{acc}}$ Time	$\mathsf{V}_{\mathsf{acc}}$ Time
Plain	$O(\ell)$	$5\ell\mathbb{T} + 6\ell\mathbb{F}$	5	2	$(6\ell, \mathbb{T})$-MSM $+(9\ell, \mathbb{F})$-MSM	$4\mathbb{G}$
Using GKR	$O(\ell)$	$5\ell\mathbb{T} + O(T^{1/c})$	7	$(c+1) \cdot \log T$	$(6\ell, \mathbb{T})$-MSM	$(c+1) \log T\mathbb{G}$

4.3 Accumulation prover runs in sublinear time

When we use the ProtoStar compiler to turn Π_{MP} into an accumulation scheme, the resulting $\mathsf{P}_{\mathsf{acc}}$ will run in time sublinear to the memory table size T, because the messages of the underlying special-sound prover, the cross error terms, and the updated accumulator can all be computed in $o(T)$ time.

Underlying special-sound prover runs in sublinear time As can be seen in Appendix B, all computations of the prover in Π_{MP} can be done in $o(T)$ time. Vectors $\boldsymbol{h}^{\text{perm}}, \boldsymbol{g}^{\text{perm}}, \boldsymbol{h}^{\text{ivlk}}, \boldsymbol{h}^{\text{mu}}$ all have size much smaller than T, so they can clearly be computed in sublinear time. Vectors $\boldsymbol{m}, \Delta, \boldsymbol{g}^{\text{ivlk}}, \boldsymbol{g}^{\text{mu}}$ have size T, but they all have at most ℓ nonzero entries, so an honest prover only needs $O(\ell)$ time to compute them. Updating the memory also takes sublinear time for an honest prover, since only Δ is sparse and only ℓ locations in the memory table need to be changed.

Computing the cross error terms in sublinear time The algorithm we describe in this subsubsection is only required when logUp-GKR (described in Section 6) is not used. Using logUp-GKR the cross error term computation (using the algorithms described in [BC23] takes only $O(c \cdot T^{1/c}) = o(T)$ time, i.e. is insignificant compared to the rest of the prover computation.

In the following, we use acc to represent the accumulator, π the current proof, and acc' to represent the updated accumulator. We refer the readers to Section 3.4 in [BC23] for a general formula on how cross error terms $[\mathbf{e}_j]_{j=1}^{d-1}$ are computed in the accumulation scheme. $\mathsf{P}_{\mathsf{acc}}$ will linearly combine the old accumulator and the current proof using a random challenge and use them as inputs to the decider (which is algebraic of degree d). For an honest prover, the zero coefficient of the polynomial should be the old accumulator's error term, and the highest-degree coefficient should be 0. The prover needs to then compute and commit to each of the coefficients in between (a.k.a. cross error terms). For most $\mathsf{V}_{\mathsf{sps}}$ checks, it is intuitive how the cross error terms can be computed in sublinear time, as the vectors will either have size much smaller than T or be sparse. The algorithm for computing the cross error term of the less intuitive $\mathbf{g}_i^{\text{ivlk}} \cdot (x_1 + x_2 \cdot i + \mathsf{OM}_i) \stackrel{?}{=} \boldsymbol{m}_i \ \ \forall i \in [T]$ check in sublinear time in the kth round of accumulation is given in Figure 1. We give a description of the idea below.

We need to compute the cross error terms for

$$\mathbf{g}_i^{\text{ivlk}} \cdot (x_1 + x_2 \cdot i + \mathsf{OM}_i) - \boldsymbol{m}_i \ \ \forall i \in [T]$$

For simplicity, we just use $\mathbf{g}$ to denote $\mathbf{g}^{\text{ivlk}}$, and OM' to denote $(x_1 \cdot \mathbf{1}_T + x_2 \cdot \boldsymbol{i} + \mathsf{OM})$, where $\boldsymbol{i} = [1, 2, \dots, T]$ is the address/index vector of OM. In vector form, the above expression is equivalent to

$$\mathbf{g} \circ \mathsf{OM}' - \boldsymbol{m}$$

When compiled by the Protostar compiler, the accumulation prover will combine acc and π using random linearly combination into the new accumulator acc', and then compute a commit to each of the expanded terms of the verifier polynomial in acc'. In other words, it will compute a commit to each of the expanded terms of the following,

$$\mathsf{acc}'.\mathbf{g} \circ \mathsf{acc}'.\mathsf{OM}' - \mathsf{acc}'.\mu \circ \mathsf{acc}'.\boldsymbol{m} =$$
$$(\mathsf{acc}.\mathbf{g} + X \cdot \pi.\mathbf{g}) \circ (\mathsf{acc}.\mathsf{OM}' + X \cdot \pi.\mathsf{OM}') - (\mathsf{acc}.\mu + X \cdot \pi.\mu) \cdot (\mathsf{acc}.\boldsymbol{m} + X \cdot \pi.\boldsymbol{m})$$

As stated earlier, for an honest prover, the zero coefficient should be 0 and the highest-degree coefficient should be 0, so the prover only needs to compute and commit to the degree-1 coefficient in this case, which is

$$\mathbf{e}_1 := \mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM}' + \pi.\mathbf{g} \circ \mathsf{acc}.\mathsf{OM}' + \mathsf{acc}.\mu \cdot \pi.\boldsymbol{m} + \pi.\mu \cdot \mathsf{acc}.\boldsymbol{m}$$

Since $\pi.\mathbf{g}, \pi.\boldsymbol{m}$ are sparse (only ℓ out of T nonzero entries), the complexity of computing $\pi.\mathbf{g} \circ \mathsf{acc}.\mathsf{OM}', \mathsf{acc}.\mu \cdot \pi.\boldsymbol{m}, \pi.\mu \cdot \mathsf{acc}.\boldsymbol{m}$ is clearly sublinear to T; moreover, since the two resulting Hadamard products will also be sparse vectors, committing them only takes sublinear time. $\pi.\mu$ is simply 1 in the accumulation scheme, so the commitment of $\pi.\mu \cdot \mathsf{acc}.\boldsymbol{m}$ is simply $\mathsf{Commit}(\mathsf{ck}, \mathsf{acc}.\boldsymbol{m})$, one of the commitments to accumulated prover messages included in the accumulator instance. The only term remaining to be analyzed is $\mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM}'$, which is equivalent to

$$\begin{aligned}\mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM}' &:= \mathsf{acc}.\mathbf{g} \circ (\pi.x_1 \cdot \mathbf{1}_T + \pi.x_2 \cdot \boldsymbol{i} + \pi.\mathsf{OM}) \\ &= \pi.x_1 \cdot \mathsf{acc}.\mathbf{g} + \pi.x_2 \cdot \mathsf{acc}.\mathbf{g} \circ \boldsymbol{i} + \mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM}\end{aligned}$$

$\pi.x_1 \cdot \mathsf{acc}.\mathbf{g}$ can be easily committed as a scalar multiplication of the commitment of $\mathsf{acc}.\mathbf{g}$, which is included in the accumulator instance as one of the accumulated commitments. As for the other two terms, we can store values from the previous round of accumulation to aid the computation.

Let $\hat{\mathsf{acc}}$ and $\hat{\pi}$ denote the accumulator and the proof in the previous round, and r denote the challenge in the previous round, i.e. $\mathsf{acc} = \hat{\mathsf{acc}} + r \cdot \hat{\pi}$. Then, we can re-express the terms in the following way,

$$\begin{aligned}\pi.x_2 \cdot \mathsf{acc}.\mathbf{g} \circ \boldsymbol{i} &= \pi.x_2 \cdot (\hat{\mathsf{acc}}.\mathbf{g} + r \cdot \hat{\pi}.\mathbf{g}) \circ \boldsymbol{i} \\ &= \pi.x_2 \cdot (\hat{\mathsf{acc}}.\mathbf{g} \circ \boldsymbol{i} + r \cdot \hat{\pi}.\mathbf{g} \circ \boldsymbol{i}) \\ \mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM} &= (\hat{\mathsf{acc}}.\mathbf{g} + r \cdot \hat{\pi}.\mathbf{g}) \circ \pi.\mathsf{OM} \\ &= \hat{\mathsf{acc}}.\mathbf{g} \circ \pi.\mathsf{OM} + r \cdot \hat{\pi}.\mathbf{g} \circ \pi.\mathsf{OM} \\ &= \hat{\mathsf{acc}}.\mathbf{g} \circ (\hat{\pi}.\mathsf{OM} + \hat{\pi}.\Delta) + r \cdot \hat{\pi}.\mathbf{g} \circ \pi.\mathsf{OM} \\ &= \hat{\mathsf{acc}}.\mathbf{g} \circ \hat{\pi}.\mathsf{OM} + \hat{\mathsf{acc}}.\mathbf{g} \circ \hat{\pi}.\Delta + r \cdot \hat{\pi}.\mathbf{g} \circ \pi.\mathsf{OM}\end{aligned}$$

Observe that $\hat{\mathsf{acc}}.\mathbf{g} \circ i$ and $\hat{\mathsf{acc}}.\mathbf{g} \circ \hat{\pi}.\mathsf{OM}$ were already computed and committed to in the previous round of accumulation. The other terms can be computed and committed to in time sublinear to T since $\hat{\pi}.\Delta$ and $\hat{\pi}.\mathbf{g}$ are sparse, and the resulting Hadamard products are also sparse. Thus, the prover can compute and commit to $\mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM}'$ in time sublinear to T, which means the prover will be able to compute and commit to $\mathbf{e}_1$ in sublinear time as well.

Updating the accumulator in sublinear time The prover still needs to compute the new accumulator $\mathsf{acc}'.\mathbf{g} \leftarrow \mathsf{acc}.\mathbf{g} + X \cdot \pi.\mathbf{g}$ and $\mathsf{acc}'.\mathsf{OM} \leftarrow \mathsf{acc}.\mathsf{OM} + X \cdot \pi.\mathsf{OM}$. While computing $\mathsf{acc}'.\mathbf{g}$ clearly takes sublinear time because $\pi.\mathbf{g}$ is sparse, the complexity for naively computing $\mathsf{acc}'.\mathsf{OM}$ is linear in T. Here, we introduce a trick that will enable us to accumulate OM in sublinear time. See Figure 2 for the algorithm.

Let the subscript (j) denote the items in the jth round. We make a key observation that for every round j and every $i \in [T]$, $\pi_{(j)}.\mathsf{OM}_i$ only differs from $\pi_{(j-1)}.\mathsf{OM}_i$ if $\pi_{(j-1)}.\Delta_i \neq 0$. Then, for $k > j > 0$ such that j is most recent round before round k such that $\pi_{(j-1)}.\Delta_i \neq 0$, the accumulator of OM_i for every $i \in [T]$ in round k will be

$$\begin{aligned}
&\mathsf{acc}_{(k)}.\mathsf{OM}_i \\
&= \mathsf{acc}_{(k-1)}.\mathsf{OM}_i + r_{(k)} \cdot \pi_{(k)}.\mathsf{OM}_i \\
&= \mathsf{acc}_{(k-2)}.\mathsf{OM}_i + r_{(k-1)} \cdot \pi_{(k-1)}.\mathsf{OM}_i + r_{(k)} \cdot \pi_{(k-1)}.\mathsf{OM}_i + r_{(k)} \cdot \pi_{(k-1)}.\Delta_i \\
&= \mathsf{acc}_{(k-2)}.\mathsf{OM}_i + \big(r_{(k-1)} + r_{(k)}\big) \cdot \pi_{(k-1)}.\mathsf{OM}_i + r_{(k)} \cdot \pi_{(k-1)}.\Delta_i \\
&= \mathsf{acc}_{(j)}.\mathsf{OM}_i + \big(r_{(j+1)} + r_{(j+2)} + \cdots + r_{(k)}\big) \cdot \pi_{(k-1)}.\mathsf{OM}_i + r_{(k)} \cdot \pi_{(k-1)}.\Delta_i \\
&= \mathsf{acc}_{(j)}.\mathsf{OM}_i + \Big(\sum_{q=j+1}^{k} r_{(q)}\Big) \cdot (\pi_{(k)}.\mathsf{OM}_i - \pi_{(k-1)}.\Delta_i) + r_{(k)} \cdot \pi_{(k-1)}.\Delta_i
\end{aligned}$$

Let $r^*_{(j)} := \sum_{q=1}^{j} r_{(q)}$ denote the sum of all challenges up to round j. Using the observation above, we can let the prover cache the sum of all the challenges seen so far $r^* = r^*_{(k)}$, the update vector from previous round $\pi_{(k-1)}.\Delta$ [8], and a separate vector $\mathbf{R} \in \mathbb{F}^T$ (initialized to all zeros at the beginning of accumulation) such that $\mathbf{R}_i = r^*_{(j)}$ where j is the most recent round such that $\pi_{(j)}.\Delta_i \neq 0$ respectively for every $i \in [T]$. Then, $\mathsf{acc}_{(k)}.\mathsf{OM}$ can be expressed as,

$$\mathsf{acc}_{(k)}.\mathsf{OM} \leftarrow \mathsf{acc}_{(k-1)}.\mathsf{OM} + (r^* \cdot \mathbf{1}_T - \mathbf{R}) \circ (\pi_{(k)}.\mathsf{OM} - \pi_{(k-1)}.\Delta) + r_k \cdot \pi_{(k-1)}.\Delta$$

In round k, for every $i \in [T]$, we evaluate $\mathsf{acc}_{(k)}$ and set $\mathbf{R}_i \leftarrow r^*$ if and only if $\pi_{(k-1)}.\Delta_i = 0$. This way, in every round, only ℓ positions of the accumulated OM and ℓ positions of $\mathbf{R}$ need to be computed, achieving sublinear complexity. Whenever we need to use $\mathsf{acc}_{(k)}.\mathsf{OM}$ (either in $\mathsf{P}_{\mathsf{acc}}$ and in the decider), we substitute it with $\mathsf{acc}_{(k-1)}.\mathsf{OM} + (r^* \cdot \mathbf{1}_T - \mathbf{R}) \circ \pi_{(k)}.\mathsf{OM}$.

Note that, importantly, using this delayed-evaluation trick will not change the computations done by $\mathsf{V}_{\mathsf{acc}}$ because the commitment to $\mathsf{acc}_{(k)}.\mathsf{OM}$ will still be computed as the random linear combination between the commitment to $\mathsf{acc}_{(k-1)}.\mathsf{OM}$ and the commitment to $\pi_{(k)}.\mathsf{OM}$.

5 An accumulation scheme for GKR

The GKR protocol is special-sound, but using GKR for the lookup relations naively in accumulation will result in $\log^2 \ell$ rounds (assuming ℓ is the number of inputs), which is expensive since the ProtoStar compiler pays linearly in the number of rounds. Hence, we wish to design a version of the GKR protocol that is better suited for accumulation, i.e. one that takes fewer rounds but retains the property that the prover only pays for the input and not any intermediate values. The core ingredient will be a bivariate sumcheck protocol which is well suited for accumulation.

[8] This is also needed for computing the cross error terms, so we don't need to store it again. See the subsection above and Appendix B

5.1 Subprotocol for the verifier to efficiently evaluate a function

Bivariate sumcheck requires the verifier to evaluate polynomials of degree $\Theta(\sqrt{n})$, where n is the width of the GKR circuit. This is prohibitively large. Fortunately, we can transform evaluation into a low-degree check by sending additional witnesses. We describe the low-degree evaluation protocol below.

Subprotocol $\mathit{\Pi}_{eval}$ for evaluating $f : \mathbb{F}^k \to \mathbb{F}$ at some $\boldsymbol{a} \in (\mathbb{F} \setminus \mathbb{H})^k$ s.t. $m := |\mathbb{H}| > \deg(f)$

Prover $\mathsf{P}(f, \boldsymbol{a} = [\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k])$		**Verifier** $\mathsf{V}(f, \boldsymbol{a}, [f(\boldsymbol{x})]_{\boldsymbol{x} \in \mathbb{H}^k})$
$\boldsymbol{a}^i \leftarrow [\boldsymbol{a}_1^i, \ldots, \boldsymbol{a}_k^i] \quad \forall i \in \{2, 4, \ldots, m\}$		
$A := (\boldsymbol{a}, \boldsymbol{a}^2, \boldsymbol{a}^4 \ldots, \boldsymbol{a}^m)$		
$L_x^{\mathbb{H}}(u) := \frac{c_x(u^m - 1)}{u - x} \quad \forall x \in \mathbb{H}$	$\xrightarrow{A,\ L_x^{\mathbb{H}}(u)\ \forall x \in \mathbb{H}}$	$A(0) \stackrel{?}{=} \boldsymbol{a} \quad A(i) \stackrel{?}{=} A(i-1)^2 \quad \forall i \in \{1, \ldots, \log m - 1\}$
		$\prod_{j=1}^{k} L_{\boldsymbol{x}_j}^{\mathbb{H}}(\boldsymbol{a}_j) \cdot \prod_{j=1}^{k} (\boldsymbol{a}_j - \boldsymbol{x}_j) \stackrel{?}{=} \prod_{j=1}^{k} c_{\boldsymbol{x}_j} \cdot (A(\log m, j) - 1) \quad \forall \boldsymbol{x} \in \mathbb{H}^k$
		$f(\boldsymbol{a}) \leftarrow \sum_{\boldsymbol{x} \in \mathbb{H}^k} \Big(\prod_{j=1}^{k} L_{\boldsymbol{x}_j}^{\mathbb{H}}(\boldsymbol{a}_j) f(\boldsymbol{x}) \Big)$

Efficiency. The verification degree is $2k$. The prover sends $m + k \cdot \log m$ values.

In the protocol above, $\mathbb{H}$ is a multiplicative subgroup of $\mathbb{F}$, and we assume $m := |\mathbb{H}|$ is a multiple of 2. This implies that the ith element of $\mathbb{H}$ is the ith root of unity and also that the Lagrange polynomial L_x has the form described above, where c_x is the barycentric weight. Note that P sends over a $\log m \times k$ matrix A. $A(i) := \boldsymbol{a}^{2^i}$ denotes the ith row of A, and $A(i,j) := \boldsymbol{a}_j^{2^i}$.

Security. The protocol has perfect completeness and soundness. The first line of checks ensure that the matrix A was computed correctly as claimed by the prover. In the second line of check, note that $A(\log m, j) = \boldsymbol{a}_j^{2^{\log m}} = \boldsymbol{a}_j^m$. Hence if the equality holds, we have

$$eq(\boldsymbol{x}, \boldsymbol{a}) = \prod_{j=1}^{k} \frac{c_{\boldsymbol{x}_j}(\boldsymbol{a}_j^m - 1)}{\boldsymbol{a}_j - \omega_{\boldsymbol{x}_j}} = \prod_{j=1}^{k} L_{\boldsymbol{x}_j}^{\mathbb{H}}(\boldsymbol{a}_j) \quad \forall \boldsymbol{x} \in \mathbb{H}^k$$

which indicates that $eq(\boldsymbol{x}, \boldsymbol{a})$ was computed correctly as claimed by the verifier. This implies that the two polynomials $f(\boldsymbol{a})$ and $\sum_{x \in \mathbb{H}^k} eq(\boldsymbol{x}, \boldsymbol{a}) f(\boldsymbol{x})$ are equal on m^k points. Since both of these polynomials have degree strictly smaller than m, being equal on m^k points indicates that they are the same polynomial.

5.2 Bivariate Sumcheck

We describe a bivariate sumcheck protocol because the Protostar compiler pays linearly in the number of rounds, and hence the number of variables. While there is a tradeoff between the number of variables and the degree in each variable, high degrees can be tolerated in the final accumulation scheme because the decider only runs once.

Bivariate Sumcheck to prove $\sum_{x\in\mathbb{G}_1,y\in\mathbb{G}_2} f(x,y)=T$, where $\mathbb{G}_1,\mathbb{G}_2\subset\mathbb{H}$ s.t. $m:=|\mathbb{H}|=\deg(f)+1$

Prover $\mathsf{P}(f,T)$		**Verifier** $\mathsf{V}(f,T)$
$f_1(X)\leftarrow\sum_{y\in\mathbb{G}_2} f(X,y)$	$f_1(\omega_i)\ \forall i\in[m]$ $\rightarrow$	
	$\leftarrow$ a	$a\leftarrow\$\ \mathbb{F}\setminus\mathbb{H}$
$f_2(Y)\leftarrow f(a,Y)$	$f_2(\omega_i)\ \forall i\in[m]$ $\rightarrow$	
	$\leftarrow$ b	$b\leftarrow\$\ \mathbb{F}\setminus\mathbb{H}$
$T^*\leftarrow f_2(b)$	T^* $\rightarrow$	Use Π_{eval} to evaluate $f_1(a),f_2(b)$
		$\sum_{x\in\mathbb{G}_1} f_1(x)\stackrel{?}{=}T$
		$\sum_{y\in\mathbb{G}_2} f_2(y)\stackrel{?}{=}f_1(a)$
		$T^*\stackrel{?}{=}f_2(b)$
		$T^*\stackrel{?}{=}f(a,b)$

Security. The protocol is clearly perfectly complete. It is (m,m)-special-sound. For a fixed challenge a_i, to show that $f_2(Y)=f(a_i,Y)$ requires the equality to hold for $\deg(f_2)+1=\deg_Y(f)+1\leq\deg(f)+1=m$ different challenges for Y, i.e. $b_1,\ldots,b_m$. Then, since $f_2(Y)=f(a_i,Y)$, checking whether $\sum_{y\in\mathbb{G}_2} f_2(y)=f_1(a_i)$ is equivalent to checking $\sum_{y\in\mathbb{G}_2} f(a_i,y)=f_1(a_i)$ for any fixed a_i. To show that $f_1(X)=\sum_{y\in\mathbb{G}_2} f(X,y)$ requires the equality to hold for $\deg(f_1)+1=\deg_X(f)+1\leq\deg(f)+1=m$ different challenges for X, i.e. $a_1,\ldots,a_m$. Therefore, with m different challenges on X and m different challenges on Y, the verifier can be sure that $\sum_{x\in\mathbb{G}_1} f_1(x)=\sum_{x\in\mathbb{G}_1,y\in\mathbb{G}_2} f(x,y)$. Finally, since $\sum_{x\in\mathbb{G}_1} f(x)=T$, it is verified that $\sum_{x\in\mathbb{G}_1,y\in\mathbb{G}_2} f(x,y)=T$.

Table 5. Efficiency Table for Accumulating Bivariate SumCheck Using Subprotocol Π_{eval} ($n:=|f|\geq m^2$). In most applications f will be a composition of multiple polynomials; in order to compute $f_1(X)$, the prover will need to perform FFTs which take $n\log n$ operations in $\mathbb{F}$.

P Time	$\lvert\mathsf{P}$ Msg$\rvert$	deg(V)	# P Msgs
$n\log n\mathbb{F}$	$4\sqrt{n}+o(\sqrt{n})\mathbb{F}$ or hashes	2	3

The number of P messages shown in Table 5 is the number when the polynomial f in the sumcheck is non-sparse. Since the polynomial f will be sparse (sublinear in the memory size T) when performing memory-proving using our lookup-powered protocol, the actual number of P messages will be much smaller.

5.3 Batching subprotocol for GKR

Description of the Batching Subprotocol for batching k sumchecks into one:

- Given a list of tuples $[(g_j \in \mathbb{F}[X_1, \ldots, X_c], T_j \in \mathbb{F})]_{j=1}^{k}$ and $\mathbb{H}^c$, such that $\sum_{\boldsymbol{x}\in\mathbb{H}^c} g_j(\boldsymbol{x}) = T_j$ for all $j \in [k]$.
- V chooses $r \leftarrow\!\!\$\ \mathbb{F}$ at random and sends it to P.
- V batches all k sumchecks checks into one as follows

$$\sum_{\boldsymbol{x}\in\mathbb{H}^c} f(\boldsymbol{x}) \stackrel{?}{=} \sum_{j=1}^{k} r^{j-1} T_j$$

for $f(\boldsymbol{x}) := \sum_{j=1}^{k} r^{j-1} g_j(\boldsymbol{x})$. Note that if $g_j(\boldsymbol{x}) = eq(\boldsymbol{z}_j, \boldsymbol{x}) g(\boldsymbol{x})$ then $f(\boldsymbol{x}) = g(\boldsymbol{x}) \cdot (\sum_{j\in[k]} r^{j-1} \cdot eq(\boldsymbol{z}_j, \boldsymbol{x}))$

Efficiency. In GKR we call this protocol with $g_j(\boldsymbol{x}) = g(\boldsymbol{x}) \cdot eq(\boldsymbol{z}_j, \boldsymbol{x})$. This means that the complexity of the batched sumcheck is equivalent to the complexity of sumcheck over g plus evaluating a random linear combination of the eq functions. This is only a small additive overhead over a single sumcheck of g.

Security. The batching subprotocol is perfectly complete. It is k-special-sound. We can define the following degree $(k-1)$ polynomial:

$$\begin{aligned} g(r) &:= \Big(\sum_{x\in\mathbb{H}^c} f(\boldsymbol{x}) \Big) - \Big(\sum_{j=1}^{k} r^{j-1} T_j \Big) \\ &= \sum_{x\in\mathbb{H}^c} \Big(\sum_{j=1}^{k} r^{j-1} g_j(\boldsymbol{x}) \Big) - \Big(\sum_{j=1}^{k} r^{j-1} T_j) \Big) = \sum_{j=1}^{k} r^{j-1} \Big(\sum_{x\in\mathbb{H}^c} g_j(\boldsymbol{x}) - T_j \Big) \end{aligned}$$

If $g(r)$ is the zero polynomial, then $\sum_{x\in\mathbb{H}^c} f(\boldsymbol{x}) = \sum_{j=1}^{k} r^{j-1} T_j$. In order to get $g = 0$, we need $\deg(g) + 1 = k$ points of r at which $g(r) = 0$.

6 LogUp GKR protocol using the batching subprotocol

We incorporate our batching subprotocol with logUp-GKR [PH23], where the circuit is designed for computing the cumulative sums of the fractions using projective coordinates for the additive group of $\mathbb{F}$. We will use this protocol to do the verifier checks for the lookup-style arguments in Π_{MP}.

Layer 0 denotes the output layer in the circuit. Let $\sqrt{m} := |\mathbb{H}|$, then $\mathbb{H}^2$ is a $\sqrt{m} \times \sqrt{m}$ 2-dimensional square. Assume $\log m/2$ is a positive integer w.l.o.g. In the protocol, $|\mathbb{H}_i| = 2|\mathbb{H}_{i-1}|$ for $i = 1, \ldots, \log m/2$, with $\mathbb{H}_0 = \{1, 1\}$ and $\mathbb{H}_{\log m/2} = \mathbb{H}$. The protocol has $\log m$ rounds in total. We describe the protocol in four phases.

Phase 1 contains round 0 of logUp-GKR. At the end of phase 1, V uses the batching subprotocol and linearly combines the four claimed evaluations sent by P using a random value.

In Phase 2 and 3, V continues using the batching subprotocol in each round. Phase 2 contains rounds 1 to $\log m/2 - 1$ of logUp-GKR. Round i in Phase 2 does sumcheck in $\mathbb{H}_0 \times \mathbb{H}_i$. Phase 3 contains rounds $\log m/2$ to $\log m - 1$ of logUp-GKR. Round i in Phase 3 does sumcheck in $\mathbb{H}_{i-\log m/2+1} \times \mathbb{H}$.

Finally, Phase 4 contains the final, direct check done by V at the input layer of the circuit.

Phase 1:

- At the start of the protocol, P sends over functions $D : \mathbb{H}^2 \to \mathbb{F}$ and $N : \mathbb{H}^2 \to \mathbb{F}$ claimed to equal d_0 and n_0 (the output functions that satisfy $d_0(1,1) = d^*$, the denominator in the cumulative sum of the fractions, and $n_0(1,1) = n^*$, the numerator in the cumulative sum), respectively.
- V picks random $x_0, y_0 \in \mathbb{F}^2$ and random $r_0 \in \mathbb{F}$, and lets $T_0 \leftarrow \widetilde{N}(x_0, y_0) + r_0 \cdot \widetilde{D}(x_0, y_0)$.
- In round $i = 0$:
  - Define the univariate polynomial

$$f_{r_0}(v) := L^{\mathbb{H}_0}(y_0, v) \cdot \Big(n_1(1, v) \cdot d_1(1, \omega \cdot v) + n_1(1, \omega \cdot v) \cdot d_1(1, v) + r_0 \cdot d_1(1, v) \cdot d_1(1, \omega \cdot v)\Big)$$

  - P claims that $\sum_{v \in \mathbb{H}_0} f_{r_0}(v) = T_0$.
  - P and V apply the sum-check protocol to f_{r_0}, up until V's final check in that protocol, when V must evaluate f_{r_0} at a randomly chosen point $y_1 \in \mathbb{F}$.
  - P sends over $[T_j^{(1)}]_{j=1}^4$, which are the claimed evaluations of $\widetilde{n}_1$ and $\widetilde{d}_1$ on $(1, y_1)$ and $(1, \omega \cdot y_1)$:

$$T_1^{(1)} := \widetilde{n}_1(1, y_1) \quad T_2^{(1)} := \widetilde{n}_1(1, \omega \cdot y_1)$$
$$T_3^{(1)} := \widetilde{d}_1(1, y_1) \quad T_4^{(1)} := \widetilde{d}_1(1, \omega \cdot y_1)$$

  - V uses $[T_j^{(1)}]_{j=1}^4$ to perform the final check in the sum-check protocol.
  - V chooses $r_1 \leftarrow\!\$\; \mathbb{F}$ at random, sends it to P, and sets

$$T_1 \leftarrow \sum_{j=1}^{4} r_1^{j-1} T_j^{(1)}$$

Phase 2:

- For $i = 1, \ldots, \log m/2$: use the batching subprotocol to combine the four checks for the evaluations of $\widetilde{n}_i, \widetilde{d}_i$ on $(1, y_i)$ and $(1, \omega^{m/2^i} \cdot y_i)$ into one sumcheck.

- Define the univariate polynomial

$$
\begin{aligned}
f_{r_i}^{(i)}(v) :=& \Big(L^{\mathbb{H}_i}(y_i, v) + r_i \cdot L^{\mathbb{H}_i}(\omega^{m/2^i} \cdot y_i, v)\Big) \\
&\cdot \Big(n_{i+1}(1, v) \cdot d_{i+1}(1, \omega^{m/2^i} \cdot v) \\
&\quad + n_{i+1}(1, \omega^{m/2^i} \cdot v) \cdot d_{i+1}(1, v)\Big) \\
&+ \Big(r_i^2 \cdot L^{\mathbb{H}_i}(y_i, v) + r_i^3 \cdot L^{\mathbb{H}_i}(\omega^{m/2^i} \cdot y_i, v)\Big) \\
&\cdot \Big(d_{i+1}(1, v) \cdot d_{i+1}(1, \omega^{m/2^i} \cdot v)\Big)
\end{aligned}
$$

- P claims that $\sum_{v \in \mathbb{H}_i} f_{r_i}^{(i)}(v) = T_i$.
- P and V apply the sum-check protocol to $f_{r_i}^{(i)}$, up until V's final check in that protocol, when V must evaluate $f_{r_i}^{(i)}$ at a randomly chosen point $y_{i+1} \in \mathbb{F}$.
- P sends over $[T_j^{(i+1)}]_{j=1}^4$, which are the claimed evaluations of $\widetilde{n}_{i+1}$ and $\widetilde{d}_{i+1}$ on $(1, y_{i+1})$ and $(1, \omega^{m/2^i} \cdot y_{i+1})$
- V uses $[T_j^{(i+1)}]_{j=1}^4$ to perform the final check in the sumcheck protocol.
- V chooses $r_{i+1} \leftarrow\!\!\$\, \mathbb{F}$ at random and sets $T_{i+1} \leftarrow \sum_{j=1}^4 r_{i+1}^{j-1} T_j^{(i+1)}$.

Phase 3:

- For $i = \log m/2 + 1, \ldots, \log m - 1$: use the batching subprotocol to combine the four checks for the evaluations of $\widetilde{n}_i, \widetilde{d}_i$ on (x_i, y_i) and $(\omega^{m/2^i} \cdot x_i, y_i)$ into one sumcheck.[a]
  - Let $i' := i - \log m/2$ Define the bivariate polynomial

$$
\begin{aligned}
&f_{r_i}^{(i)}(u, v) := \\
&\Big(L^{\mathbb{H}_{i'}}(x_i, u) \cdot L^{\mathbb{H}}(y_i, v) + r_i \cdot L^{\mathbb{H}_{i'}}(\omega^{m/2^i} \cdot x_i, u) \cdot L^{\mathbb{H}}(y_i, v)\Big) \\
&\cdot \Big(n_{i+1}(u, v) \cdot d_{i+1}(\omega^{m/2^i} \cdot u, v) + n_{i+1}(\omega^{m/2^i} \cdot u, v) \cdot d_{i+1}(u, v)\Big) \\
&+ \Big(r_i^2 \cdot L^{\mathbb{H}_{i'}}(x_i, u) \cdot L^{\mathbb{H}}(y_i, v) + r_i^3 \cdot L^{\mathbb{H}_{i'}}(\omega^{m/2^i} \cdot x_i, u) \cdot L^{\mathbb{H}}(y_i, v)\Big) \\
&\cdot \Big(d_{i+1}(u, v) \cdot d_{i+1}(\omega^{m/2^i} \cdot u, v)\Big)
\end{aligned}
$$

  - P claims that $\sum_{u,v \in \mathbb{H}_{i'}, \mathbb{H}} f_{r_i}^{(i)}(u, v) = T_i$.
  - P and V apply the sum-check protocol to $f_{r_i}^{(i)}$, up until V's final check in that protocol, when V must evaluate $f_{r_i}^{(i)}$ at a randomly chosen point $(x_{i+1}, y_{i+1}) \in \mathbb{F}^2$.

- P sends over $[T_j^{(i+1)}]_{j=1}^4$, which are the claimed evaluations of $\widetilde{n}_{i+1}$ and $\widetilde{d}_{i+1}$ on (x_{i+1}, y_{i+1}) and $(\omega^{m/2^i} \cdot x_{i+1}, y_{i+1})$.
- V uses $[T_j^{(i+1)}]_{j=1}^4$ to perform the final check in the sumcheck protocol.
- V chooses $r_{i+1} \leftarrow\$ \mathbb{F}$ at random and sets $T_{i+1} \leftarrow \sum_{j=1}^4 r_{i+1}^{j-1} T_j^{(i+1)}$.

[a] It is implicitly defined that $x_{\log m/2} = 1$.

Phase 4:
– Let $d := \log m$. V checks directly whether

$$T_d \stackrel{?}{=} \widetilde{n}_d(x_d, y_d) + r_d \cdot \widetilde{n}_d(\omega \cdot x_d, y_d) + r_d^2 \cdot \widetilde{d}_d(x_d, y_d) + r_d^3 \cdot \widetilde{d}_d(\omega \cdot x_d, y_d)$$

Further reducing communication and rounds. The bivariate GKR protocol only uses $3 \cdot \log_2(k)$ rounds and has communication complexity $\sqrt{k}$. This is significantly fewer rounds than GKR with the standard multi-linear sumcheck which would use $O(\log^2 k)$ rounds. In most cases the additional communication of $\sqrt{k}$ is only marginal, as the prover needed to commit to the input and output layers (of size k). However, in particular when using the protocol with sparse inputs the $\sqrt{k}$ may indeed become dominant.

c-variate sumcheck. Fortunately, we can naturally generalize the protocol by relying on a c-variate sumcheck. In this case, the protocol has $(c + 1) \cdot \log_2(k)$ rounds but the communication complexity is only $O(c \cdot k^{1/c})$. This exponentially decays as c gets bigger. In the protocol we would expand the dimension in each variable, one by one, such that the size of the layer still grows by a factor of 2 in each round.

Higher degree reductions. Another optimization is to combine 2 rounds of GKR into one. This increases the degree of the GKR round polynomial by a factor of 2 but also decreases the number of rounds by the same factor. Using the Protostar compiler we only pay for the highest degree verification check, so this optimizations is particularly useful if the circuit already contains high degree checks.

Splitting the summation for round reduction. The core motivation for proving the fractional sum within GKR instead of proving it directly, is that the prover does not need to commit to the inverses. When the numerator and denominator are composed of c-bit values and $\log |\mathbb{F}| = \Theta(\lambda)$ then this can reduce the commitment cost from $O(\lambda m)$ to just $O(c \cdot m)$, i.e. save a factor of $\frac{\lambda}{c}$. Note that the circuit computed by GKR has a triangle form and each layer is half the width of its parent layer. We can take advantage of this by splitting the sum into p parts each of $\frac{m}{p}$, component. The prover would need to commit to the outputs of each sum, i.e. p fractions. The total commitment cost is $O(c \cdot m + \lambda p)$. As long as

$p \geq \frac{c \cdot m}{\lambda}$, the total commitment cost is still $O(c \cdot m)$. However, the sums computed within GKR are now significantly smaller, and only $\log \lambda - \log c$ GKR layers are required. A similar optimization applies when the input layer is sparse; however, then more layers are required to significantly bring down the cost of committing to the dense output layer.

Table 6. Efficiency Table for Accumulating GKR. See Table 3 for an explanation of the columns and units.

Variant	P Time	|P Msg|	deg(V)	# P Msgs	$\mathsf{P}_{\mathsf{acc}}$ Time	$\mathsf{V}_{\mathsf{acc}}$ Time
bivariate	$O(n \log n)$	$O(n^{1/2})$	7	$3 \log n$	$O(n^{1/2})$-MSM $+O(n \log n)\mathbb{F}$	$3 \log n + 2\mathbb{G}$
c-variate $\sum$	$O(n \log n)$	$O(c \cdot n^{1/c})$	7	$(c+1) \log n$	$O(c \cdot n^{1/c})$-MSM $+O(n \log n)\mathbb{F}$	$(c+1) \log n$ $+2\mathbb{G}$
k-round GKR	$O(n \log n)$	$O(c \cdot n^{1/c})$	7	$(c+1)k$	$O(c \cdot n^{1/c})$-MSM $+O(n \log n)\mathbb{F}$	$(c+1) \cdot k$ $+2\mathbb{G}$

Other applications of GKR. GKR has many applications beyond the use in lookup protocols. For instance, GKR can be used to more efficiently prove that a scalar multiplication was done correctly. This is particularly intriguing as group scalar multiplications are the most expensive operations within the recursive circuit. Concretely the GKR circuit for group scalar multiplication takes as input, a scalar s in bit representation $s_{\lambda-1} \ldots s_1 s_0$ where s_i is either 0 or 1 for every $i \in \{0, \ldots, \lambda - 1\}$ and $s_{\lambda-1}$ is the most significant bit, a base elliptic curve point in projective coordinates (X, Y, Z), and an output curve point also in projective coordinates. The reason to use projective coordinates is that the double-and-add operation can be represented using low-degree (specifically degree 11) algebraic formulas [RCB16]. Using GKR, the prover would only need to commit to 6 scalars and λ bits. However, the depth of the circuit might be a bottleneck. We can further reduce the number of layers by providing more intermediary values. E.g. by providing k additional curve points, we can reduce the depth from λ to $\lambda/(k+1)$.

Concrete Formula for short Weierstass curves $Y^2 = X^3 + b$. Suppose we are given a scalar s in bit representation $s_{\lambda-1} \ldots s_1 s_0$, and a base elliptic curve point in projective coordinates $G = (X, Y, Z)$ which is represented using three scalars such that $Y^2 Z = X^3 + bZ^3$.. We give a concrete example below for scalar multiplication $s \cdot G$ using GKR in the special case short Weierstrass curves with $a = 0$. Before running the GKR protocol, prover sends $s = s_{\lambda-1} \ldots s_0$ and G. Note that when the protocol is compiled using the Protostar compiler, s and G will be sent in commitments. Even so, this will not be a problem for accessing s and G while running GKR, because only the decider will be running GKR with the prover and the decider has access to the original values of all the prover messages/commitments.

Initialize $A_\lambda = (X_\lambda, Y_\lambda, Z_\lambda)$ to the identity point $(0, 1, 0)$. At the ith layer, suppose we have intermediary elliptic curve projective coordinates $A_i = (X_i, Y_i, Z_i)$. Let $A_i' = (X_i', Y_i', Z_i')$ be the point of doubled coordinates of A_i. Specifically, the doubling formulas are

$$\begin{aligned} X_i' &= 2X_iY_i(Y_i^2 - 9bZ_i^2), \\ Y_i' &= (Y_i^2 - 9bZ_i^2)(Y_i^2 + 3bZ_i^2) + 24bY_i^2Z_i^2, \\ Z_i' &= 8Y_i^3Z_i. \end{aligned}$$

Then, using the double-and-add heuristic, we compute $A_{i-1} = (X_{i-1}, Y_{i-1}, Z_{i-1})$ as

$$\begin{aligned} X_{i-1} =&(1 - s_i) \cdot X_i' \\ &+ s_i \cdot \Big((X_i'Y + XY_i')(Y_i'Y - 3bZ_i'Z) - 3b(Y_i'Z + YZ_i')(X_i'Z + XZ_i') \Big) \\ Y_{i-1} =&(1 - s_i) \cdot Y_i' \\ &+ s_i \cdot \Big((Y_i'Y + 3bZ_i'Z)(Y_i'Y - 3bZ_i'Z) + 9bX_i'X(X_i'Z + XZ_i') \Big) \\ Z_{i-1} =&(1 - s_i) \cdot Z_i' \\ &+ s_i \cdot \Big((Y_i'Z + YZ_i')(Y_i'Y + 3bZ_i'Z) + 3X_i'X(X_i'Y + XY_i') \Big) \end{aligned}$$

$A_0 = (X_0, Y_0, Z_0)$ being the final output of the scalar multiplication $s \cdot G$.

The degree is 11 as (X_i', Y_i', Z_i') can be computed using a degree 4 formula and X_{i-1} has a $s_i \cdot X_i'Y \cdot Y_i'Y$ term. We can turn these algebraic expressions into a layered GKR protocol by having each layer consist of the tuple (X_i, Y_i, Z_i). This results in 3 checks per layer. We can combine them using the batch sumcheck protocol (Section 5.3). If we are doing multiple EC multiplications in parallel then these can be combined using Lagrange polynomials.

Acknowledgments. We would like to thank Arasu Arun and Lev Soukhanov for inspiring conversations on memory-checking and accumulation for GKR. We thank Shang Gao for pointing out several typos throughout our paper.

References

[AFK22] Thomas Attema, Serge Fehr, and Michael Klooß. "Fiat-Shamir Transformation of Multi-round Interactive Proofs". In: *TCC 2022, Part I*. Ed. by Eike Kiltz and Vinod Vaikuntanathan. Vol. 13747. LNCS. Springer, Heidelberg, Nov. 2022, pp. 113–142. DOI: 10.1007/978-3-031-22318-1_5.

[APPK24] Kasra Abbaszadeh, Christodoulos Pappas, Dimitrios Papadopoulos, and Jonathan Katz. *Zero-Knowledge Proofs of Training for Deep Neural Networks*. Cryptology ePrint Archive, Paper 2024/162. https://eprint.iacr.org/2024/162. 2024. URL: https://eprint.iacr.org/2024/162.

[AST23] Arasu Arun, Srinath Setty, and Justin Thaler. *Jolt: SNARKs for Virtual Machines via Lookups*. Cryptology ePrint Archive, Paper 2023/1217. https://eprint.iacr.org/2023/1217. 2023. URL: https://eprint.iacr.org/2023/1217.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. "Verifiable Delay Functions". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 757–788. DOI: 10.1007/978-3-319-96884-1_25.

[BC23] Benedikt Bünz and Binyi Chen. *ProtoStar: Generic Efficient Accumulation/Folding for Special Sound Protocols*. Cryptology ePrint Archive, Paper 2023/620. https://eprint.iacr.org/2023/620. 2023. URL: https://eprint.iacr.org/2023/620.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. "Recursive composition and bootstrapping for SNARKS and proof-carrying data". In: *45th ACM STOC*. Ed. by Dan Boneh, Tim Roughgarden, and Joan Feigenbaum. ACM Press, June 2013, pp. 111–120. DOI: 10.1145/2488608.2488623.

[BCGTV13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. "SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge". In: *CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 90–108. DOI: 10.1007/978-3-642-40084-1_6.

[BCLMS21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. "Proof-Carrying Data Without Succinct Arguments". In: *CRYPTO 2021, Part I*. Ed. by Tal Malkin and Chris Peikert. Vol. 12825. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 681–710. DOI: 10.1007/978-3-030-84242-0_24.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. "Recursive Proof Composition from Accumulation Schemes". In: *TCC 2020, Part II*. Ed. by Rafael Pass and Krzysztof Pietrzak.

Vol. 12551. LNCS. Springer, Heidelberg, Nov. 2020, pp. 1–18. DOI: 10.1007/978-3-030-64378-2_1.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves". In: *CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. LNCS. Springer, Heidelberg, Aug. 2014, pp. 276–294. DOI: 10.1007/978-3-662-44381-1_16.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture". In: *USENIX Security 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, Aug. 2014, pp. 781–796.

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. "Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments". In: *CRYPTO 2021, Part I*. Ed. by Tal Malkin and Chris Peikert. Vol. 12825. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 649–680. DOI: 10.1007/978-3-030-84242-0_23.

[BEGKN91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. "Checking the Correctness of Memories". In: *32nd FOCS*. IEEE Computer Society Press, Oct. 1991, pp. 90–99. DOI: 10.1109/SFCS.1991.185352.

[BFRSBW13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. *Verifying Computations with State (Extended Version)*. Cryptology ePrint Archive, Report 2013/356. https://eprint.iacr.org/2013/356. 2013.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. *Halo: Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Report 2019/1021. https://eprint.iacr.org/2019/1021. 2019.

[BMRS20] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. *Coda: Decentralized Cryptocurrency at Scale*. Cryptology ePrint Archive, Report 2020/352. https://eprint.iacr.org/2020/352. 2020.

[CBBZ22] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. *HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates*. Cryptology ePrint Archive, Report 2022/1355. https://eprint.iacr.org/2022/1355. 2022.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: *EUROCRYPT 2023, Part II*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14005. LNCS. Springer, Heidelberg, Apr. 2023, pp. 499–530. DOI: 10.1007/978-3-031-30617-4_17.

[CCDW20] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. *Reducing Participation Costs via Incremental Verification for Ledger Systems*. Cryptology ePrint Archive, Report 2020/1522. https://eprint.iacr.org/2020/1522. 2020.

[CDvGS03] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. "Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking". In: *ASIACRYPT 2003*. Ed. by Chi-Sung Laih. Vol. 2894. LNCS. Springer, Heidelberg, 2003, pp. 188–207. DOI: 10.1007/978-3-540-40061-5_12.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. "Practical verified computation with streaming interactive proofs". In: *ITCS 2012*. Ed. by Shafi Goldwasser. ACM, Jan. 2012, pp. 90–112. DOI: 10.1145/2090236.2090245.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. "Fractal: Post-quantum and Transparent Recursive Proofs from Holography". In: *EUROCRYPT 2020, Part I*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12105. LNCS. Springer, Heidelberg, May 2020, pp. 769–793. DOI: 10.1007/978-3-030-45721-1_27.

[CT10] Alessandro Chiesa and Eran Tromer. "Proof-Carrying Data and Hearsay Arguments from Signature Cards". In: *ICS 2010*. Ed. by Andrew Chi-Chih Yao. Tsinghua University Press, Jan. 2010, pp. 310–331.

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. "Cluster Computing in Zero Knowledge". In: *EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. LNCS. Springer, Heidelberg, Apr. 2015, pp. 371–403. DOI: 10.1007/978-3-662-46803-6_13.

[DNRV09] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. "How Efficient Can Memory Checking Be?" In: *TCC 2009*. Ed. by Omer Reingold. Vol. 5444. LNCS. Springer, Heidelberg, Mar. 2009, pp. 503–520. DOI: 10.1007/978-3-642-00457-5_30.

[EFG22] Liam Eagen, Dario Fiore, and Ariel Gabizon. *cq: Cached quotients for fast lookups*. Cryptology ePrint Archive, Report 2022/1763. https://eprint.iacr.org/2022/1763. 2022.

[EG23] Liam Eagen and Ariel Gabizon. *ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances*. Cryptology ePrint Archive, Paper 2023/1106. https://eprint.iacr.org/2023/1106. 2023. URL: https://eprint.iacr.org/2023/1106.

[GK22] Ariel Gabizon and Dmitry Khovratovich. *flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size*. Cryptology ePrint Archive, Report 2022/1447. https://eprint.iacr.org/2022/1447. 2022.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. "Delegating computation: interactive proofs for muggles". In:

40th ACM STOC. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 113–122. DOI: 10.1145/1374376.1374396.

[Hab22] Ulrich Haböck. *Multivariate lookups based on logarithmic derivatives*. Cryptology ePrint Archive, Report 2022/1530. https://eprint.iacr.org/2022/1530. 2022.

[KB20] Assimakis Kattis and Joseph Bonneau. *Proof of Necessary Work: Succinct State Verification with Fairness Guarantees*. Cryptology ePrint Archive, Report 2020/190. https://eprint.iacr.org/2020/190. 2020.

[KS23] Abhiram Kothapalli and Srinath Setty. *HyperNova: Recursive arguments for customizable constraint systems*. Cryptology ePrint Archive, Paper 2023/573. https://eprint.iacr.org/2023/573. 2023. URL: https://eprint.iacr.org/2023/573.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *CRYPTO 2022, Part IV*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13510. LNCS. Springer, Heidelberg, Aug. 2022, pp. 359–388. DOI: 10.1007/978-3-031-15985-5_13.

[PH23] Shahar Papini and Ulrich Haböck. *Improving logarithmic derivative lookups using GKR*. Cryptology ePrint Archive, Paper 2023/1284. https://eprint.iacr.org/2023/1284. 2023. URL: https://eprint.iacr.org/2023/1284.

[PK22] Jim Posen and Assimakis A. Kattis. *Caulk+: Table-independent lookup arguments*. Cryptology ePrint Archive, Report 2022/957. https://eprint.iacr.org/2022/957. 2022.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. "Complete Addition Formulas for Prime Order Elliptic Curves". In: *EUROCRYPT 2016, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. LNCS. Springer, Heidelberg, May 2016, pp. 403–428. DOI: 10.1007/978-3-662-49890-3_16.

[SAGL18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. *Proving the correct execution of concurrent services in zero-knowledge*. Cryptology ePrint Archive, Report 2018/907. https://eprint.iacr.org/2018/907. 2018.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. *Unlocking the lookup singularity with Lasso*. Cryptology ePrint Archive, Paper 2023/1216. https://eprint.iacr.org/2023/1216. 2023. URL: https://eprint.iacr.org/2023/1216.

[Tea22] Polygon Zero Team. *Plonky2: Fast Recursive Arguments with PLONK and FRI*. GitHub. 2022. URL: https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf.

[Val08a] Paul Valiant. "Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency". In: *TCC 2008*. Ed.

by Ran Canetti. Vol. 4948. LNCS. Springer, Heidelberg, Mar. 2008, pp. 1–18. DOI: 10.1007/978-3-540-78524-8_1.

[Val08b] Paul Valiant. "Testing symmetric properties of distributions". In: *40th ACM STOC*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 383–392. DOI: 10.1145/1374376.1374432.

[XZZPS19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: *CRYPTO 2019, Part III*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11694. LNCS. Springer, Heidelberg, Aug. 2019, pp. 733–764. DOI: 10.1007/978-3-030-26954-8_24.

[ZBKMNS22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. "Caulk: Lookup Arguments in Sublinear Time". In: *ACM CCS 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM Press, Nov. 2022, pp. 3121–3134. DOI: 10.1145/3548606.3560646.

[ZGKMR22] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. *Baloo: Nearly Optimal Lookup Arguments*. Cryptology ePrint Archive, Report 2022/1565. https://eprint.iacr.org/2022/1565. 2022.

A Full Memory-Proving Protocol

Special-sound Lookup-Powered Memory-Proving Protocol Π_{MP}. $T := |\mathsf{OM}|$, $k := |\mathsf{Rd}|$, $\ell := |R|$.

Prover $\mathsf{P}(\mathsf{OM}, \mathsf{Rd} = \mathsf{RS}\|W, \mathsf{Wr} = \mathsf{WS}\|R)$		**Verifier** $\mathsf{V}(\mathsf{OM}^{\mathsf{V}}, \mathsf{RS}, \mathsf{WS})$
Compute $\boldsymbol{b} \in \mathbb{F}^\ell, \mathbf{r} \in \mathbb{F}^\ell, \boldsymbol{m} \in \mathbb{F}^T$, $\mathbf{w} \in \mathbb{F}^\ell, \Delta \in \mathbb{F}^T$ such that: $\boldsymbol{b} := R.\boldsymbol{a} \quad \mathbf{r} := R.\boldsymbol{v}$ $\boldsymbol{m}_i := \sum_{j=1}^{\ell} \mathbb{1}(\mathbf{r}_j = \mathsf{OM}_i) \;\; \forall i \in [T]$ $\mathbf{w} := W.\boldsymbol{v} - R.\boldsymbol{v}$ $\Delta_i := \mathbf{w}_j$ if $i = \boldsymbol{b}_j \; \exists j \in [\ell]$, 0 otherwise. $\forall i \in [T]$		
Efficiently computes $\mathsf{NM} \leftarrow \Delta + \mathsf{OM}$	$\xrightarrow{R, W, \boldsymbol{m}, \mathsf{NM}}$	$R.\boldsymbol{a} \stackrel{?}{=} W.\boldsymbol{a} \quad \mathsf{Rd} := \mathsf{RS}\|W \quad \mathsf{Wr} := \mathsf{WS}\|R$
	$\xleftarrow{x_1, x_2, x_3}$	$x_1, x_2, x_3 \leftarrow\!\!\$\; \mathbb{F}^3$
Compute, on the fly, the linear combinations of the values in the tuples of Rd, Wr: $\mathsf{Rd}'_j \leftarrow \mathsf{Rd}_j.a + x_2 \cdot \mathsf{Rd}_j.v + x_3 \cdot \mathsf{Rd}_j.t \;\; \forall j \in [k]$ $\mathsf{Wr}'_j \leftarrow \mathsf{Wr}_j.a + x_2 \cdot \mathsf{Wr}_j.v + x_3 \cdot \mathsf{Wr}_j.t \;\; \forall j \in [k]$ $\mathbf{h}^{\text{perm}}_j := \frac{1}{x_1 + \mathsf{Rd}'_j} \;\; \forall j \in [k]$ $\mathbf{g}^{\text{perm}}_j = \frac{1}{x_1 + \mathsf{Wr}'_j} \;\; \forall j \in [k]$ $\mathbf{h}^{\text{ivlk}}_j := \frac{1}{x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{r}_j} \;\; \forall j \in [\ell]$ $\mathbf{g}^{\text{ivlk}}_i = \frac{\boldsymbol{m}_i}{x_1 + x_2 \cdot i + \mathsf{OM}_i} \;\; \forall i \in [T]$ $\mathbf{h}^{\text{mu}}_j := \frac{\mathbf{w}_j}{x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j} \;\; \forall j \in [\ell]$ $\mathbf{g}^{\text{mu}}_i = \frac{\Delta_i}{x_1 + x_2 \cdot i + \Delta_i} \;\; \forall i \in [T]$		$\boldsymbol{b}^{\mathsf{V}} := R.\boldsymbol{a} \quad \mathbf{r}^{\mathsf{V}} := R.\boldsymbol{v}$ $\mathbf{w} := W.\boldsymbol{v} - R.\boldsymbol{v} \quad \Delta := \mathsf{NM} - \mathsf{OM}^{\mathsf{V}}$ Similarly, compute the linear combinations on the fly: $\mathsf{Rd}'_j \leftarrow \mathsf{Rd}_j.a + x_2 \cdot \mathsf{Rd}_j.v + x_3 \cdot \mathsf{Rd}_j.t \;\; \forall j \in [k]$ $\mathsf{Wr}'_j \leftarrow \mathsf{Wr}_j.a + x_2 \cdot \mathsf{Wr}_j.v + x_3 \cdot \mathsf{Wr}_j.t \;\; \forall j \in [k]$
	$\xrightarrow{\mathbf{h}^{\text{perm}}, \mathbf{g}^{\text{perm}}, \mathbf{h}^{\text{ivlk}}, \mathbf{g}^{\text{ivlk}}, \mathbf{h}^{\text{mu}}, \mathbf{g}^{\text{mu}}}$	$\sum_{j=1}^{k} \mathbf{h}^{\text{perm}}_j \stackrel{?}{=} \sum_{j=1}^{k} \mathbf{g}^{\text{perm}}_j$ $\sum_{j=1}^{\ell} \mathbf{h}^{\text{ivlk}}_j \stackrel{?}{=} \sum_{i=1}^{T} \mathbf{g}^{\text{ivlk}}_i$ $\sum_{j=1}^{\ell} \mathbf{h}^{\text{mu}}_j \stackrel{?}{=} \sum_{i=1}^{T} \mathbf{g}^{\text{mu}}_i$ $\mathbf{h}^{\text{perm}}_j \cdot (x_1 + \mathsf{Rd}'_j) \stackrel{?}{=} 1 \;\; \forall j \in [k]$ $\mathbf{g}^{\text{perm}}_j \cdot (x_1 + \mathsf{Wr}'_j) \stackrel{?}{=} 1 \;\; \forall j \in [k]$ $\mathbf{h}^{\text{ivlk}}_j \cdot (x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{r}_j) \stackrel{?}{=} 1 \;\; \forall j \in [\ell]$ $\mathbf{g}^{\text{ivlk}}_i \cdot (x_1 + x_2 \cdot i + \mathsf{OM}_i) \stackrel{?}{=} \boldsymbol{m}_i \;\; \forall i \in [T]$ $\mathbf{h}^{\text{mu}}_j \cdot (x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j) \stackrel{?}{=} \mathbf{w}_j \;\; \forall j \in [\ell]$ $\mathbf{g}^{\text{mu}}_i \cdot (x_1 + x_2 \cdot i + \Delta_i) \stackrel{?}{=} \Delta_i \;\; \forall i \in [T]$ Updates $\mathsf{OM}^{\mathsf{V}} \leftarrow \mathsf{NM}$

B Subalgorithms for sublinear prover time

Given:
ρ_{acc} – random oracle for accumulation,
ck – commitment key,
$\mathsf{acc}.(x_1, x_2, \mu, \mathbf{g}, \mathsf{OM})$ – items accumulated up to $(k-1)$th round,
$C_{\mathbf{g}} = \mathsf{Commit}(\mathsf{ck}, \mathsf{acc}.\mathbf{g}), C_{\boldsymbol{m}} = \mathsf{Commit}(\mathsf{ck}, \mathsf{acc}.\boldsymbol{m})$ – accumulated commitments,
$\pi.(x_1, x_2, \mu, \mathbf{g}, \mathsf{OM})$ – proof items,
The items stored from the previous round:

- $\hat{\mathsf{acc}}.\mathbf{g}$ – item accumulated up to $(k-2)$th round,
- $\hat{\pi}.(\mathbf{g}, \varDelta)$ – proof items from the $(k-1)$th round,
- $r_{(k-1)}$ – random accumulation challenge from the $(k-1)$th round,
- $\hat{E}_{a2} = \mathsf{Commit}(\mathsf{ck}, \hat{\mathsf{acc}}.\mathbf{g} \circ \boldsymbol{i})$, $\hat{E}_{a3} = \mathsf{Commit}(\mathsf{ck}, \hat{\mathsf{acc}}.\mathbf{g} \circ \hat{\pi}.\mathsf{OM})$ – partial commitments from the $(k-1)$th round.

Goal: Compute a commitment to the degree-1 cross error term

$$\mathbf{e}_1 := \mathsf{acc}.\mathbf{g} \circ \pi.\mathsf{OM}' + \pi.\mathbf{g} \circ \mathsf{acc}.\mathsf{OM}' + \mathsf{acc}.\mu \cdot \pi.\boldsymbol{m} + \pi.\mu \cdot \mathsf{acc}.\boldsymbol{m}$$

DO:

$E_{a2} \leftarrow \hat{E}_{a2} + r_{(k-1)} \cdot \mathsf{Commit}(\mathsf{ck}, \hat{\pi}.\mathbf{g} \circ \boldsymbol{i})$,

$E_{a3} \leftarrow \hat{E}_{a3} + \mathsf{Commit}(\mathsf{ck}, \hat{\mathsf{acc}}.\mathbf{g} \circ \hat{\pi}.\varDelta + r_{(k-1)} \cdot \hat{\pi}.\mathbf{g} \circ \pi.\mathsf{OM})$,

$E_a \leftarrow \pi.x_1 \cdot C_{\mathbf{g}} + \pi.x_2 \cdot E_{a2} + E_{a3}$,

$\mathbf{e}_b \leftarrow \pi.\mathbf{g} \circ \mathsf{acc}.\mathsf{OM}'$,

$\mathbf{e}_c \leftarrow \mathsf{acc}.\mu \cdot \pi.\boldsymbol{m}$,

$E_1 \leftarrow E_a + \mathsf{Commit}(\mathsf{ck}, \mathbf{e}_b + \mathbf{e}_c) + C_{\boldsymbol{m}}$,

Use ρ_{acc} to generate the random accumulation for this round $r_{(k)}$.

END DO

Return:
E_1 – commitment to the degree-1 cross error term $\mathbf{e}_1$,
Items to store for the next round of accumulation:

- $\mathsf{acc}.\mathbf{g}$ – item accumulated up to $(k-1)$th round,
- $\pi.(\mathbf{g}, \varDelta)$ – proof item from this round,
- $r_{(k)}$ – the random accumulation challenge of this round,
- E_{a2}, E_{a3} – the partial commitments.

Fig. 1. Algorithm for computing and committing to the cross error term for $\mathbf{g}_i^{\mathrm{ivlk}} \cdot (x_1 + x_2 \cdot i + \mathsf{OM}_i) \stackrel{?}{=} \boldsymbol{m}_i \ \ \forall i \in [m]$ in time sublinear to T in the kth round of accumulation, where $T := |\mathsf{OM}|$. For simplicity, we refer to $\mathbf{g}^{\mathrm{ivlk}}$ by just $\mathbf{g}$ in the algorithm. OM' denotes the sum $(x_1 \cdot \mathbf{1}_T + x_2 \cdot \boldsymbol{i} + \mathsf{OM}_i)$, where $\boldsymbol{i} = [1, 2, \ldots, T]$ is the address/index vector for the memory.

Given:
r^* – sum of all random accumulation challenges up to the $(k-1)$th round,
$r_{(k)}$ – the random accumulation challenge of this round,
$\mathbf{R}$ – helper vector such that $\mathbf{R}_i = \sum_{q=1}^{j} r_{(q)}$, where j is the most recent round such that $\pi_{(j)}.\Delta_i \neq 0$ for the corresponding $i \in [T]$,
$\mathsf{acc}_{(k-1)}.\mathsf{OM}$ – the current accumulated OM,
$\pi_{(k)}.(\mathsf{OM}, \Delta)$ – memory and update vectors in the new proof.
Item stored from the previous round:
$\hat{\pi}_{(k-1)}.\Delta$ – update vector from the previous round.

Goal: Compute the updated accumulator item

$$\mathsf{acc}_{(k)}.\mathsf{OM} := \mathsf{acc}_{(k-1)}.\mathsf{OM} + r_{(k)} \cdot \pi_{(k)}.\mathsf{OM}$$

DO:

$r^* \leftarrow r^* + r_{(k)}$

for $i \in [T]$ s.t. $\pi_{(k-1)}.\Delta_i \neq 0$:

$\mathsf{acc}_{(k)}.\mathsf{OM}_i \leftarrow \mathsf{acc}_{(k-1)}.\mathsf{OM}_i +$
$(r^* - \mathbf{R}_i) \cdot (\pi_{(k)}.\mathsf{OM}_i - \pi_{(k-1)}.\Delta_i) + r_{(k)} \cdot \pi_{(k-1)}.\Delta_i$

$\mathbf{R}_i \leftarrow r^*$

end for

\\Whenever $\mathsf{acc}_{(k)}.\mathsf{OM}$ is needed for computation,
\\substitute it with $\mathsf{acc}_{(k)}.\mathsf{OM} + (r^*_{(k)} \cdot \mathbf{1}_T - \mathbf{R}) \circ \pi_{(k)}.\mathsf{OM}$

END DO

Return:
$\mathsf{acc}_{(k)}.\mathsf{OM}$ – updated accumulator item,
r^* – sum of all random accumulation challenges up to this round,
$\mathbf{R}$ – updated helper vector.
Item to store for the next round of accumulation:
$\pi_{(k)}.\Delta$ – update vector of this round.

Fig. 2. Algorithm for accumulating OM in the kth round of accumulation in time sublinear to T, where $T := |\mathsf{OM}|$. Since evaluation/update at any accumulated memory index is delayed to when a nonzero change occurs at that index of $\pi.\mathsf{OM}$, only 2ℓ updates need to be done per round.